Adobe

Adobe Summit

LAB

WORKBOOK

L713 – Turbocharge Front-end Applications with AEM Headless

Gilles Knobloch
Sean Steimer
Mathias Siegel
Adobe

1.2

Adobe Summit 2023

Lab overview 2
Introduction 3
Tools for this Lab 3
Tools Setup 3
How to do individual exercises 7
Exercise 1 – Author Content via UI and API 8
Exercise 1.1 – Creating Content Fragment Models 8
Exercise 1.2 – Using Content Fragments Admin Console 10
Exercise 1.3 – Creating Content Fragments 11
Exercise 1.4 – Creating Content Fragments and Models via API 13
Exercise 2 – Deliver content via GraphQL 24
Exercise 2.1 – Create GraphQL endpoint 24
Exercise 2.2 – Explore GraphiQL 25
Exercise 2.3 – Build Sample Queries 27
Exercise 2.4 – Building persisted queries 50
Exercise 3 – Build an application consuming AEM content 56
Objective 56
Lesson context 56
Exercise 3.1 – Understand the structure of the application 56
Exercise 3.2 – Extend the application to consume the content previously created 57
Exercise 3.3 – Allow the application to be edited in-context 63
Next steps 70
Thank you 70
Additional resources 70

Lab overview

In this lab, you'll get hands on with AEM Sites Headless, and learn how it can be used to create, manage, and deliver content headless content to enable exceptional experiences. You will learn:

- How to author content for headless, first using the UI, and then via API
- How to deliver that content with GraphQL
- How to use the AEM Headless SDK to integrate that content into a React Application
- How to use the new Universal Editor to edit the application in context

Introduction

Objective

1. Learn what tools you will need
2. Setup your training machine for the exercises

Tools for this Lab

Your lab machine is already pre-loaded with several things you will need to complete the lab:

- Google Chrome
- Visual Studio Code
- Thunder Client VS Code Extension
- AEM Content packages for use during the exercises
- A git repository checkout, containing:
    - The react app we will use for exercise 3
    - A set of bookmarks for you to import
    - Thunder Client Collection and Environment files

In addition, you have a dedicated AEM Sandbox for the lab. This sandbox has WKND and other content you will need to complete the lab pre-installed.

Tools Setup

Updating the git repo

Before you do anything else, open a terminal to the git repo and execute `git pull` to ensure everything is up to date.

Identifying your environment

Click the *Cloud Manager* bookmark. You will be redirected to the login page.

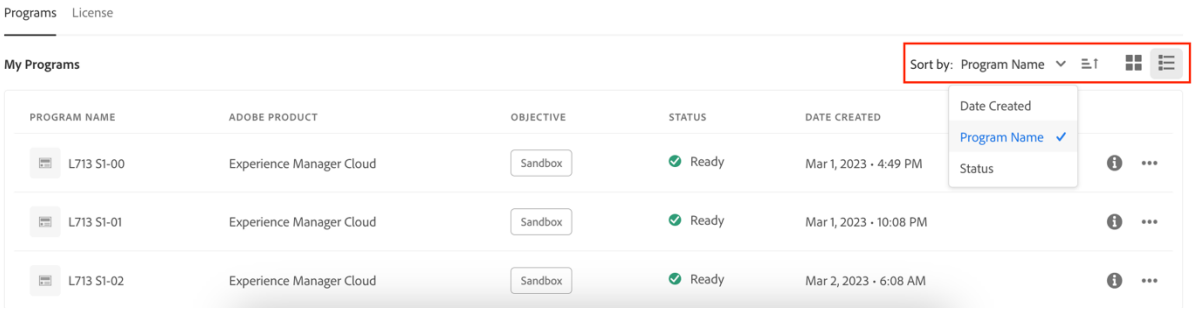
Username	L713+S**{SessionNumber}**-**{SeatNumber}**@summitlab.us
Password	*Provided by lab instructor*

- Replace **{SessionNumber}** by **1** or **2**, depending which session you are part of. Lab instructor is going to mention it.
- Replace **{SeatNumber}** by your lab seat number.

Example username: *L713+S1-14@summitlab.us* for session 1, seat 14.

You are redirected to Cloud Manager listing all the programs.

Sort the program by names, ascending order and switch to list view



Scroll down to identify your program.
Make sure you select the right session id!

Importing Bookmarks
Open the provided bookmarklets.html with Visual Studio Code.

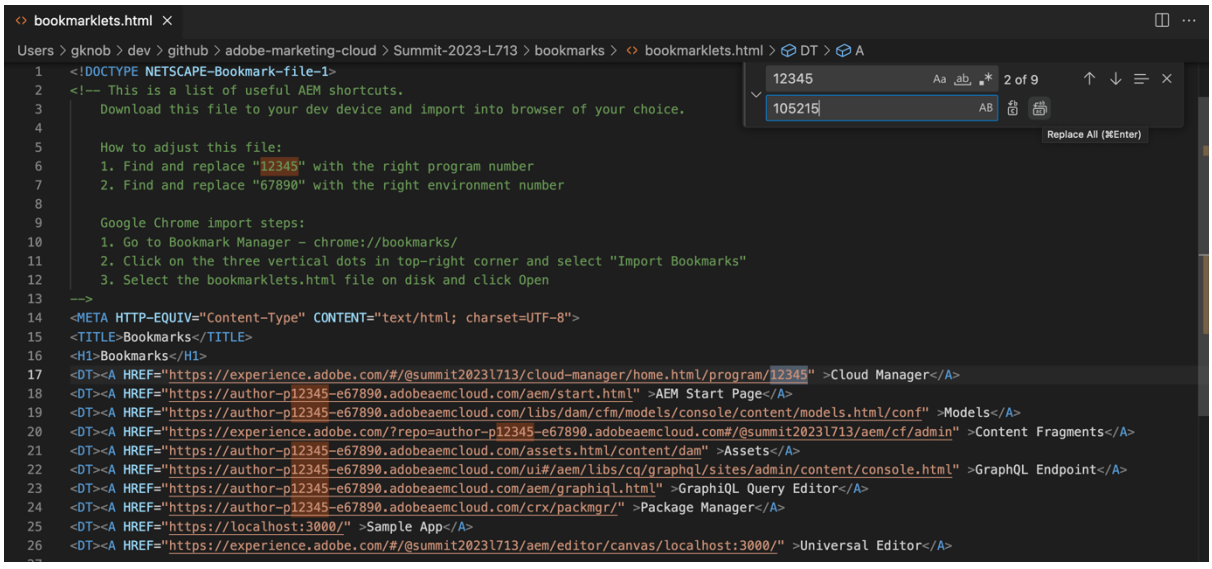
Before you can import the bookmarks into chrome, some adjustments must be made to ensure the bookmarks go to the correct environment and program for your lab sandbox.

The file contains instructions for doing, which are repeated below for ease of use.

Your program and environment number can be found in the L713-Users-Mapping.xlsx file. Simply use the appropriate program/environment for your lab seat number.

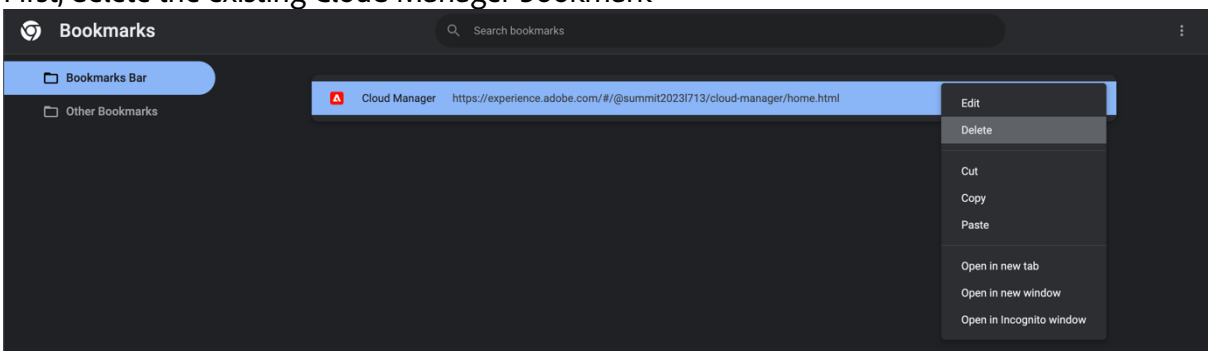
How to adjust this file:

1. Find and replace "12345" with the right program number
2. Find and replace "67890" with the right environment number



Google Chrome import steps:

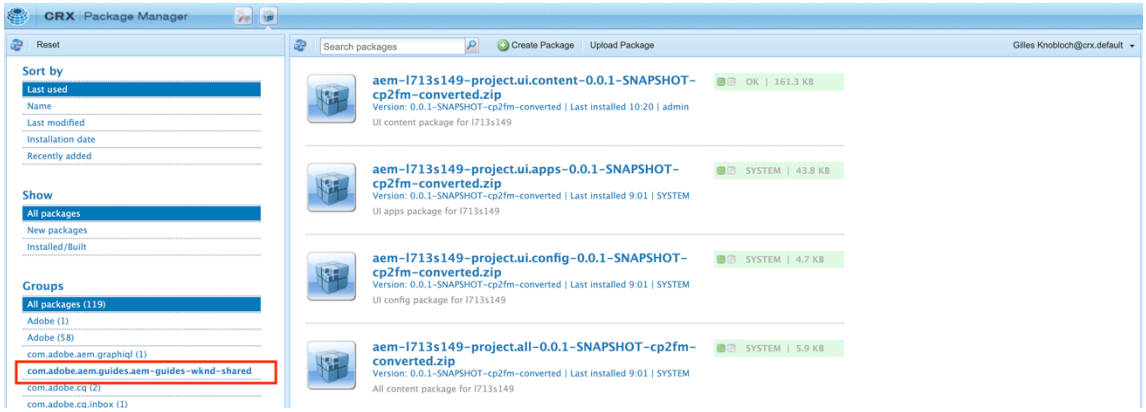
1. Go to Bookmark Manager - chrome://bookmarks/
2. First, delete the existing Cloud Manager bookmark



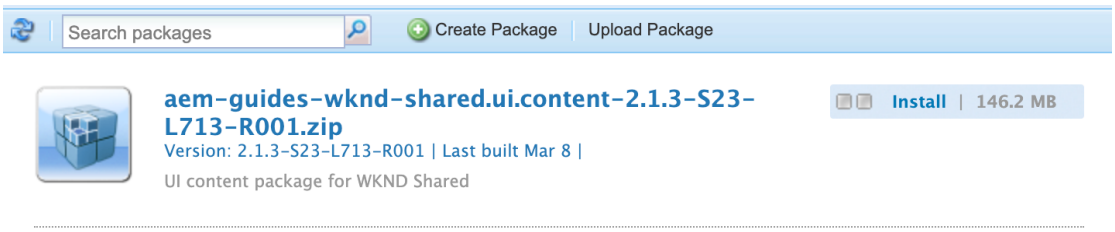
3. Click on the three vertical dots in top-right corner and select "Import Bookmarks"
4. Select the bookmarklets.html file on disk and click Open

Sample Content

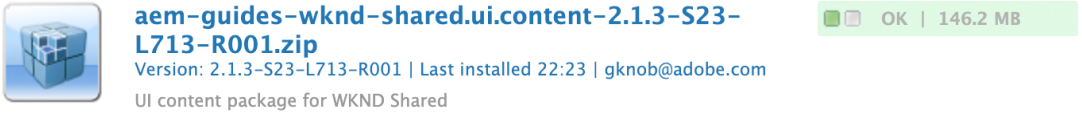
Open the *Package Manager* bookmark, sign-in with Adobe if needed, and look for *com.adobe.aem.guides.aem-guides-wknd-shared* in the list of package groups.



If not present, upload the *aem-guides-wknd-shared.ui.content-2.1.3-S23-L713-R001.zip* package from the Summit Lab resources folder.



Install that package if required:



Setting up your environment

Click the *Cloud Manager* bookmark. That one should now bring you to the right program id.

From here, access the developer console by clicking the three dots next to your environment and selecting *Developer Console*.

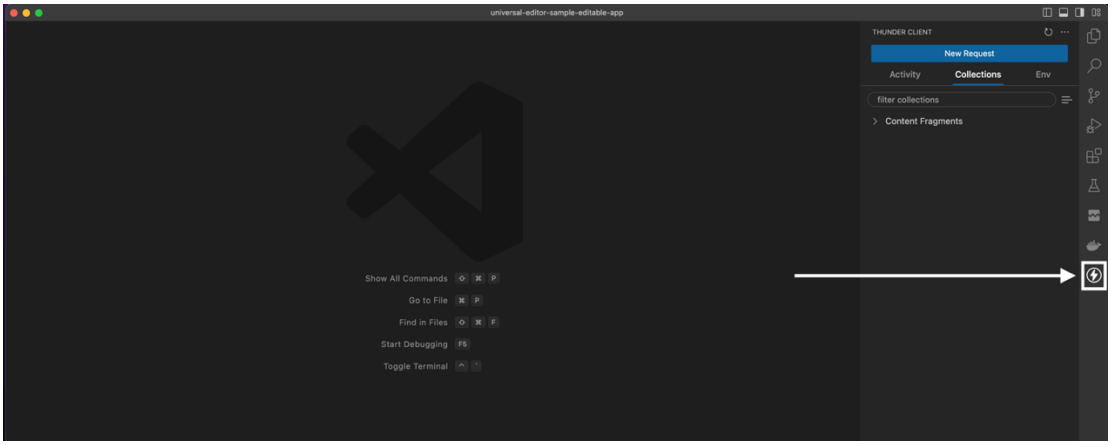
Tip: You may need to logout and log back in, or use an incognito window, for developer console to work

---

From there, click Integrations -> Local Token -> Get Local Development Token

Copy the value for the access token. Save this value somewhere for now, as we will use it in the next steps.

Now open Visual Studio Code and access Thunder Client by clicking the icon in the sidebar



Click on Collection, then on the menu (3 lines next to filter collections input box). Select Import, and browse to the collection file in the thunder-client folder.

Now Click on Env, then on the menu. Select Import, and browse to the environment file in the thunder-client folder.

Open the env:

Paste the access token we acquired earlier from developer console into the TOKEN field.

In the URL Field, paste the url for your sandbox (ensure it does not have a trailing /)

Save the environment File.

You can test that this is working by going to Collections -> Content Fragments. -> Exercise 1 > List Models. Click Send and ensure it returns a 200 response code.

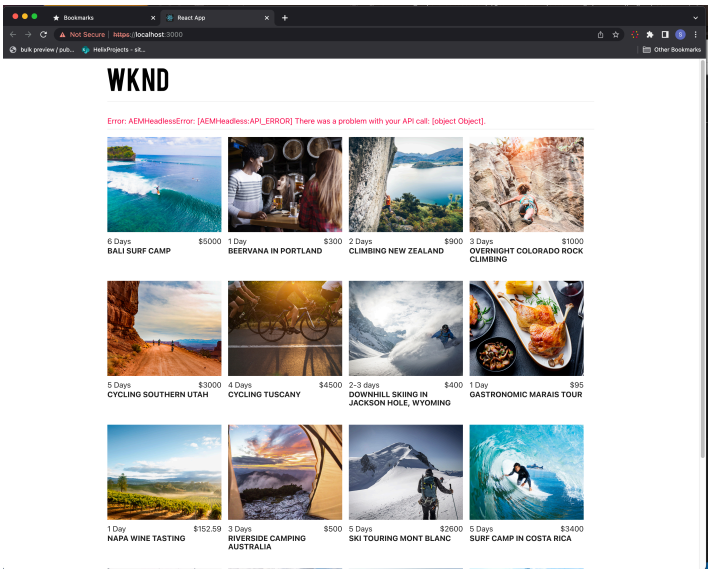
Finally, open the react-app folder in VS Code. Navigate to the .env file and open it.

For the REACT_APP_HOST_URI and REACT_APP_PUBLISH_URI, you can use the same find/replace steps you used on the bookmarks file to update the program and environment numbers.

For the REACT_APP_SERVICE_TOKEN, simple paste in the token value we acquired earlier.

Save the file and from a terminal, execute `yarn start`. The app should open automatically on https://localhost:3000.

At this point you should see something like this (the error is expected and will be resolved as we go through the lab)



Tip: If you get a warning that the connection is not private click Advanced -> "proceed to localhost (unsafe)"

How to do individual exercises

The exercises in this lab follow a narrative and can be done seamlessly in order. However, it is also possible to skip some exercises or jump directly to an exercise.

- The result of each exercise is available as a content packages which can be installed to your AEM Sandbox if you get behind or skip some steps.
- The result of exercise 3 can be retrieved by checking out the associated branch from git

Exercise 1 – Author Content via UI and API

Objective

1. Learn about the Content Fragments Admin Console
2. Learn how to create and edit content fragment models
3. Learn how to create and edit content fragments
4. Learn how to use AEM APIs to manage content fragments and models

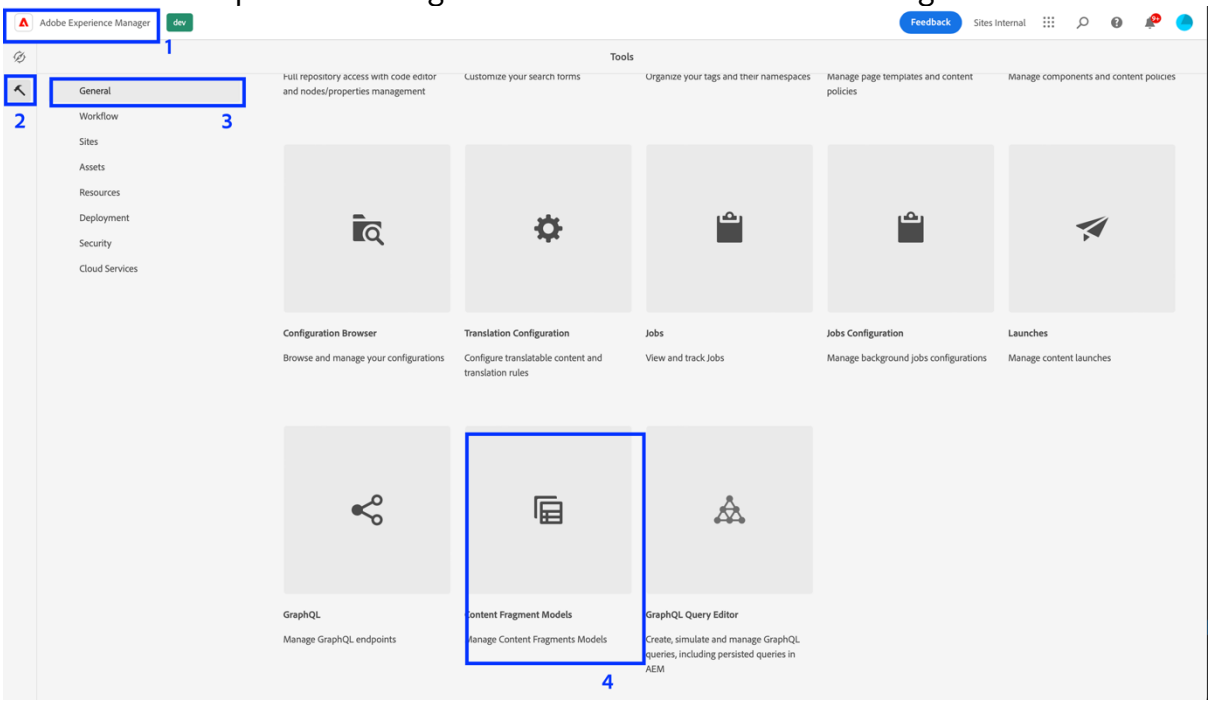
Lesson context

In this lesson, we will create the content to form the basis of the WKND Events application. Your AEM sandbox should already have the WKND Shared code and content installed, which we will build upon to create our events content.

Exercise 1.1 – Creating Content Fragment Models

Navigate to the AEM Start Page

Click on Adobe Experience Manager -> Tools -> General -> Content Fragment Models



Navigate into the WKND Shared Folder, then click the "Create" button in the upper right to create a new model.

Set the Model Title to "Event" and click **Create**.

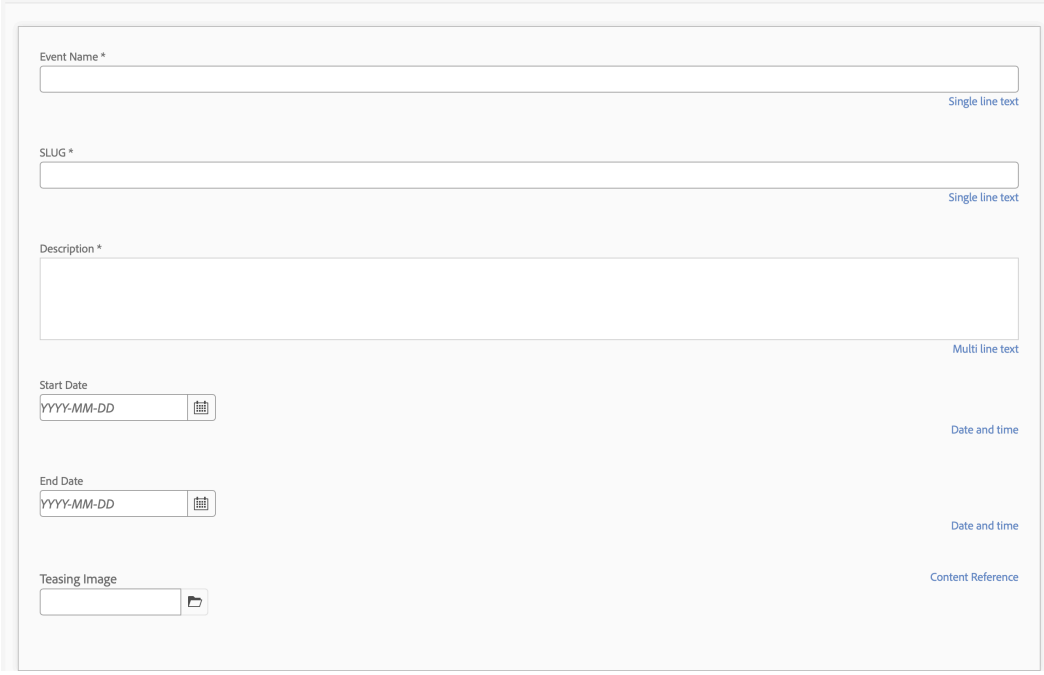
Once created, **Open** the model.

Add the following fields to the model:

Data Type	Field Label	Additional Attributes
Single line text	Event Name	• Required
Single line text	SLUG	• Required • Unique
Multi line text	Description	• Default Type. = "Rich text" • Required
Date and time	Start Date	• Type = "Date" • Placeholder = "YYYY-MM-DD"
Date and time	End Date	• Type = "Date" • Placeholder = "YYYY-MM-DD"
Content Reference	Teasing Image	• Accept only specified Types = "Image" • Show Thumbnail • Root Path = "/content/dam/wknd-shared/en"

Tip: The Property Name field affects the GraphQL query used for delivery, which you will create in Lesson 3. Please take care that these are correct.

Your model should now look like this:

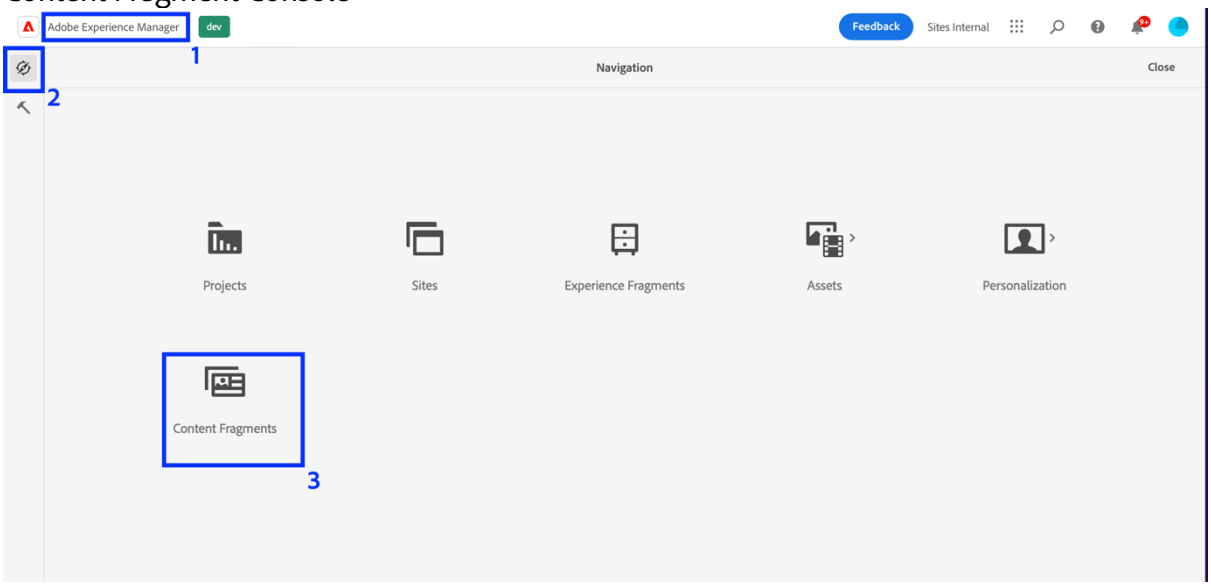


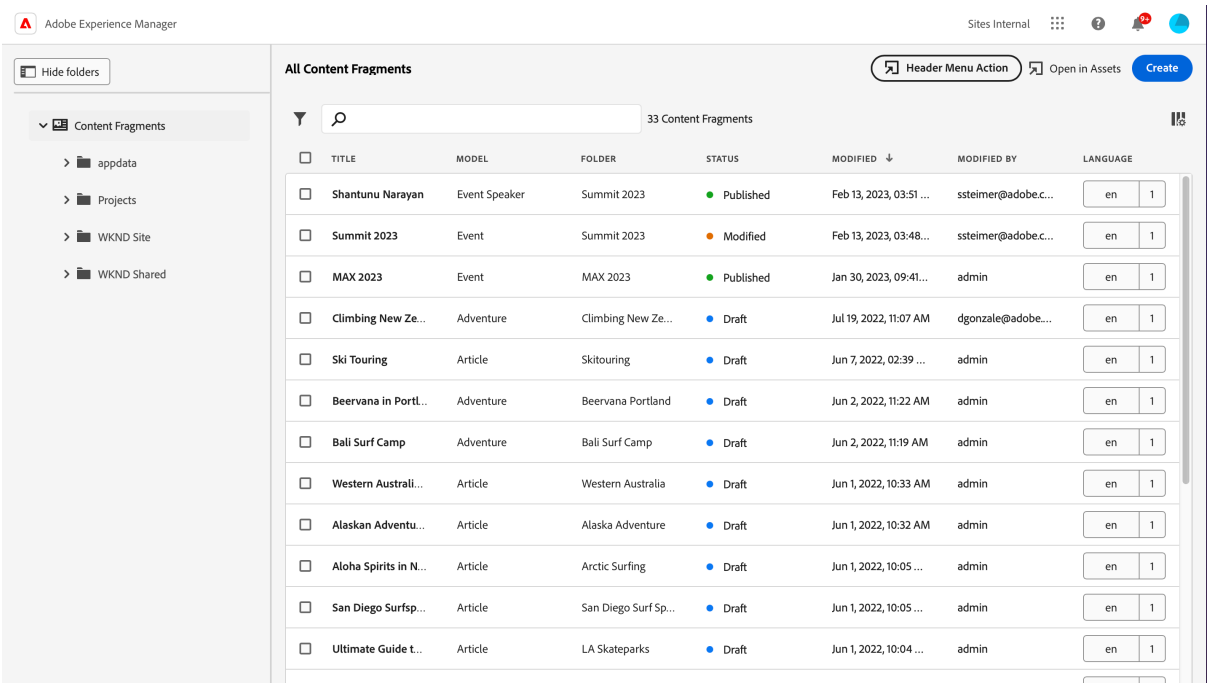
Click the **Save** button in the upper right to save your model.

Exercise 1.2 – Using Content Fragments Admin Console

Navigate to your AEM Sandbox.

Click on Adobe Experience Manager -> Navigation -> Content Fragments to access the Content Fragment Console





Familiarize yourself with the console by:

- navigating the folder structure in the left rail
    - try browsing to WKND Shared/English/Adventures)
- searching for fragments using the search box
    - try searching for the term “skiing”
- sorting fragments by click on the column headers
    - try sorting by the Modified date
- click on the title of a fragment to open it in the editor

Exercise 1.3 – Creating Content Fragments

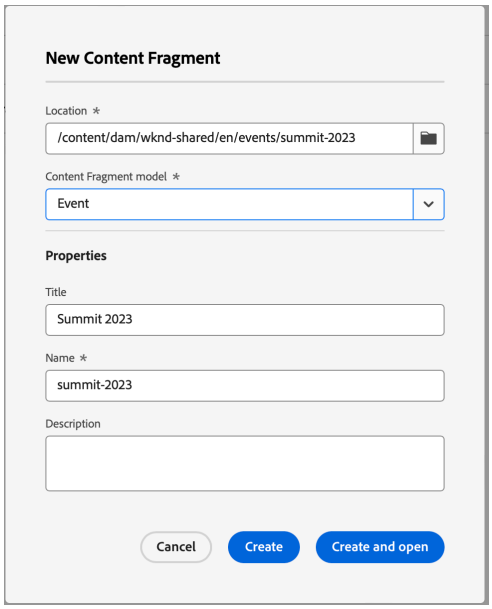
From the Content Fragment Console, click the "Create" button in the upper right.

Choose "/content/dam/wknd-shared/en/events/summit-2023" as the Location

Choose "Event", the model we created previously, as the Model

Set the Title to "Summit 2023"

Click the "Create and Open" button



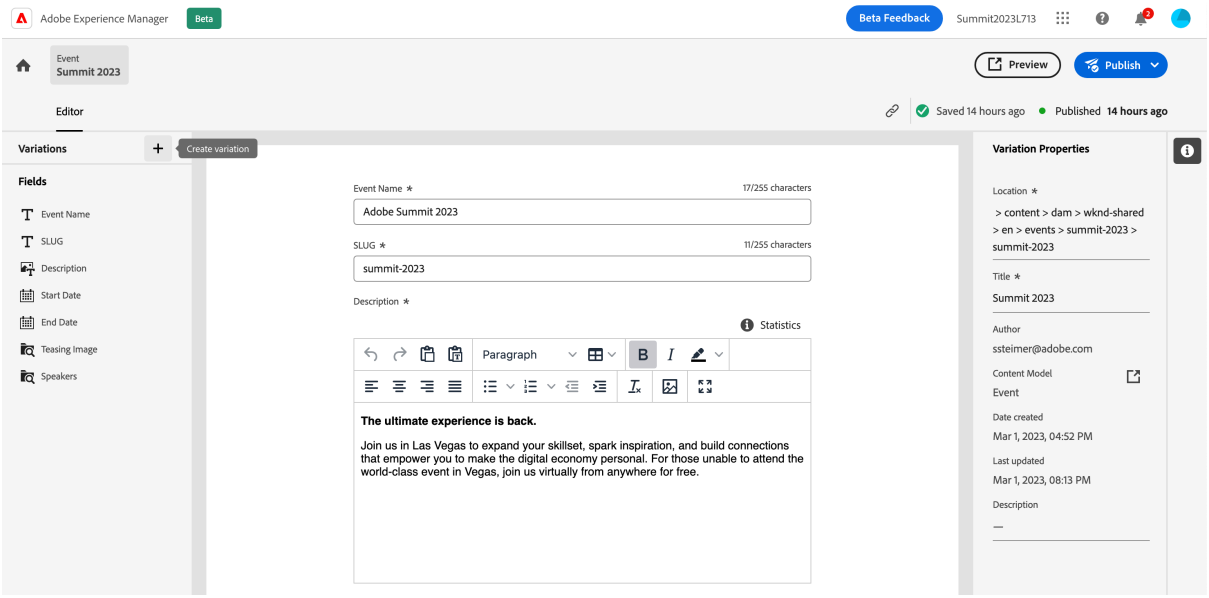
Set the Event Name, Slug, Description, Start and End Date, and Teasing Image.

The following values can be copy/pasted in:

Field Label	Value
Event Name	Adobe Summit 2023
SLUG	summit-2023
Description	**The ultimate experience is back.** Join us in Las Vegas to expand your skillset, spark inspiration, and build connections that empower you to make the digital economy personal. For those unable to attend the world-class event in Vegas, join us virtually from anywhere for free.
Start Date	2023-03-21
End Date	2023-03-23
Teasing Image	/content/dam/wknd-shared/en/events/summit-2023/summit-2023.png

Tip: The Content Fragment editor will auto-save as you make changes

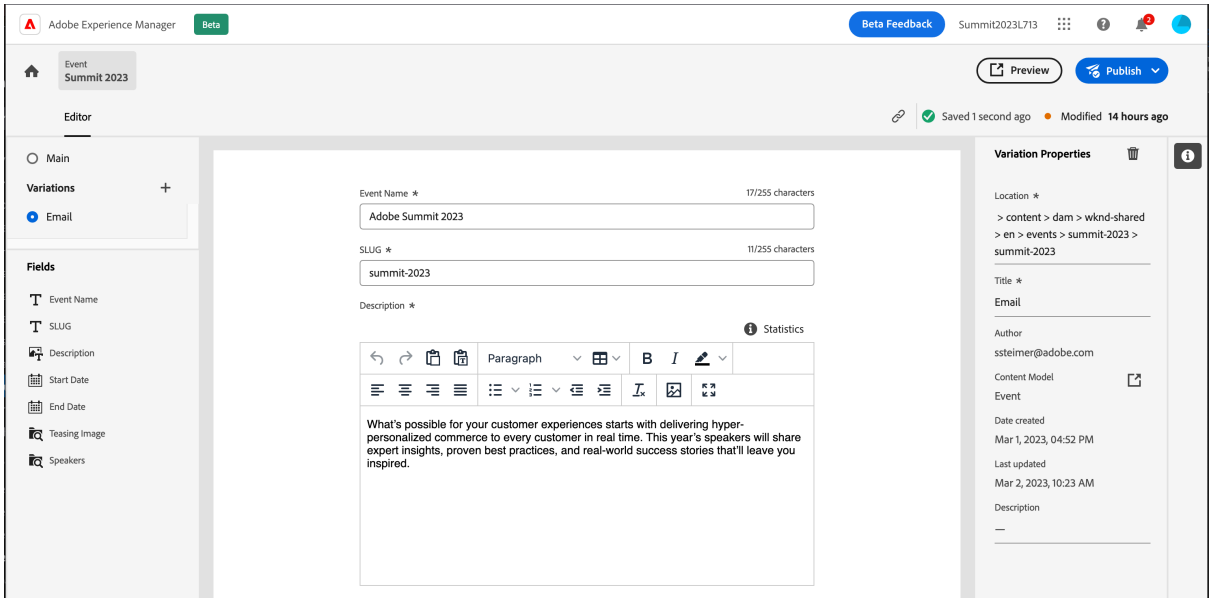
While we are here, lets also create a variant. Suppose we want to also use this fragment for an email campaign. Start by create a new variation, and name it “email”



You’ll notice when we first creat the variation, it has all the same data as main.

Now update the Description field to use the following text:

> *What’s possible for your customer experiences starts with delivering hyper-personalized commerce to every customer in real time. This year’s speakers will share expert insights, proven best practices, and real-world success stories that’ll leave you inspired.*



We can easily switch between these 2 variations, and make changes as needed. We'll see later how we can deliver different variations via graphQL.

Finally, repeat the steps above to create an Event fragment for Adobe Max in the MAX 2022 folder.

Field Label	Value
Event Name	Adobe MAX 2022
SLUG	max-2022
Description	With 200+ sessions and hundreds of inspiring speakers, MAX 2022 was an amazing event. You can still experience this global celebration of creativity by viewing the best of the sessions on demand.
Start Date	2022-10-18
End Date	2022-10-22
Teasing Image	/content/dam/wknd-shared/en/events/max-2022/max-2022.png

Exercise 1.4 – Creating Content Fragments and Models via API

Open Visual Studio Code and open the Thunder Client Plugin

Exploring the Models and Fragments Created

Navigate to the List Models API Request

Click Send to execute the request. You should see a response something like this:

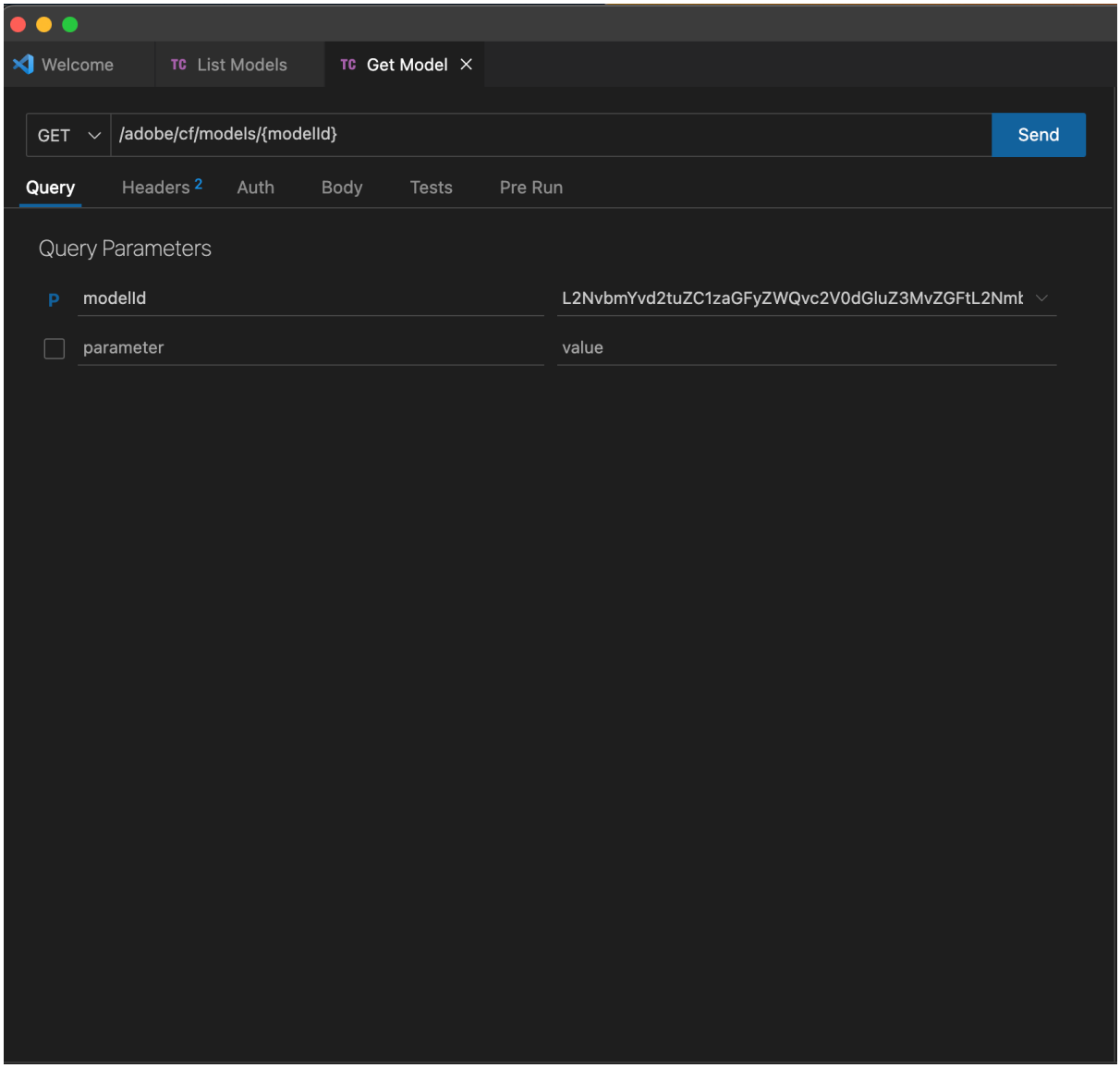
```
{
   "items": [
      { ... },
      { ... },
      {
         "locked": true,
         "id": "L2NvbmYvd2tuZC1zaGFyZWQvc2V0dGluZ3MvZGFtL2NmbS9tb2RlbHMvZXZlbnQ",
         "name": "Event",
         "description": "",
         "lastModified": 1676586355606,
         "lastModifiedBy": "ssteimer@adobe.com",
         "status": "enabled",
         "fields": [
           {
             "name": "eventName",
             "label": "Event Name",
             "required": true,
             "type": {
               "id": "text",
               "metadata": {
                 "maxLength": 255
               }
             },
             "multiple": false
           },
           ...
         ]
      },
      ...
   ]
}
```


Take a minute to familiarize yourself with the response payload. You should see all the fields we created in the Event model, as well as all other Content Fragment Models available in your AEM Sandbox.

Now find the Event model by hitting Cmd+f and search for "Event" and copy the id. It's value should start with L2N...

Next, go to the *Get Model* API request.

Paste the value you copied for the model Id into the *modelId* Query Parameter and click **Send**.



Next, go to the *List Content Fragments* API request.

Tip: You'll notice the path parameter in this request. This can used to filter for different paths, or removed entirely to list all fragments.

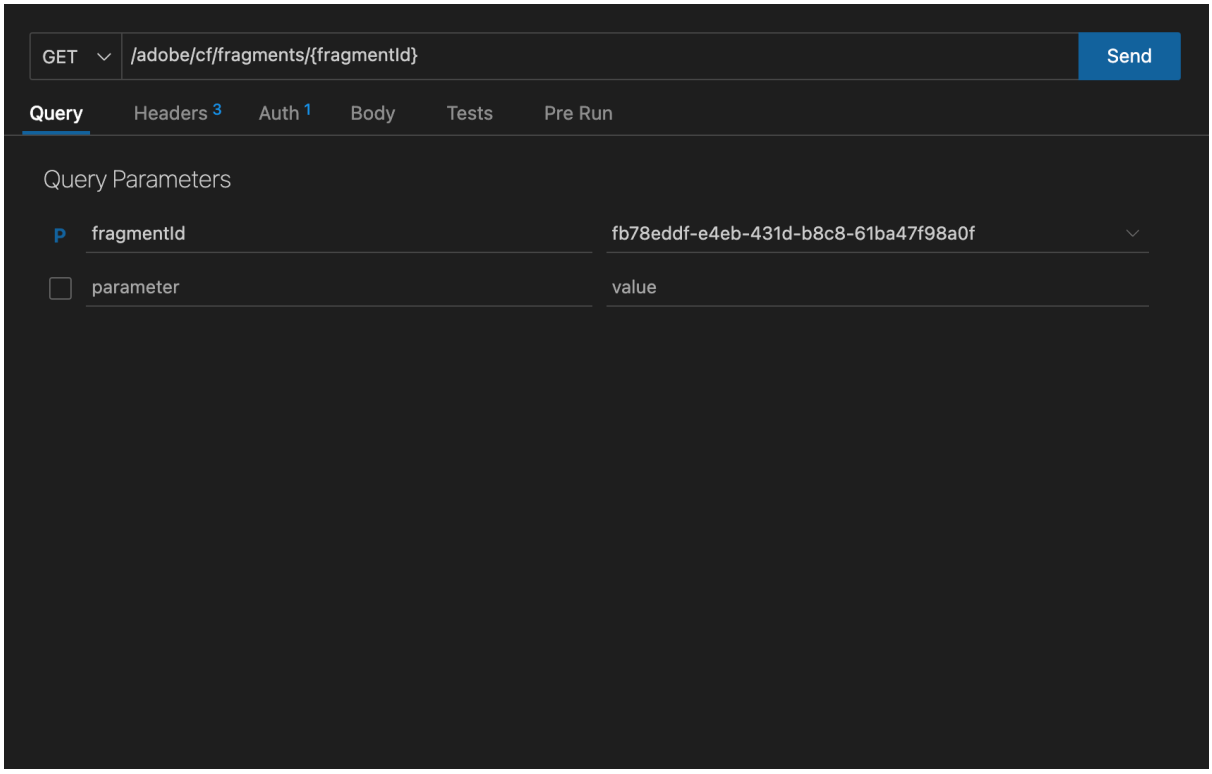
Click **Send** to execute the request. You should see a response like this, containing the 2 events we already created.

```
{
   "items": [
      { ... },
      { ... },
      {
         "title": "MAX 2022",
         "description": "",
         "model": {
```

```
            "path": "/conf/wknd-shared/settings/dam/cfm/models/event",
            "title": "Event"
          },
          "metadata": {},
          "path": "/content/dam/wknd-shared/en/events/max-2022/max-2022",
          "id": "830dee19-4f87-4150-991a-690d43b630f8",
          "created": {
            "at": "2023-02-17T11:10:06.560Z",
            "by": "gknob@adobe.com"
          },
          "modified": {
            "at": "2023-02-17T11:04:30.116Z",
            "by": "gknob@adobe.com"
          },
          "published": {
            "at": "2023-02-17T11:04:34.843Z",
            "by": "gknob@adobe.com"
          },
          "status": "PUBLISHED",
          "elements": {
            "eventName": "Adobe MAX 2022",
            "description": "<p>Example text</p>",
            "slug": "max-2022",
            "eventStart": "2022-10-18",
            "eventEnd": "2022-10-20",
            "teasingImage": "/content/dam/wknd-shared/en/events/max-2022/max-
2022.png"
          },
          "references": {
            "teasingImage": {
              "name": "max-2022.png",
              "size": 970813,
              "mimeType": "image/png",
              "width": 2000,
              "height": 876,
              "id": "72c2af1f-0ea4-410d-979f-aa94e0c6cd3e",
              "path": "/content/dam/wknd-shared/en/events/max-2022/max-2022.png",
              "created": {
                "at": "2023-02-17T11:10:06.616Z",
                "by": "gknob@adobe.com"
              },
              "status": "PUBLISHED"
            }
          },
          "variations": {}
        },
        ...
    ]
}
```


Copy the Id from one of the events we created.

Finally, navigate to the *Get Content Fragment by ID* API request. Paste the Id into the fragmentId query parameter and click **Send**.



Creating and Updating Models

Navigate to the *Create Speaker Model* API request.

Familiarize yourself with the request body:

```
{
  "configurationFolder": "/conf/wknd-shared",
  "name": "Event Speaker",
  "description": "",
  "status": "enabled",
  "fields": [
        {
          "name": "name",
          "label": "Name",
          "required": true,
          "type": {
            "id": "text",
            "metadata": {
              "maxLength": 255
            }
          },
          "multiple": false
        },
        {
          "name": "title",
          "label": "Title",
          "required": true,
          "type": {
            "id": "text",
            "metadata": {
              "maxLength": 255
            }
          },
          "multiple": false
```

```
        },
        {
          "name": "profilePicture",
          "label": "Profile Picture",
          "required": true,
          "type": {
            "id": "content-reference",
            "metadata": {
              "allowedContentTypes": [
                "image"
              ]
            }
          },
          "multiple": false
        }
      ]
}
```

Tip: To understand what metadata is available to be set on each field type, use the Get Data Types *API request.*

When you execute this request, it will return 201 Created. If you look at the response headers, you will see a Location header, e.g.:

```
Location:
/adobe/cf/models/L2NvbmYvd2tuZC1zaGFyZWQvc2V0dGluZ3MvZGFtL2NmbS9tb2RlbHMvZXZlbnQtc3BlYWtlci10ZXN0
```

The last portion of that Location (L2N…) constitutes the Model ID.

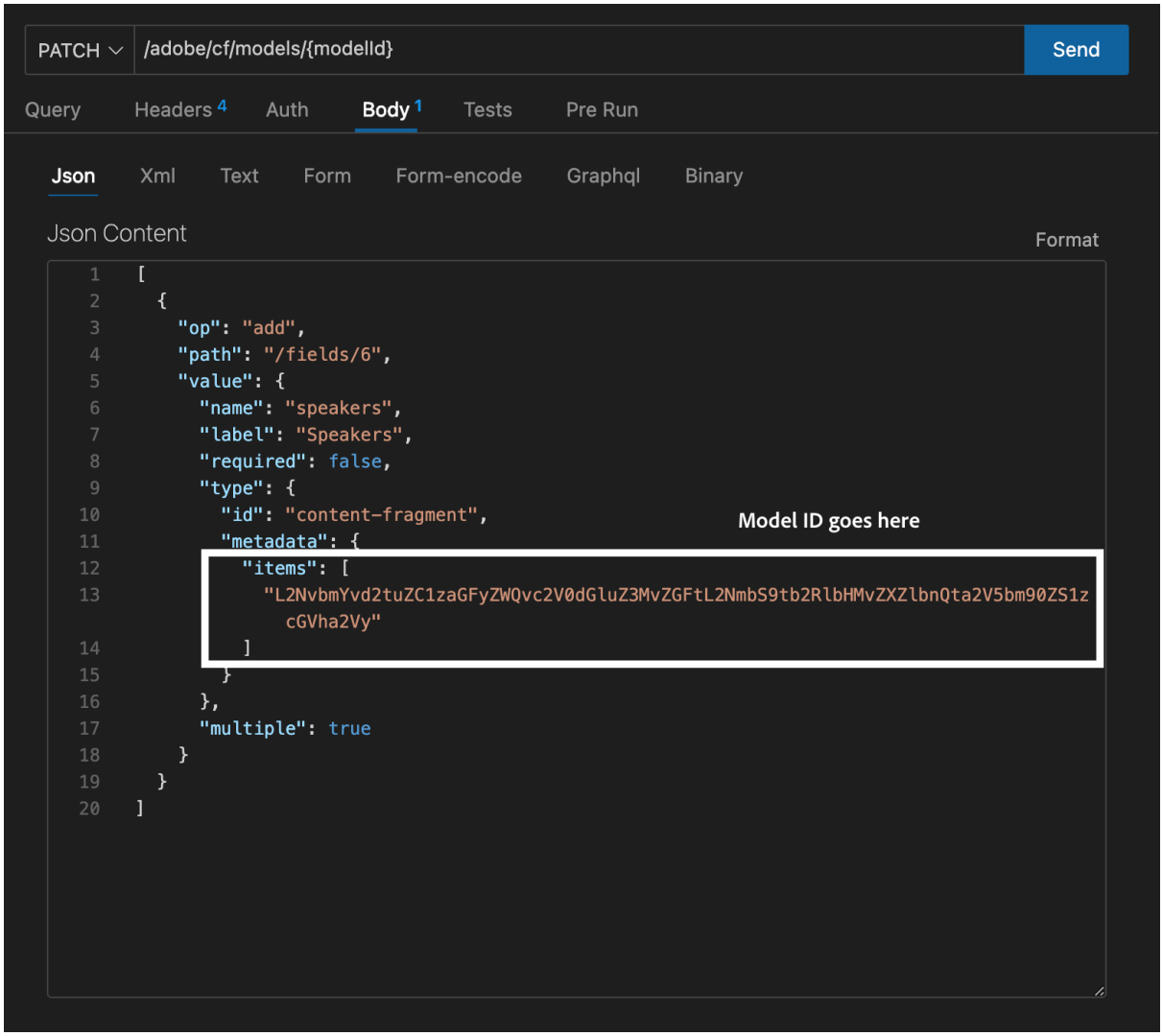
Next, execute the *Create Keynote Speaker Model* API request.

Tip: To make the Model IDs easier to find, leave the api responses open for later reference. If you lose track of these model IDs, you can use List Models API from earlier to find them again.

Finally, we need to update the Event Model to reference the Event Speaker. The *Update Event Model API* request serves as the baseline to do this, but requires us to paste in a few values:

- First, in the *Query* Tab, add the model Id of the Event model.
- Next, in the Body tab, adjust request body by paste in the Model ID for the Event Speaker model we just created.

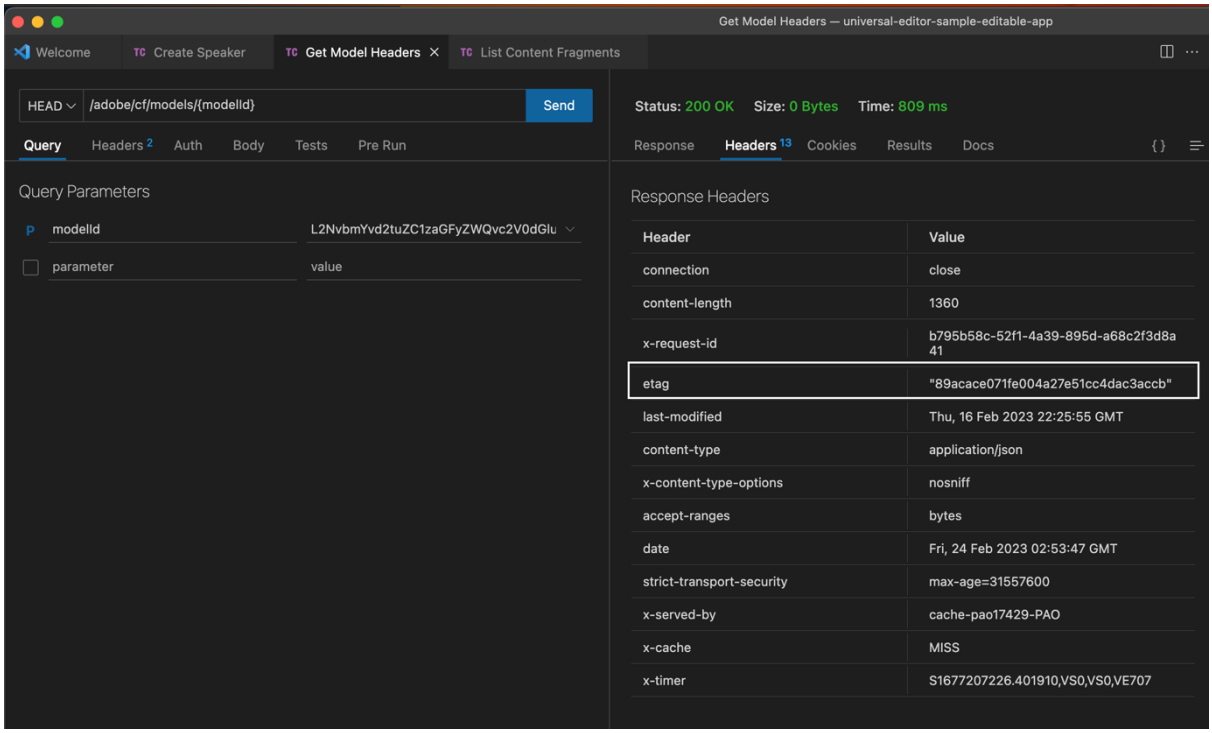
Note: for now, we only want to reference Event Speaker, not Keynote Speaker. The Event model will be updated again to add Keynote Speaker as part of exercise 2.

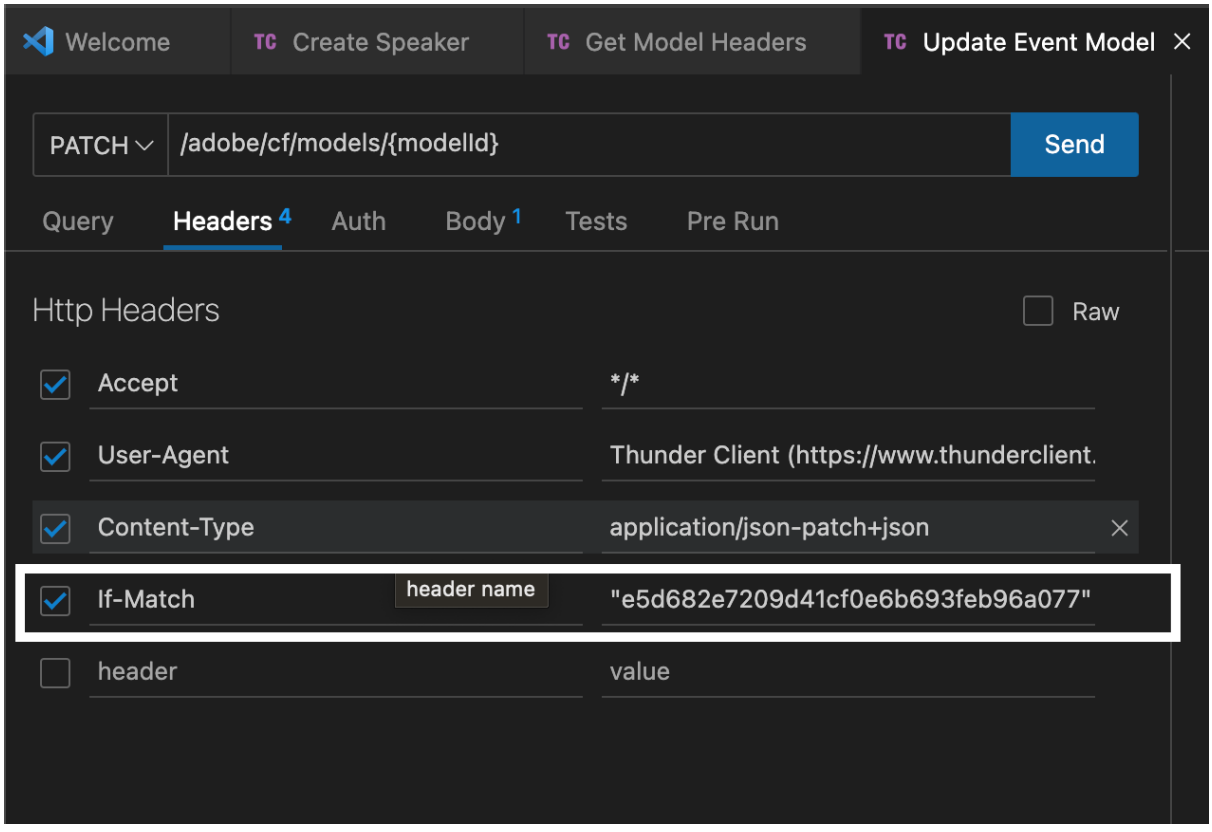


Finally, in the Request Headers, we need to replace the value for the If-Match Header.

Tip: The IF-Match header acts as a validation check Because the request is a Patch, and contains only the changes we are making, not the full model definition, we need to pass this value to ensure the current model definition is consistent with the definition we want to update.

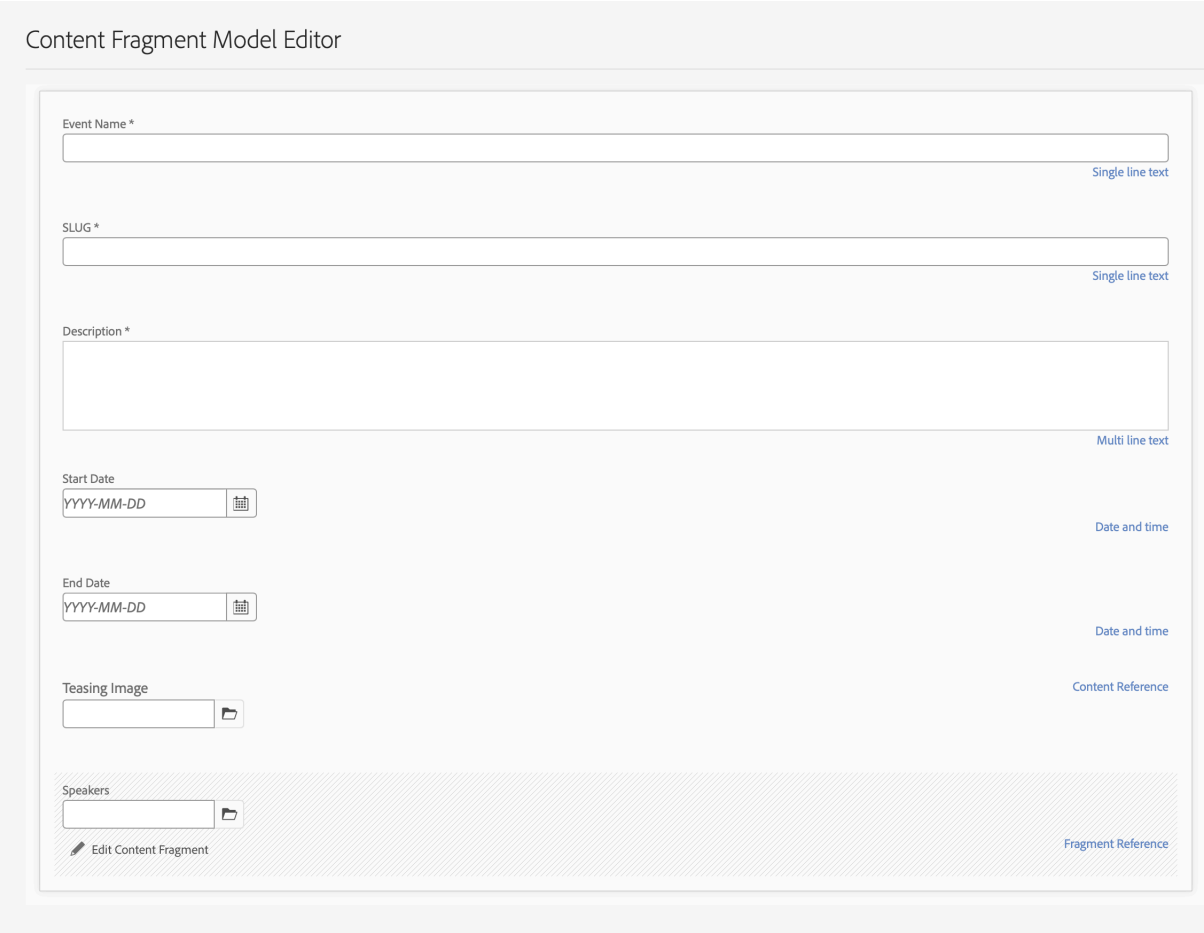
The correct value for this can be retrieved from the eTag response header by using the Get Model Headers API Request.





Once you've done that, click Send to execute the API request to Update the Event Model with the Speakers field.

You can validate this is successful either by looking at the model in the UI or re-executing the get Model API request and checking the fields in the response.



Creating and Updating Fragments

In the Exercise 1 Collection, you'll see a number of requests for creating speakers. First, take a look at the request body for creating an Event Speaker (Shantanu Narayen) and a Keynote Speaker (Aaron Sorkin)

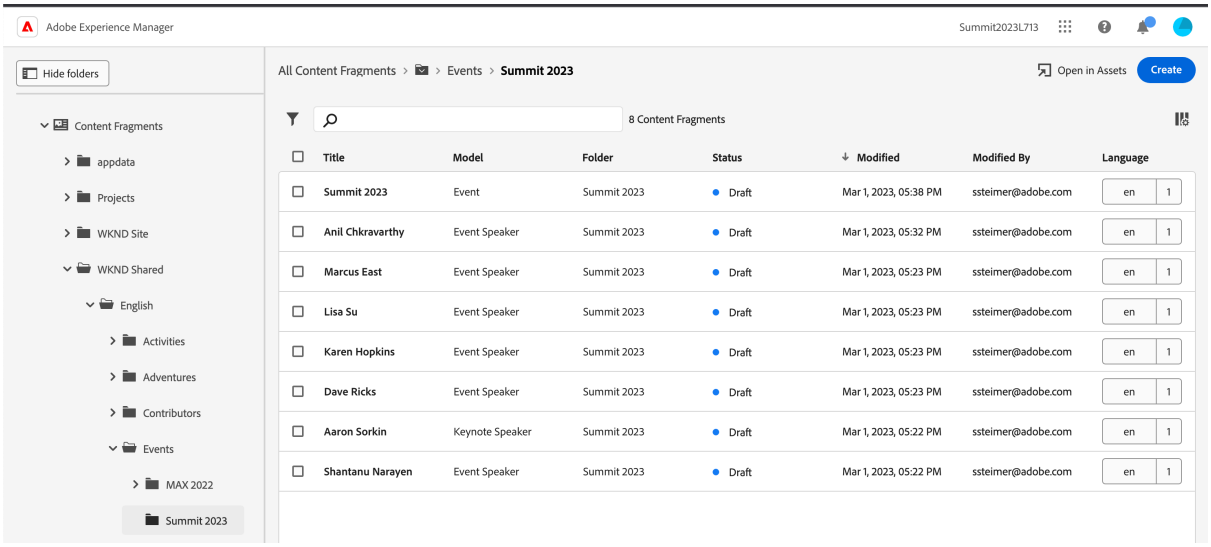
```
{
  "title": "Shantanu Narayen",
  "model": "/conf/wknd-shared/settings/dam/cfm/models/event-speaker",
  "parentPath": "/content/dam/wknd-shared/en/events/summit-2023",
  "data": {
    "name": "Shantanu Narayen",
    "title": "Chairman and CEO, Adobe",
    "profilePicture": "/content/dam/wknd-shared/en/events/summit-2023/shantanu-
narayen.png"
  }
}
```

```
{
  "title": "Aaron Sorkin",
  "model": "/conf/wknd-shared/settings/dam/cfm/models/keynote-speaker",
  "parentPath": "/content/dam/wknd-shared/en/events/summit-2023",
  "data": {
    "name": "Aaron Sorkin",
    "title": "Academy Award-Winning Writer, Director, and Playwright",
    "heroImage": "/content/dam/wknd-shared/en/events/summit-2023/aaron-sorkin-
hero.png",
```

```
    "profilePicture": "/content/dam/wknd-shared/en/events/summit-2023/aaron-
sorkin.jpg",
    "biography": "<p>Academy-Award winning writer, director and renowned playwright
Aaron Sorkin graduated from Syracuse University with a B.F.A. in
Theatre.</p>\n<p>He made his Broadway playwriting debut at the age of 28 with the
military courtroom drama A Few Good Men, for which he received the John Gassner
Award as Outstanding New American Playwright. The following year saw the debut of
his off-Broadway play Making Movies, and in 2007, he returned to Broadway with The
Farnsworth Invention, directed by Des McAnuff.</p>\n<p>Sorkin made the jump to
feature films with his 1993 adaptation of his own play A Few Good Men. The film was
nominated for four Academy Awards including Best Picture. He followed this success
with the screenplays for Malice, starring Alec Baldwin and Nicole Kidman, The
American President, starring Michael Douglas and Annette Bening, and Charlie
Wilson's War, starring Tom Hanks, Philip Seymour Hoffman, and Julia
Roberts.</p>\n<p>In 2011, Sorkin won the Academy Award, Critics' Choice Award,
British Academy of Film and Television Arts Award, and Writers Guild Award in the
Best Adapted Screenplay category as well as the USC Scripter Award for The Social
Network. </p>\n<p>The following year, Sorkin adapted, alongside Steve Zaillian
with story by Stan Chervin, Moneyball for the big screen. The film won Sorkin the
Critics' Choice Award and New York Film Critics' Award for Best Adapted Screenplay,
and went on to receive four Academy Award nominations including Best Picture and
Best Adapted Screenplay. In 2015, Sorkin wrote the feature film Steve Jobs based on
the Walter Isaacson biography of the late Apple co-founder. His adaptation garnered
him nominations for a Broadcast Film Critics' Association (BFCA) Critics' Choice
Award, Writers Guild Award, and multiple regional critics' association
awards.</p>\n<p>Sorkin made his directorial debut in 2017 with Molly's Game, which
he also wrote based on the personal memoir by Molly Bloom. It made its world
premiere at the 2017 Toronto International Film Festival to rave reviews and
garnered Sorkin Best Screenplay nominations for an Academy Award, Writers Guild
Award, and BAFTA Award. In 2020, Sorkin premiered his feature drama The Trial of
the Chicago 7, which he wrote and directed for Netflix. The picture features an
all-star ensemble cast and garnered Sorkin six Critics' Choice Awards and three SAG
awards.</p>\n<p>For television, Sorkin created and produced NBC's renowned series
"The West Wing," which earned nine Emmy nominations in its first season. The series
went on to win a total of 26 Primetime Emmy Awards, including Outstanding Drama
Series for four consecutive years from 2000-2003. For his work on the series,
Sorkin twice received the Peabody Award and Humanitas Prize, as well as three
Television Critics Association Awards and Producers Guild Awards, and a Writers
Guild Award. He also produced and wrote the television series "Sports Night" for
ABC, which garnered eight Emmy nominations and won the Humanitas Prize and the
Television Critics Association Award. Additionally, Sorkin created the series
"Studio 60 on the Sunset Strip," which took place behind-the-scenes of a live
sketch-comedy show and received five Emmy nominations in 2007.</p>\n<p>In 2012,
Sorkin made his return to television with the HBO drama "The Newsroom," bringing in
an average of 7 million viewers per episode. The show won a Critics Choice
Television Award for Most Exciting New Series and has been nominated for numerous
awards, including five Primetime Emmy Awards, a Writers Guild Award, and Directors
Guild Award. The third and final season aired on HBO in 2014, closing the series on
a ratings season high.</p>\n<p>In 2018, Sorkin premiered his Broadway stage
adaptation of Harper Lee's iconic American novel To Kill a Mockingbird. The
production  currently holds the title of the highest grossing American play in
Broadway history. </p>\n"
  }
}
```

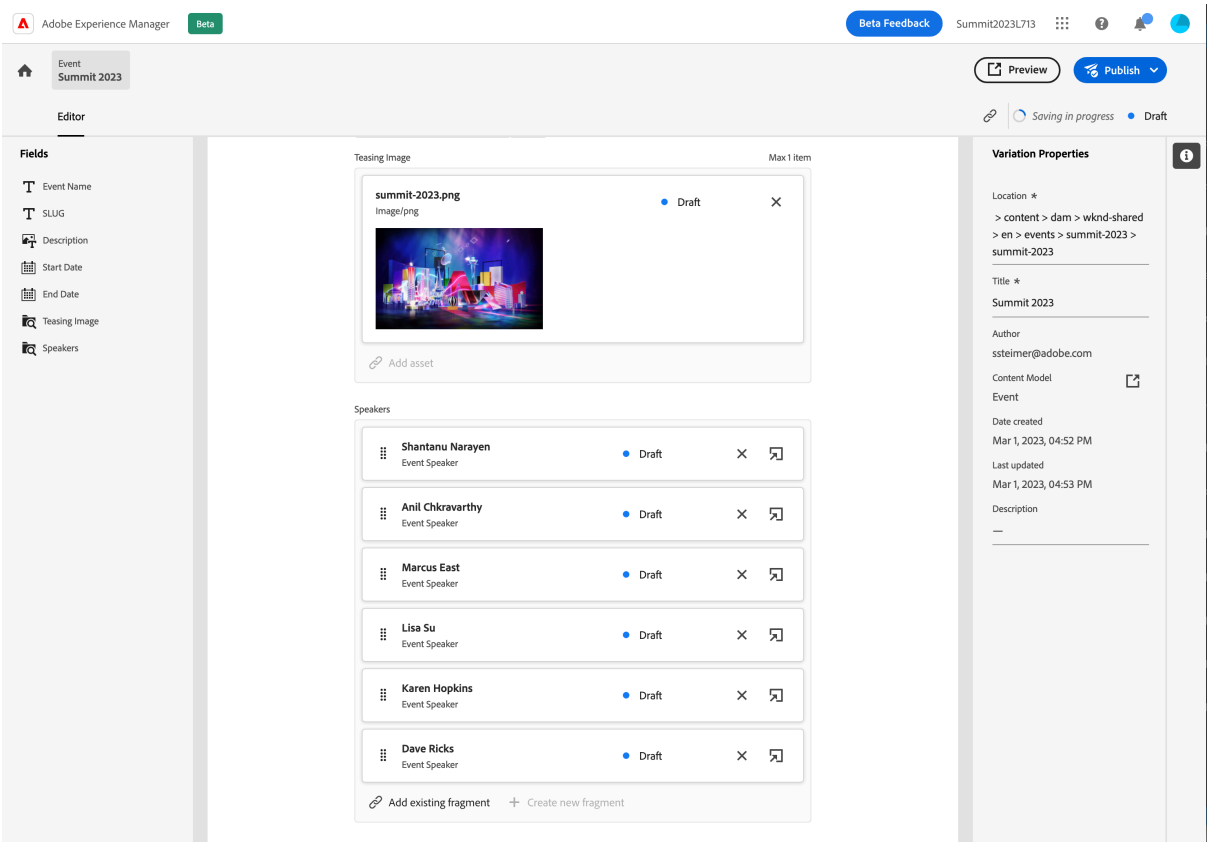

Once you are familiar with the api request structure, go ahead and execute all the requests for creating speakers, 7 in total.

Having created all these speakers, we can validate they were created successfully in the UI



Now we can update the Summit 2023 Event to reference the speakers we just created.

Open the Summit 2023 event in the Content Fragment Editor. Use the “Add Existing Fragment” button to add references to each of the 6 speakers, until it looks like this:



Exercise 2 – Deliver content via GraphQL

Objective

1. Learn how to query content using GraphQL API
2. Explore query capabilities
3. Understand performance best practices
4. Build persisted queries

Lesson context

In this lesson, we will learn how to retrieve AEM content stored in Content Fragments via GraphQL API and GraphiQL user interface, then explore various query capabilities via some examples, and finally understand best practices for optimized delivery performance, leveraging persisted queries.

Exercise 2.1 – Create GraphQL endpoint

A GraphQL endpoint must be configured to enable GraphQL API queries for Content Fragments.
The GraphQL endpoints are created based on a project configuration and will only enable queries against Content Fragment Models belonging to that project.

Given we are working with the WKND sample app, those steps will not be needed.

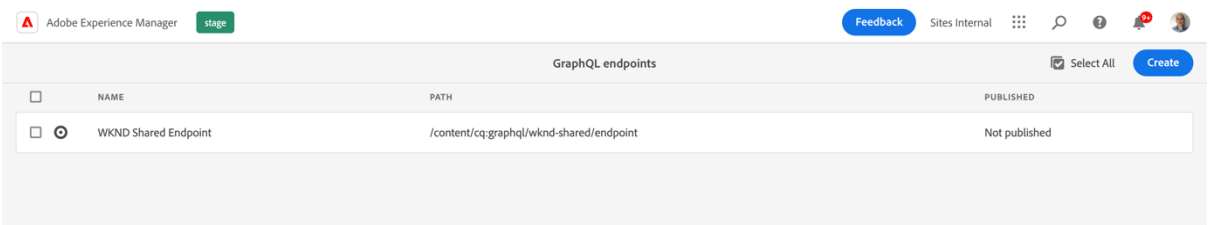
1. Click on Adobe Experience Manager -> Tools -> General -> GraphQL

2. Click on **Create** in the top-right corner, and specify the following values in the dialog:

Field	Value
Name	WKND
Use GraphQL schema provided by ...	wknd-shared

3. Click on **Create** to save the endpoint.

The list is now showing a GraphQL endpoint on your environment.



You would need to publish that endpoint to be able to run queries on the publish tier.

4. Select the GraphQL endpoint you just created.

5. Click on **Publish**

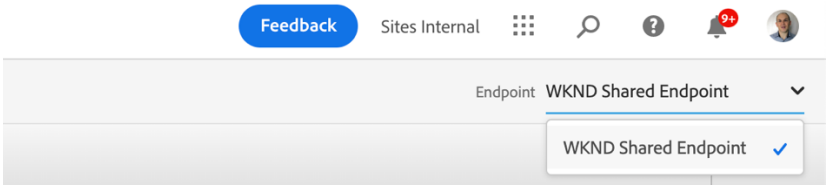
Observe the endpoint is now marked as Published.

Exercise 2.2 – Explore GraphiQL

GraphiQL is a tool included with AEM that enables developers to create and test queries against the content on the current environment.
It also allows users to save queries to be used by client applications in a production setup.

1. Click on Adobe Experience Manager -> Tools -> General -> GraphQL Query Editor

2. The top right corner allows for selecting which endpoint to run the queries against. Make sure the **WKND Shared** endpoint you created previously is selected.



Exercise 2.2.1 – Understand basic syntax

1. On the top right, click on < **Docs** to open the documentation.

2. Click on **QueryType** and observe the list that contains multiple queries.

3. Search for *event*.

Each model is automatically generating a GraphQL schema definition with 3 types of queries:
- `...ByPath` - to retrieve a single Content Fragment by path
- `...List` - to retrieve a list of Content Fragments (offset based)
- `...Paginated` - to retrieve a list of Content Fragments (cursor based)

The name of the query is derived from the Content Fragment Model title.

4. Click on *eventByPath*.

Observe the type of results and the arguments; for this query, providing a *_path* is mandatory.

5. Click on *EventModelResult*, then on *item*.

You can see the list of the different properties defining a model.

Exercise 2.2.2 – Build first query

1. In the left part of the screen, enter the text field under the Play button.

2. Enter your first query:

```
{
  eventByPath(
    _path: "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
  ) {
    item {
      _path
      eventName
      slug
      startDate
      endDate
    }
  }
}
```

3. Run the query by clicking on the ▶ button.
   You should see a result like:

```
{
  "data": {
    "eventByPath": {
      "item": {
        "_path": "/content/dam/wknd-shared/en/events/summit-2023/summit-2023",
        "eventName": "Adobe Summit 2023",
        "slug": "summit-2023",
        "startDate": "2023-03-21",
        "endDate": "2023-03-23"
      }
    }
  }
}
```

4. Add the *description* property to the query.

Observe that you can specify one or more formats in which you want to get the content.

Example:

```
{
  eventByPath(
    _path: "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
  ) {
    item {
      # Add to previous
      description {
        html
      }
    }
  }
}
```

Returns

```
{
  "data": {
    "eventByPath": {
      "item": {
        (...)
        "description": {
          "html": "<h2>The ultimate experience is back.</h2>\n<p>Join us in Las
Vegas to expand your skillset, spark inspiration, and build connections that
empower you to make the digital economy personal. For those unable to attend the
world-class event in Vegas, join us virtually from anywhere for free.</p>\n"
        }
      }
    }
  }
}
```

Exercise 2.3 – Build Sample Queries

Exercise 2.3.1 – Querying fragment references

As we saw while modelling the data structure, a Content Fragment can have references.

1. Press *command* key and click on *item* in the query name (*eventByPath*).
   Observe this is returning a list of *EventSpeakerModel*

2. Click on *EventModelResult!* in the right panel, you see that it contains an *item* field of type *EventSpeakerModel*.

3. Click on *EventSpeakerModel* to visualize all the possible fields.

4. Observe one of the field is *speakers*, which is an array of *EventSpeakerModel*.
   Clicking on the *EventSpeakerModel*, you see it contains the fields of the Speaker model.

5. Modify your query to return the name and *title* of the *speakers*:

```
{
  eventByPath(
    _path: "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
  ) {
    item {
      # Add to previous
      speakers {
        name
        title
      }
    }
  }
}
```

6. Run the query and observe the speaker information is now displayed, as an array containing the entries.

```
{
  "data": {
    "eventByPath": {
      "item": {
        (...)
        "speakers": [
          {
            "name": "Shantunu Narayan",
            "title": "Chairman and CEO, Adobe"
          },
          {
            "name": "Anil Chakravarthy",
            "title": "President, Digital Experience Business, Adobe"
          },
          ...
        ]
      }
    }
  }
}
```

Exercise 2.3.2 – Using GraphQL Union

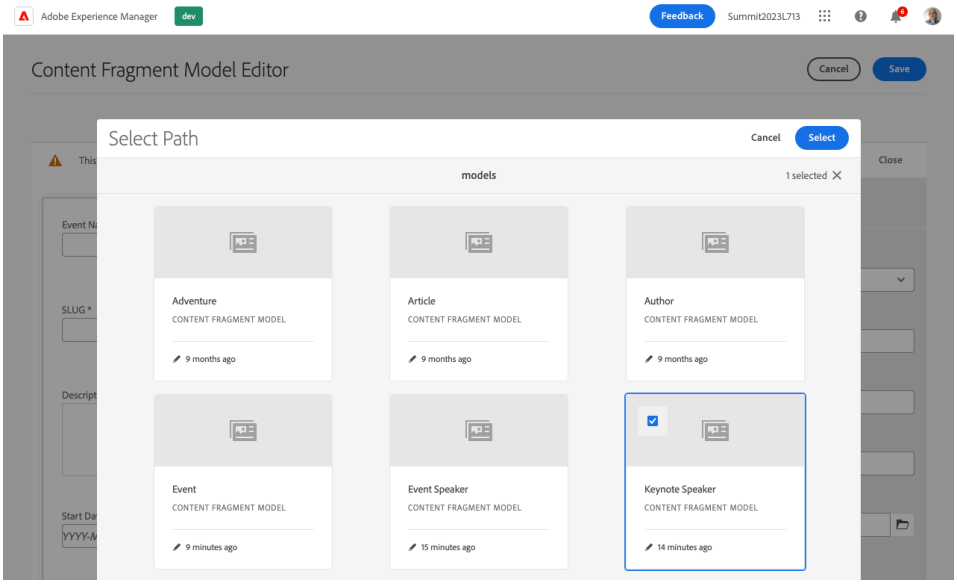
We want to allow 2 types of speakers referenced in an event:

- Regular speakers
- Keynote speakers

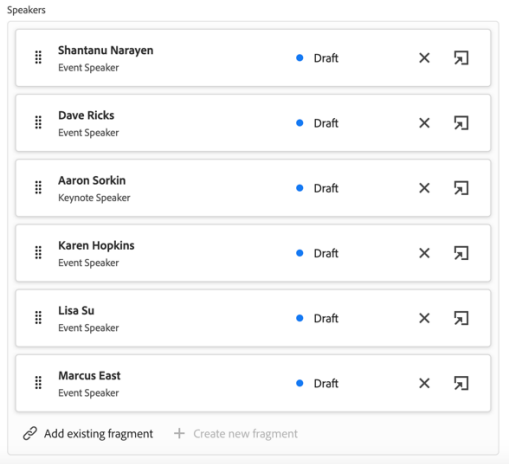
1. Go to Models console and open the **Event** model.

2. If getting a warning when editing the model, acknowledge the warning by clicking on **Edit**.

3. Select the *Speakers* field and open its *Properties*. Under *Allowed Content Fragment Models*, click on the icon to select *Keynote Speaker* on top of *Event Speaker*.
Click on **Save** to persist model changes.



4. Go to *Content Fragments* admin console, add *Aaron Sorkin* in the speakers and drag that entry as 3rd entry of the list.



5. Using GraphQL union allows to return different results based on different models. Adjust the query by updating the *speakers* section of the query:

```
{
  eventByPath(
    _path: "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
  ) {
    item {
      # Replaces previous "speakers" section
      speakers {
        # For speakers we still return the same fields
        ...on EventSpeakerModel {
          name
          title
        }
        # For keynote speakers we return slightly different information
        ...on KeynoteSpeakerModel {
          name
          biography {
            html
          }
        }
      }
    }
  }
}
```

6. Check how the results are returned as part of the same array:

```
{
  "data": {
    "eventByPath": {
      "item": {
        (...)
        "speakers": [
          {
            "name": "Shantunu Narayan",
            "title": "Chairman and CEO, Adobe"
          },
          {
            "name": "Anil Chakravarthy",
            "title": "President, Digital Experience Business, Adobe"
          },
          {
            "name": "Aaron Sorkin",
            "biography": {
              "html": ...
            }
          }
        ]
      }
    }
  }
}
```

Exercise 2.3.3 – Query content references

Query Image References

1. You can request relevant information for content references, for instance images:

```
{
  eventByPath(
    _path: "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
  ) {
    item {
      # Add to previous
      teasingImage {
        ... on ImageRef {
          _path
          _publishUrl
          mimeType
          width
          height
        }
      }
    }
  }
}
```

2. Check results:


```
{
  "data": {
    "eventByPath": {
      "item": {
        (...)
        "teasingImage": {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/Summit.png",
          "_publishUrl": "https://publish-p50166-
e978689.adobeaemcloud.com/content/dam/wknd-shared/en/events/summit-
2023/Summit.png",
          "mimeType": "image/png",
          "width": 3104,
          "height": 1866
        }
      }
    }
  }
}
```


Depending on the content type, you can retrieve other types of objects.

Requesting references

1. You can run a query that gets all the references in a single array, which can be useful for pre-fetching images:

```
{
  eventByPath(
    _path: "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
  ) {
    # Add to previous
    _references {
      ... on ImageRef {
        _path
        _publishUrl
        mimeType
        width
        height
      }
      ... on MultimediaRef {
        _publishUrl
        format
      }
      ... on DocumentRef {
        _publishUrl
        type
        author
      }
    }
    item {
      # Adjust the information requested from speakers
      speakers {
        name
        title
        # Add profile picture information (only the path)
        profilePicture {
          ... on ImageRef {
            _path
          }
        }
      }
```

```
      # Replace previous "teaserImage" block (only the path)
      teasingImage {
        ... on ImageRef {
          _path
        }
      }
    }
  }
}
```

2. Observe how references are returned in a single object, regardless where in content hierarchy they are included.


```
{
  "data": {
    "eventByPath": {
      "_references": [
        {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/anil-
chakravarthy.png",
          "_publishUrl": "https://publish-p50166-
e972896.adobeaemcloud.com/content/dam/wknd-shared/en/events/summit-2023/anil-
chakravarthy.png",
          "mimeType": "image/png",
          "width": 540,
          "height": 540
        },
        {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/Summit.png",
          "_publishUrl": "https://publish-p50166-
e972896.adobeaemcloud.com/content/dam/wknd-shared/en/events/summit-
2023/Summit.png",
          "mimeType": "image/png",
          "width": 3104,
          "height": 1866
        },
        {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/Shantanu-
450x650.jpg.img.jpg",
          "_publishUrl": "https://publish-p50166-
e972896.adobeaemcloud.com/content/dam/wknd-shared/en/events/summit-2023/Shantanu-
450x650.jpg.img.jpg",
          "mimeType": "image/jpeg",
          "width": 450,
          "height": 650
        }
      ],
      "item": {
        (...)
      }
    }
  }
}
```

Dynamic Images

You can use AEM Dynamic Media to:

- **Pass Dynamic Imaging commands into GraphQL queries**
  This means that the commands get applied during query execution, in the same way as URL parameters on GET requests for those images.
- **Dynamically create image renditions in JSON delivery**
  This avoids having to manually create and store those renditions in the repository.

The solution in GraphQL means you can:

- use *_dynamicUrl* on the *ImageRef* reference
- add *_assetTransform* as an argument to your query

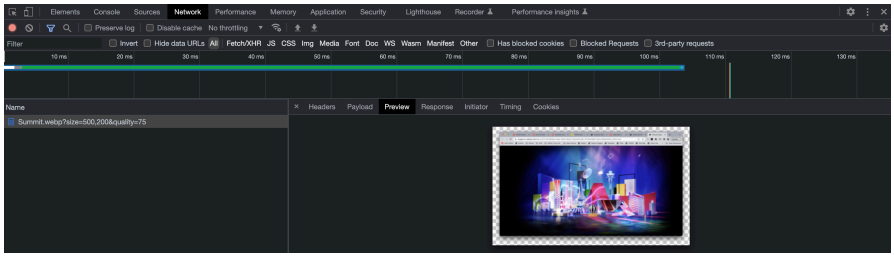
1. Adjust the query to retrieved transformed assets:

```
{
  eventByPath(
    _path: "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
    # Add asset transformation instructions
    _assetTransform: {
      format: WEBP
      size: {
        height: 200
        width: 500
      }
      quality: 75
    }
  ) {
    # Remove "_references" information
    item {
      # Replace teasing image with a dynamic URL
      teasingImage {
        ... on ImageRef {
          _path
          _dynamicUrl
          mimeType
        }
      }
    }
  }
}
```

2. This is providing a dynamic URL:

```
{
  "data": {
    "eventByPath": {
      "item": {
        (...)
        "teasingImage": {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/Summit.png",
          "_dynamicUrl": "/adobe/dynamicmedia/deliver/dm-aid--1730341e-08eb-4d81-
88ee-138e185249e6/Summit.webp?size=500,200&quality=75",
          "mimeType": "image/png"
        }
      }
    }
  }
}
```

- You can copy that URL and append it to your AEM environment's host, for instance: https://author-p50166-e972896.adobeaemcloud.com/adobe/dynamicmedia/deliver/dm-aid--1730341e-08eb-4d81-88ee-138e185249e6/Summit.webp?size=500,200&quality=75



Exercise 2.3.4 – Using parameters

1. While we have been hardcoding the path of the event in our queries so far, we want to make sure the same query can be reused with any event.
   GraphQL supports variables for that purpose:

```
# Name the query and specify parameter(s)
query eventByPath($event: String!) {
  eventByPath(
    # Replace hardcoded path with the variable
    _path: $event
  ) {
    item {
      _path
      eventName
      slug
    }
  }
}
```

2. In the variables section of GraphiQL, specify the variable value:

```
{
  "event": "/content/dam/wknd-shared/en/events/summit-2023/summit-2023"
}
```

3. Run the query, observe you get the expected results:

```
{
  "data": {
    "eventByPath": {
      "item": {
        "_path": "/content/dam/wknd-shared/en/events/summit-2023/summit-2023",
        "eventName": "Adobe Summit 2023",
        "slug": "summit-2023"
      }
    }
  }
}
```

4. Change the variable value to the path of another event:

```
{
  "event": "/content/dam/wknd-shared/en/events/max-2022/max-2022"
}
```

5. Re-run the query, observe you get results for the provided event:

```
{
  "data": {
    "eventByPath": {
      "item": {
        "_path": "/content/dam/wknd-shared/en/events/max-2022/max-2022",
        "eventName": "Adobe MAX 2022",
        "slug": "max-2022"
      }
    }
  }
}
```

Note:

- Your query could use multiple variables.
- You need to specify their type (*String*, *ID*, *Boolean*, *Date*, etc.)
- You can specify which ones are mandatory – this is achieved by suffixing the variable with ! (example: *String!*)

Exercise 2.3.5 – Retrieve a list of fragments

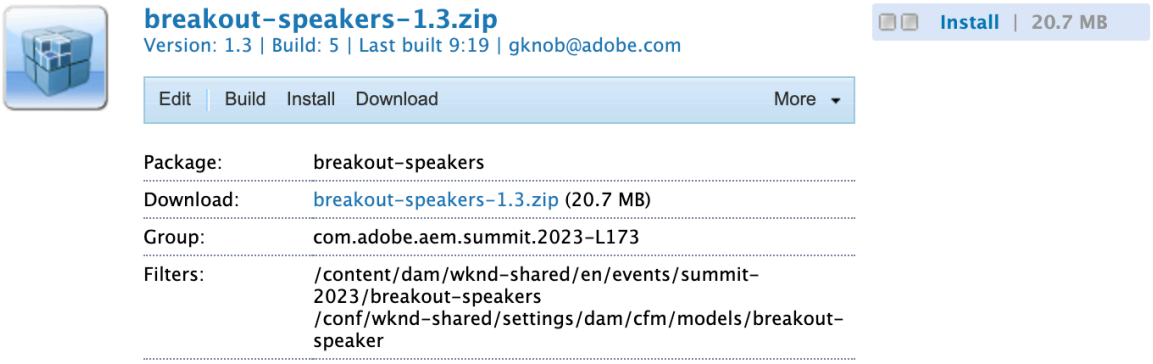
Preparation

We are now first going to import a package that contains some breakout speakers. This will install ~70 speakers.

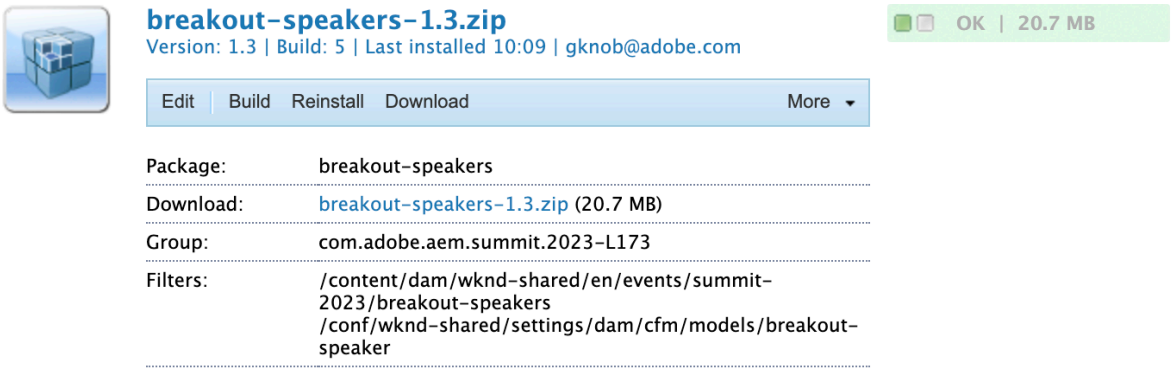
1. Click on Adobe Experience Manager -> Tools -> Deployment -> Packages.

2. Click on **Upload Package**.

3. Navigate to the lab resources folder and select the file **breakout-speakers-1.3.zip**.



4. Click on **Install**, then confirm in the opened modal by clicking **Install**.
   After some time the status should move to *OK*.



5. Confirm by opening *Assets* console and navigate to
   WKND Shared > *English* > *Events* > *Summit 2023* > *Breakout Speakers*
   You should see a list of content fragments and their associated profile picture.

Retrieving list of breakout speakers

We are going to build a query that will list all the breakout speakers.

1. Open GraphiQL and enter the following query.

Note the differences in syntax:

- Using the **...List** query to retrieve a list of items.
- The first child object is a list of **items** (was previously a single **item**).

```
{
  # Retrieves a list of content speakers
  breakoutSpeakerList {
    items {
      _path
      name
      title
      company
    }
  }
}
```

2. Look at the results:

- The result in an array, see the [ ] surrounding the responses.
- Scroll down the list of responses, to understand that we are listing all the results.

```
{
  "data": {
    "breakoutSpeakerList": {
      "items": [
        {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/adam-pazik",
          "name": "Adam Pazik",
          "title": "Director - Managed Services, Adobe",
          "company": "Adobe"
        },
        {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/aditi-dutt-chaudhuri",
          "name": "Aditi Dutt Chaudhuri",
          "title": "Sr. Product Manager, Adobe",
          "company": "Adobe"
        },
        ...
      ]
    }
  }
}
```

Limiting the results

We are now going to leverage *limit* parameter to restrict the number of results to first 10 entries.

1. Adjust the query as following:

```
{
  breakoutSpeakerList (
    # Limit results to first 10 entries
    limit: 10
  ) {
    items {
      _path
      name
      title
      company
    }
  }
}
```

2. Observe the response now only contains 10 entries.

Querying next set of results

The next step is to understand how to query the next set of entries, using the *offset* parameter.

1. Adjust the query as following:

```
{
  breakoutSpeakerList (
    # Limit results to first 10 entries
    limit: 10
    # Provide offset to indicate how many entries should be skipped
    offset: 10
  ) {
    items {
      _path
      name
      title
      company
    }
  }
}
```

2. Observe the results now contain different entries.

```
{
  "data": {
    "breakoutSpeakerList": {
      "items": [
        {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/benjie-wheeler",
          "name": "Benjie Wheeler",
          "title": "Sr. Technical Instructor, Adobe",
          "company": "Adobe"
        },
        {
          "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/bertrand-de-coatpont",
          "name": "Bertrand de Coatpont",
          "title": "Sr. Director, Product Management, Adobe",
          "company": "Adobe"
        },
        ...
      ]
    }
  }
}
```

Exercise 2.3.6 – Retrieve a paginated set of fragments

Another way of requested lists of content fragments is by leveraging **...Paginated** queries.

1. Adjust the query as following:

```
{
  # Use of ...Paginated instead of ...List
  breakoutSpeakerPaginated {
    # Results are contained in "edges" object
    edges {
      # Each result is a "node"
      node {
        # From there on, same properties, this node represents a breakout speaker
        _path
        name
        title
        company
      }
    }
  }
}
```

2. Observe the results:

- *edges* is also an array, see the [ ]
- Within the array, each result is wrapped in a *node* object
- Scroll down the list of responses, to understand that we are not listing all the results, the default entry set is limit to 50

```
{
  "data": {
    "breakoutSpeakerPaginated": {
      "edges": [
        {
          "node": {
            "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/victor-reiss",
            "name": "Victor Reiss",
            "title": "VP Consumer Marketing & Insights, UNC Health",
            "company": "UNC Health"
          }
        },
        {
          "node": {
            "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/hyman-chung",
            "name": "Hyman Chung",
            "title": "Sr. Product Manager, Adobe",
            "company": "Adobe"
          }
        },
        ...
      ]
    }
  }
}
```

Limiting results

1. Adjust the query to *limit* the results to first 10 entries

```
{
  breakoutSpeakerPaginated (
    # Get first 10 entries
    first: 10
  ) {
    edges {
      node {
        _path
        name
        title
        company
      }
    }
  }
}
```

2. Look at the results, and see the list is now returning only 10 results

Navigating through pages

1. Adjust the query to request page information

- *hasNextPage/hasPreviousPage* to understand if there's a next/previous page of results
- *startCursor/endCursor* to get the cursor corresponding to first and last result of the entries

```
{
  breakoutSpeakerPaginated (
    first: 10
  ) {
    edges {
      node {
        _path
        name
        title
        company
      }
    }
    # Request page information
    pageInfo {
      hasNextPage
      hasPreviousPage
      startCursor
      endCursor
    }
  }
}
```

2. Observe the results

- *hasNextPage/hasPreviousPage* to understand if there's a next/previous page of results
- *startCursor/endCursor* to get the cursor corresponding to first and last result of the

```
{
  "data": {
    "breakoutSpeakerPaginated": {
      "edges": [
        ...
      ],
      "pageInfo": {
        "hasNextPage": true,
        "hasPreviousPage": false,
        "startCursor": "MDY5ZTNlNzUtMjQ4MC00YzY5LWI4NGItNTE4N2Q5ZmQxZTNm",
        "endCursor": "MzNhMjMwMzMtNmFjMC00NWY1LTlmMDUtYzMzZjAzMzcwMDEy"
      }
    }
  }
}
```

3. Request the next set of results, by specifying the *after* cursor value, which is the *endCursor* of the previous result set

```
{
  breakoutSpeakerPaginated (
    first: 10
    # Request page information
    after: "MzNhMjMwMzMtNmFjMC00NWY1LTlmMDUtYzMzZjAzMzcwMDEy"
  ) {
    edges {
      node {
        _path
        name
        title
        company
      }
    }
    pageInfo {
      hasNextPage
      hasPreviousPage
      startCursor
      endCursor
    }
  }
}
```

4. Observe the list of results is returning the next set of entries.

This type of queries could easily be used in an application, for instance to do infinite scrolling:

- Request a paginated list of entries
- Once reaching the end of the list, make a new request specifying the *after* cursor
- Repeat until *hasNextPage* is *false*

Note that we for now only support forward cursor navigation.

Exercise 2.3.7 – Sort results

As you can see, the results in the paginated query are not necessarily sorted.

1. Adjust the query to *filter* by *name*

```
{
  breakoutSpeakerPaginated (
    first: 10
    sort: "name"
  ) {
    edges {
      node {
        _path
        name
        title
        company
      }
    }
    pageInfo {
      hasNextPage
      hasPreviousPage
      startCursor
      endCursor
    }
  }
}
```

2. See results are now sorted.

```
{
  "data": {
    "breakoutSpeakerPaginated": {
      "edges": [
        {
          "node": {
            "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/adam-pazik",
            "name": "Adam Pazik",
            "title": "Director - Managed Services, Adobe",
            "company": "Adobe"
          }
        },
        {
          "node": {
            "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/aditi-dutt-chaudhuri",
            "name": "Aditi Dutt Chaudhuri",
            "title": "Sr. Product Manager, Adobe",
            "company": "Adobe"
          }
        },
        ...
      }
    ],
    "pageInfo": {
      ...
    }
  }
}
```

Exercise 2.3.8 – Filter results

Results might have to be filtered to return the appropriate set of entries.

Basic Property filtering

1. Enter the following filter to do a simple formatting on a String property

- Specify which property to filter on
- Provide a list of operations and their definitions

```
{
  breakoutSpeakerPaginated (
    filter: {
      # Filter on the "name" property
      name: {
        _expressions: [
          # List of expressions
          {
            # Value to filter on
            value: "Ben Snyder"
            # Operator to use
            _operator: EQUALS
          }
        ]
      }
    }
  ) {
    edges {
      node {
        _path
        name
        title
        company
      }
    }
  }
}
```

2. See results are filtered to return a single entry

```
{
  "data": {
    "breakoutSpeakerList": {
      "edges": [
        {
          "node": {
            "_path": "/content/dam/wknd-shared/en/events/summit-2023/breakout-
speakers/ben-snyder",
            "name": "Ben Snyder",
            "title": "Practice Lead, Technical Validation, Adobe",
            "company": "Adobe"
          }
        }
      ]
    }
  }
}
```

3. You might want to do more advanced filtering, as an outcome of a search parameter, for instance listing all speakers which name contains "Adam", regardless of the case. Adjust the query

```
{
  breakoutSpeakerPaginated (
    filter: {
      name: {
        _expressions: [
          {
            # Change search criteria
            value: "ben"
            # Change operator to contains
            _operator: CONTAINS
            # Case is not important
            _ignoreCase: true
          }
        ]
      }
    }
  ) {
    edges {
      ...
    }
  }
}
```

- Observe results now return a list of entries: someone else with first name starting with "Ben" but even someone called "Ruben" as we used operator *CONTAINS*.

Combined filters

1. Filters could be combined
   Adjust filter to request anyone with name contains "ben" or equals "Adam Pazik"

```
{
  breakoutSpeakerPaginated (
    filter: {
      name: {
        # Define how to combine filters
        _logOp: OR
        _expressions: [
          {
            value: "ben"
            _operator: CONTAINS
            _ignoreCase: true
          },
          # Second criteria
          {
            value: "Adam Pazik"
            _operator: EQUALS
          }
        ]
      }
    }
  ) {
    edges {
      ...
    }
  }
}
```

2. In the results, observe you get a combined list.

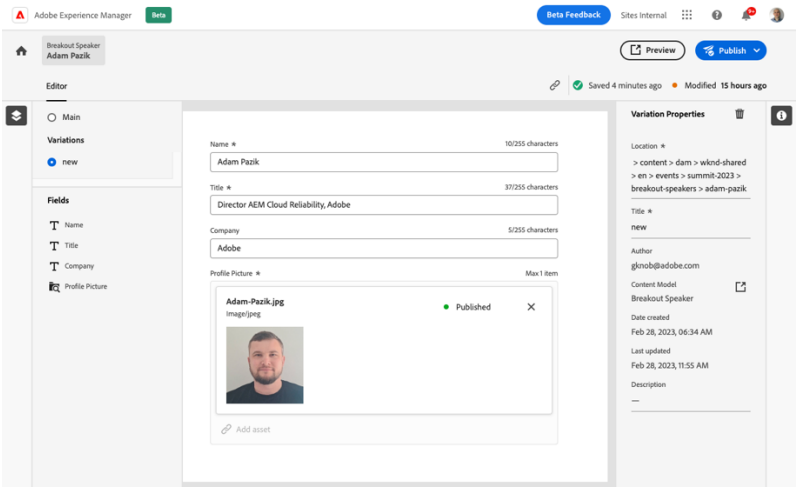
3. You could even decide to filter on multiple properties, for instance all employees working at Adobe (company filter) which title contains "*Director*"
   Adjust the query:

```
{
  breakoutSpeakerPaginated (
    filter: {
      # Combine filter on multiple properties
      _logOp: AND
      company: {
        # Filter on "company"
        _expressions: {
          value: "Adobe"
          _operator: EQUALS
        }
      }
      title: {
        # Filter on "title"
        _expressions: [
          {
            value: "Director"
            _operator: CONTAINS
          }
        ]
      }
    }
  ) {
    edges {
      ...
    }
  }
}
```

4. Look at the results, it only returns breakout speakers matching the 2 conditions.

Querying content variations

There might be multiple **variations** of the same content, which you can filter on.
For instance, Adam Pazik has a specific variation called "new", which is adjusting his business title.



1. Adjust the query to request the content using the *new* variation

```
{
  breakoutSpeakerPaginated (
    filter: {
      _logOp: AND
      company: {
        _expressions: {
          value: "Adobe"
          _operator: EQUALS
        }
      }
      title: {
        _expressions: [
          {
            value: "Director"
            _operator: CONTAINS
          }
        ]
      }
    }
    # Filter to return a specific variation
    variation: "new"
  ) {
    edges {
      ...
    }
  }
}
```

2. Check the results for "Adam Pazik" and observe the title was adjusted.
   However, there's a fallback to master for the other entries which don't have the *new* variation.

Querying localized content

1. Content might be localized, but you might only need one version at the time; results can be filtered by locale.
   Create a new query to list adventures:

```
{
  # Return articles based on a filter on difficulty
  adventureList (
    filter: {
      difficulty: {
        _expressions: {
          value: "Intermediate"
          _operator: EQUALS
        }
      }
    }
  ) {
    items {
      _path
      title
      description {
        plaintext
      }
    }
  }
}
```

2. Observe the results contain entries in multiple languages.

```
{
  "data": {
    "adventureList": {
      "items": [
        {
          "_path": "/content/dam/wknd-shared/en/adventures/climbing-new-
zealand/climbing-new-zealand",
          "title": "Climbing New Zealand",
          "description": {
            "plaintext": ...
          }
        },
        ...
        {
          "_path": "/content/dam/wknd-shared/fr/adventures/climbing-new-
zealand/climbing-new-zealand",
          "title": "Escalade en Nouvelle-Zélande",
          "description": {
            "plaintext": ...
          }
        }
      ]
    }
  }
}
```

3. Filter the results to return only adventures in French by adjusting the query:

```
{
  adventureList (
    filter: {
      difficulty: {
        _expressions: {
          value: "Intermediate"
          _operator: EQUALS
        }
      }
    }
    # Restrict results to "French"
    _locale: "fr"
  ) {
    items {
      ...
    }
  }
}
```

4. Observe results are down to a single entry, with content in French.

Filtering on nested properties

Content fragments can refer other content fragments, and you might want to filter based on such properties.

1. Create a new query in GraphiQL (either by refreshing the page or clicking the + button next to *Persisted Queries* in the left pane:

```
{
  articleList {
    items {
      _path
      title
      slug
      # Author is a nested content fragment
      authorFragment {
        firstName
        lastName
        profilePicture {
          ...on ImageRef {
            _path
            _authorUrl
            _publishUrl
          }
        }
      }
    }
  }
}
```

2. Observe all articles are returned.

3. Filter articles based on author's *lastname*.

```
{
  articleList(
    filter: {
      # Filter on nested content fragment
      authorFragment: {
        # Specify on which property to filter
        lastName: {
          _expressions: {
            value: "Sjöberg",
            _operator: EQUALS
          }
        }
      }
    }
  ) {
    items {
      ...
    }
  }
}
```

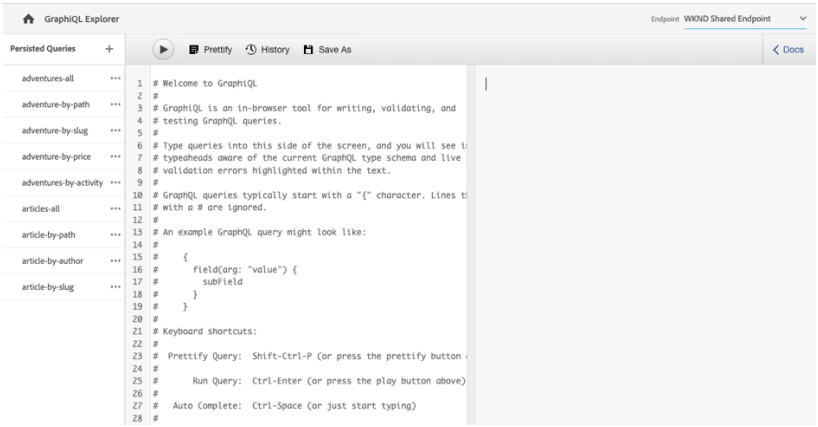
4. Observe only the articles written by *Sofia Sjöberg* are returned.

Exercise 2.4 – Building persisted queries

Persisted queries are stored on AEM. Clients can send an HTTP GET request with the query name to execute it. The benefit of this approach is URL stability and cache-ability.

Creating a persisted query via GraphiQL
We are going to create a new persisted query for listing the events.

1. Open GraphiQL and observe the left panel, that contains existing persisted queries.



2. Click on + icon to create a new query:

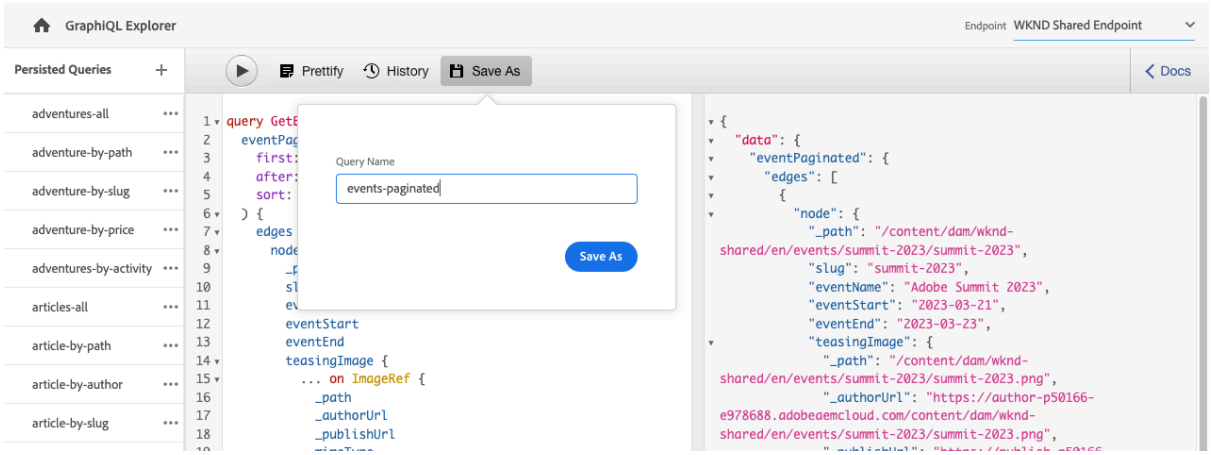
```
query GetEvents($count: Int!, $after: String) {
  eventPaginated(
    first: $count,
    after: $after,
    sort: "startDate desc"
  ) {
    edges {
      node {
        _path
        slug
        eventName
        startDate
        endDate
        teasingImage {
          ... on ImageRef {
            _path
            _authorUrl
            _publishUrl
            mimeType
            width
            height
          }
        }
      }
    }
    pageInfo {
      hasPreviousPage
      hasNextPage
      startCursor
      endCursor
    }
  }
}
```

3. Specify variables:

```
{
  "count": 10
}
```

4. Run the query and validate it is working fine.

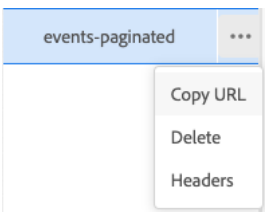
5. Click on **Save As** to persist the query.



6. In the prompt, enter the name **events-paginated**, and click on **Save As**.

Observe a new persisted query has been created

7. Click on **...** next to the new persisted query and select **Copy URL**.
A toast is confirming the URL was copied.



8. Open a new tab and paste the URL.
   Observe the error message, due to missing mandatory variable (*count*).


```
{
  "errors": [
    {
      "message": "Variable 'count' has an invalid value : Expected type 'Int' but
was 'String'.",
      "locations": [
        {
          "line": 1,
          "column": 17
        }
      ],
      "extensions": {
        "classification": "ValidationError"
      }
    }
  ]
}
```


9. Adjust the URL to *…/graphql/execute.json/wknd-shared/events-paginated;**count=10***

10. The query should now return results.


```
// 20230228140304
// https://author-p50166-e978688.adobeaemcloud.com/graphql/execute.json/wknd-shared/events-paginated;count=10

{
  "data": {
    "eventPaginated": {
      "edges": [
        {
          "node": {
            "_path": "/content/dam/wknd-shared/en/events/summit-2023/summit-2023",
            "slug": "summit-2023",
            "eventName": "Adobe Summit 2023",
            "eventStart": "2023-03-21",
            "eventEnd": "2023-03-23",
            "teasingImage": {
              "_path": "/content/dam/wknd-shared/en/events/summit-2023/summit-2023.png",
              "_authorUrl": "https://author-p50166-e978688.adobeaemcloud.com/content/dam/wknd-
shared/en/events/summit-2023/summit-2023.png",
              "_publishUrl": "https://publish-p50166-e978688.adobeaemcloud.com/content/dam/wknd-
shared/en/events/summit-2023/summit-2023.png",
              "mimeType": "image/png",
              "width": 1866,
              "height": 1130
            }
          }
        },
```


11. Go back to GraphiQL and select the newly created persisted query. Click on **Publish**.

12. Replace the author-pXXXXX-eYYYYY by publish-pXXXXX-eYYYYY in the previous tab to test the query on publish.
    Note: you might need to publish the models from models console first.

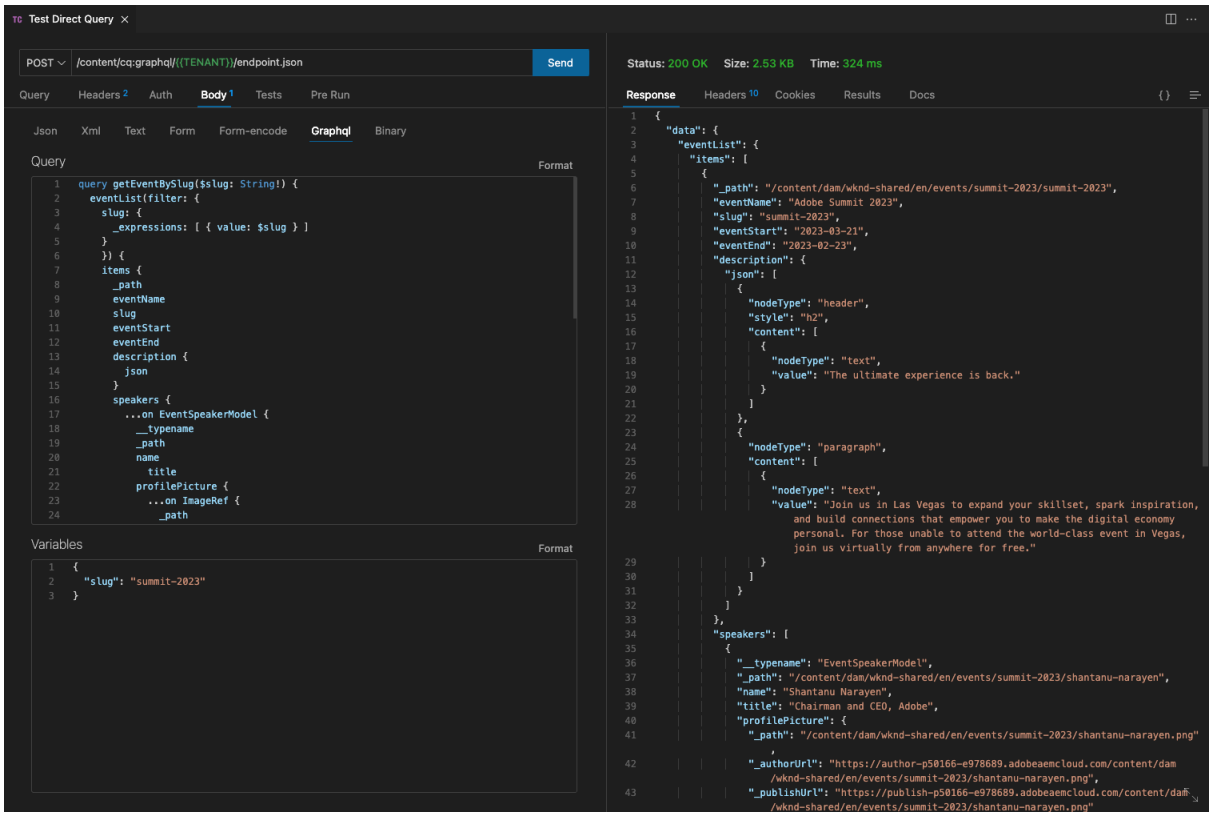
Creating a persisted query via API

1. Open Visual Studio code and go the Thunder Client panel.

2. Open the collection called **Exercise 2**.

3. Select the **Test Direct Query** request and execute it.

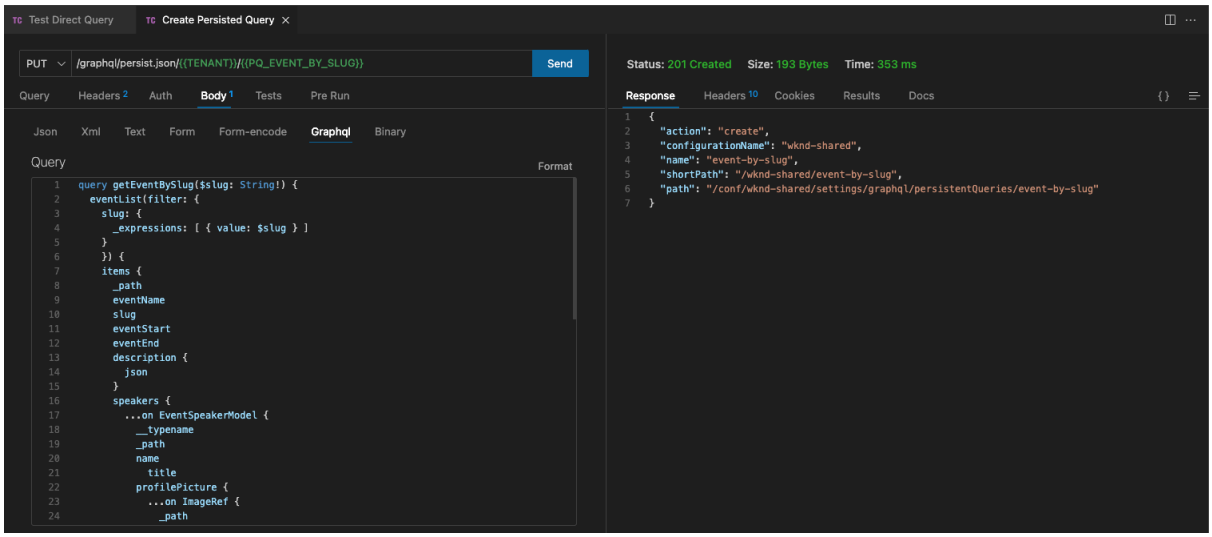
   This request is working exactly like GraphiQL but using a different client.



4. We are now going to persist this query. Select the **Create Persisted Query** request.

- A PUT to /graphql/persist.json will be used to save the query.
- Rest of the URL is composed of the tenant and the name of the query.

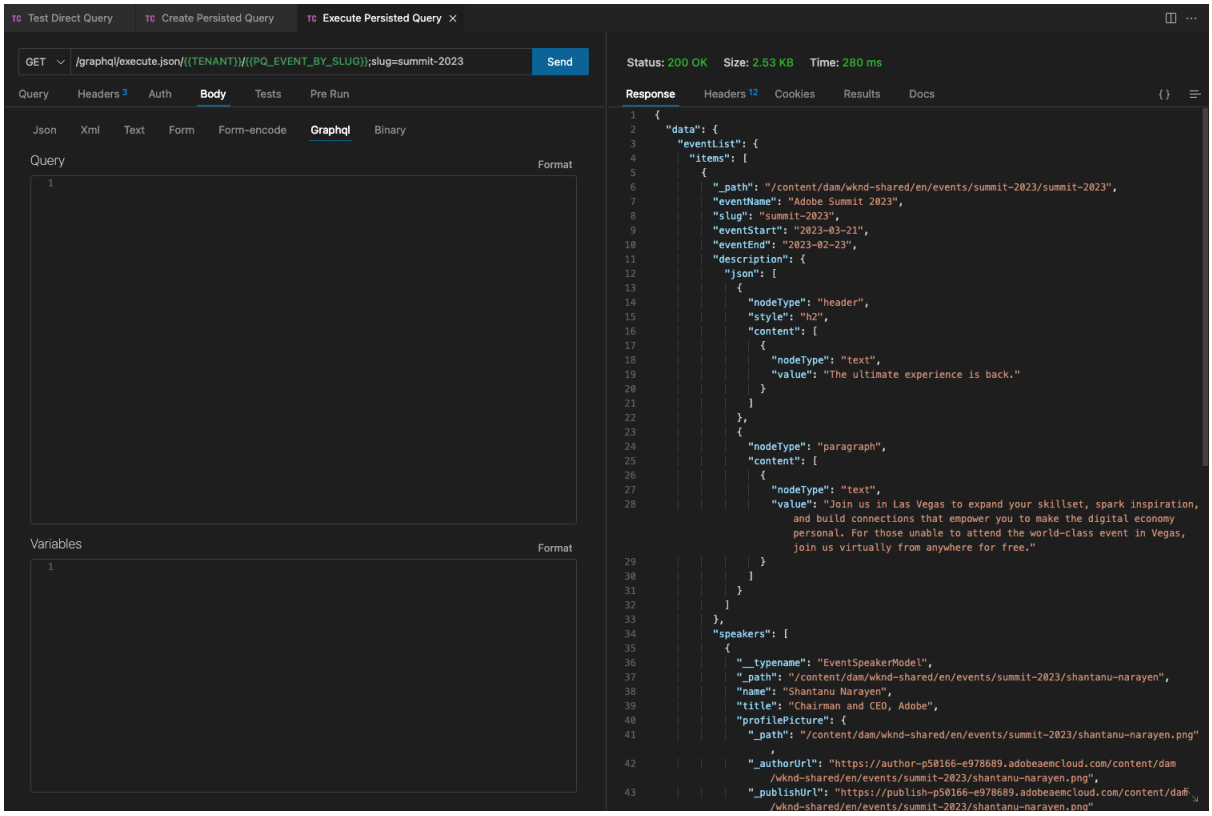
Execute the request.



5. Refresh GraphiQL to see a new query was created.

6. Select the **Execute Persisted Query** request and execute it.

- A GET to /graphql/execute.json is used to execute the query.
- Since the query has variables, you can provide those in the URL, for instance *;slug=summit-2023*
- Note that special characters like semicolons (;), equal sign (=), slashes (/), and space must be converted to use the corresponding UTF-8 encoding.

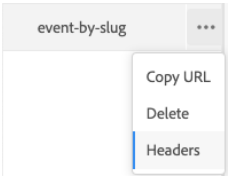


Manage Caching

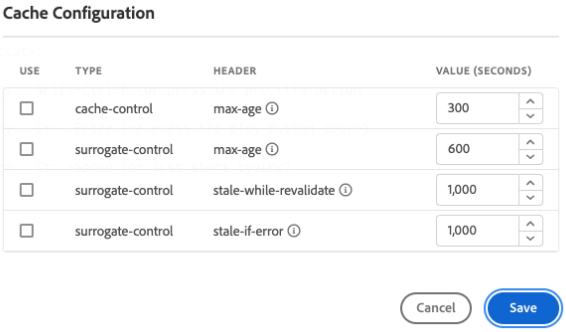
AEM GraphQL Persisted Queries allow you to cache content and define cache-control parameters to your queries to optimize delivery performance.

1. Select the newly created *event-by-slug* query and click on **...**

2. Select the **Headers** option.



3. The opened modal allows to configure cache duration.



- *cache-control / max-age* defines the browser TTL (Time To Live).
- *surrogate-control / max-age* is same but applies specifically to proxy caches, like the CDN.
- *surrogate-control / stale-while-revalidate* defines for how long caches may continue to serve a cached response after it becomes stale.
- *surrogate-control / stale-if-error* defines for how long caches may continue to serve a cached response in case of or origin error.

The setup will apply to Publish and Preview tier only.

Using Persisted Queries in Production is a best practice to provide the best experience to consuming applications.

Tip: On an AEM instance with a high number of Content Fragments that share the same model, GraphQL list queries can become costly

Review https://experienceleague.adobe.com/docs/experience-manager-cloud-service/content/headless/graphql-api/graphql-optimization.html to understand how to optimize them

---

Exercise 3 – Build an application consuming AEM content

Objective

1. Understand the structure of an existing application
2. Learn how to consume the content in an application
3. Understand how to allow the application to be edited in-context

Lesson context

In this lesson, we will look at a React application (could be any other client-side application) that was created, understand how to consume content from AEM, and finally see what is needed to enable any existing application to be edited in-context via Universal Editor.

Exercise 3.1 – Understand the structure of the application

The app we are starting with displays WKND Events and Adventures. For this lab, we will be working with the event portion.

The app itself is relatively straight forward, using GraphQL to fetch content from AEM, then displaying it in the appropriate areas via react components.

The app uses yarn, and can be started by running `yarn start` from a terminal window.

There are several key files you should look at, to familiarize yourself with the app:

App.js – defines the routes used by the application. We will need to update this to add our event detail route.

Events.jsx - the events component, renders the events list on the home page.

AdventureDetail.jsx – Not something we will use directly, but useful as a reference to understand how our event detail component should function.

useGraphQL.js – Defines a custom react hook, used by many components, for executing AEM persisted queries.

Tip: useGraphQL is using the AEM Headless javascript client. You can use this for building your own application, regardless of framework. Find out more at https://github.com/adobe/aem-headless-client-js

---

Exercise 3.2 – Extend the application to consume the content previously created

Create the event detail component.

In the components folder, add 2 new files: EventDetail.jsx and EventDetail.scss

Start by adding some basic structure for EventDetail.jsx

```
/*
Copyright 2023 Adobe
All Rights Reserved.

NOTICE: Adobe permits you to use, modify, and distribute this file in
accordance with the terms of the Adobe license agreement accompanying
it.
*/
import React, { useMemo } from 'react';
import { Link, useNavigate, useParams} from "react-router-dom";
import backIcon from '../images/icon-close.svg';
import Error from './Error';
import Loading from './Loading';
import { mapJsonRichText } from '../utils/renderRichText';
import './EventDetail.scss';
import useGraphQL from '../api/useGraphQL';

function EventDetail() {

    // params hook from React router
    const { slug } = useParams();
    const navigate = useNavigate();
    const eventSlug = slug.substring(1);

    // define the name of the persistentQuery for this component
    const persistentQuery = '';

    // Use a custom React Hook to execute the GraphQL query
    const params = useMemo(() => {
        return {slug: eventSlug };
    }, [eventSlug]);
    const { data, errorMessage } = useGraphQL('', persistentQuery, params);

    // If there is an error with the GraphQL query
    if(errorMessage) return <Error errorMessage={errorMessage} />;

    // If query response is null then return a loading icon...
    if(!data) return <Loading />;

    // add event rendering logic here
}

export default EventDetail;
```

Next, add the name of the query we created in Exercise 2, 'wknd-shared/event-by-slug'. This is the persisted query the event details component will use to retrieve it's data.

```
function EventDetail() {
    // replace this on line 26
    const persistentQuery = 'wknd-shared/event-by-slug';
}
```

Now, let's add some code which parses the responses and ensures the query returned an event as expected:

```
function EventDetail() {
    // add event rendering logic here
    // Set event properties variable based on graphQL response
    const currentEvent = getEvent(data);

    // Set references of current event
    const references = data.eventList._references;

    // Must have title, path, and image
    if( !currentEvent) {
      return <NoEventFound />;
    }
}
```

And the associated getEvent and NoEventFound functions

```
// these should go after the import statement but before the EventDetail() function
function NoEventFound() {
    return (
        <div className="event-detail">
            <Link className="event-detail-close-button" to={"/"}>
                <img className="Backbutton-icon" src={backIcon} alt="Return" />
            </Link>
            <Error errorMessage="Missing data, event could not be rendered." />
        </div>
    );
}

/**
 * Helper function to get the first event from the response
 * @param {*} response
 */
function getEvent(data) {

    if (data && data.eventList && data.eventList.items) {
        // expect there only to be a single event in the array
        if(data.eventList.items.length === 1) {
            return data.eventList.items[0];
        }
    }
    return undefined;
}
```

Finally render the html event markup

```
    // this goes at the end of the EventDetal() function
    return (<div className="event-detail">
      <button className="event-detail-close-button" onClick={() => navigate(-1)} >
          <img className="Backbutton-icon" src={backIcon} alt="Return" />
      </button>
      <EventDetailRender {...currentEvent} references={references}/>
    </div>);
```

And the associated functions used to do that rendering

```
// these should go before the EventDetail() function but after the GetEvent()
function we added before.
function EventDetailRender(props) {
    return (
        <div>
            <h1 className="event-detail-title">{props.eventName}</h1>
            <div className="event-detail-content">
                <img className="event-detail-teasingImage"
                    src={props.teasingImage._publishUrl} alt={props.eventName} />
                <div>
                <div className="event-detail-dates">
                    <span className="event-item-date">{props.startDate}</span>
                    <span> to </span>
                    <span className="event-item-date">{props.endDate}</span>
                </div>
                <div className="event-detail-
description">{mapJsonRichText(props.description.json,
customRenderOptions(props.references))}</div>
                </div>
                <div className="event-detail-speakers">
                    <h2>Featured Speakers</h2>
                    <ul className="event-detail-speakers-grid">
                        {
                            props.speakers.map((speaker) => {
                                return <EventDetailSpeaker key={speaker._path}
{...speaker} />
                            })
                        }
                    </ul>
                </div>
            </div>
        </div>
    );
}

/**
 * Example of using a custom render for in-line references in a multi line field
 */
function customRenderOptions(references) {

    const renderReference = {
        // node contains merged properties of the in-line reference and _references
object
        'ImageRef': (node) => {
            // when __typename === ImageRef
           return <img src={node._path} alt={'in-line reference'} />
        },
        'EventModel': (node) => {
            // when __typename === EventModel
            return <Link to={'/event:${node.slug}'}>{`${node.eventName}:
${node.capacity}`}</Link>;
        }
    };

    return {
        nodeMap: {
            'reference': (node, children) => {

                // variable for reference in _references object
                let reference;

                // asset reference
                if(node.data.path) {
                    // find reference based on path
                    reference = references.find( ref => ref._path ===
node.data.path);
                }
                // Fragment Reference
```

```
                if(node.data.href) {
                    // find in-line reference within _references array based on
href and _path properties
                    reference = references.find( ref => ref._path ===
node.data.href);
                }

                // if reference found return render method of it
                return reference ?
renderReference[reference.__typename]({...reference, ...node}) : null;
            }
        },
    };
}

function EventDetailSpeaker(props) {
    const IsKeynoteSpeaker = (props.__typename === 'KeynoteSpeakerModel');
    const keynnoteSpeakerClass = IsKeynoteSpeaker ? ' event-detail-keynote-speaker'
: '';
    const picture = IsKeynoteSpeaker ? props.heroImage : props.profilePicture;

    return (
        <li className={"event-detail-speaker" + keynnoteSpeakerClass}>
            <div className="event-detail-speaker-card">
                <span className="event-detail-speaker-name">{props.name}</span>
                <span className="event-detail-speaker-title">{props.title}</span>
            </div>
            <img className="event-detail-speaker-img"
                    src={picture._publishUrl} alt={props.name}/>
        </li>
    );
}
```

Most of the code we just added is pretty straightforward, but I think it's worth paying extra attention to 2 things:

- First, the customRenderOptions function, which can render Images or other Events referenced from the event description.

- Second, the EventDetailSpeaker, which deals with the fact that the speaker could be one of 2 different models.

At this point, the full file contents should match https://github.com/Adobe-Marketing-Cloud/Summit-2023-L713/blob/react-app-ex32/react-app/src/components/EventDetail.jsx

Finally, lets add the scss to style everything, also found at https://github.com/Adobe-Marketing-Cloud/Summit-2023-L713/blob/react-app-ex32/react-app/src/components/EventDetail.scss if that is easier for you.

```
/*
Copyright 2022 Adobe
All Rights Reserved.
NOTICE: Adobe permits you to use, modify, and distribute this file in
accordance with the terms of the Adobe license agreement accompanying
it.
*/
@use "sass:math";

@import '../styles/variables';
```

```
.event-detail {
  .event-detail-close-button {
    width: 24px;
    float: right;
    margin: 1em;
    background: none;
    border: none;
  }

  .event-detail-content {
    display: grid;
    gap: 2rem;
    grid-template-columns: 1fr;
    width: 100%;

    .event-detail-teasingImage {
      width: 100%;
      height: auto;
    }

    .event-detail-speakers {
      grid-column: 1 / -1;

       .event-detail-speakers-grid {
        list-style: none;
        display: grid;
        grid-template-columns: repeat(2, 1fr);
        gap: 2rem;
        margin: 2rem;

        .event-detail-speaker {
          position: relative;
          // width: 250px;
          height: 350px;
          display: grid;
          grid-template-rows: 1fr auto;

          &.event-detail-keynote-speaker {
            background-image: url("../images/keynote-bg.png");
          }

          .event-detail-speaker-card {
            background-color: rgba(#000, .7);
            color: white;
            grid-row: 2;
            padding: .5rem 1rem;
            display: flex;
            flex-direction: column;
            align-items: center;
            text-align: center;
            z-index: 1;

            > span {
              margin-bottom: .4rem;
            }

            .event-detail-speaker-name {
              font-weight: 700;
            }
          }

          .event-detail-speaker-img {
            position: absolute;
            object-fit: cover;
            inset: 0;
            height: 100%;
            width: 100%;
```

```
          }
        }
      }
    }
  }
}

@media only screen and (min-width: $mobile-breakpoint) {
  .event-detail {
    .event-detail-content {
      grid-template-columns: 1fr 1fr;

      .event-detail-speakers {
        .event-detail-speakers-grid {
          grid-template-columns: repeat(3, 1fr);
        }
      }
    }
  }
}

@media only screen and (min-width: $tablet-breakpoint) {
  .event-detail {
    .event-detail-content {
      .event-detail-speakers {
        .event-detail-speakers-grid {
          grid-template-columns: repeat(4, 1fr);

          .event-detail-speaker.event-detail-keynote-speaker {
            grid-column: span 2;
          }
        }
      }
    }
  }
}
```

Add the event detail route to App.js.

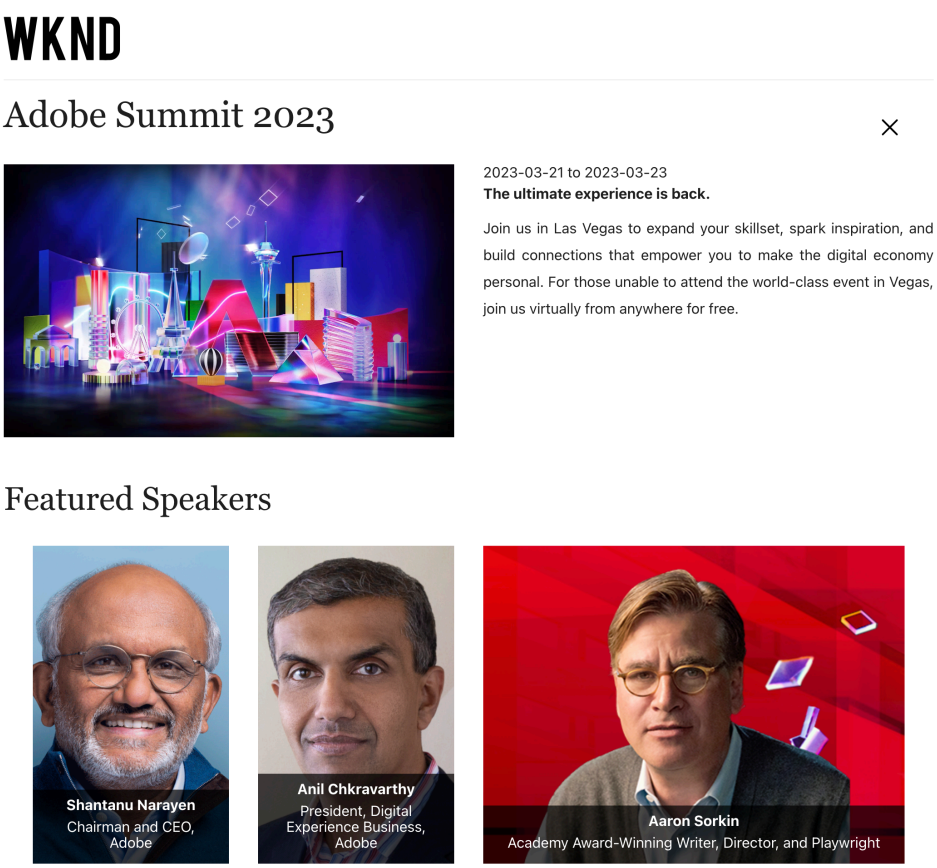
In the <Router> section (starting on line 43) add a new route. This tells the app to use our new component when given an appropriate url.

```
<Route path="/event:slug" element={<EventDetail />} />
```

You'll also need to add an import for the EventDetail component on line 11.

```
import EventDetail from "./components/EventDetail";
```

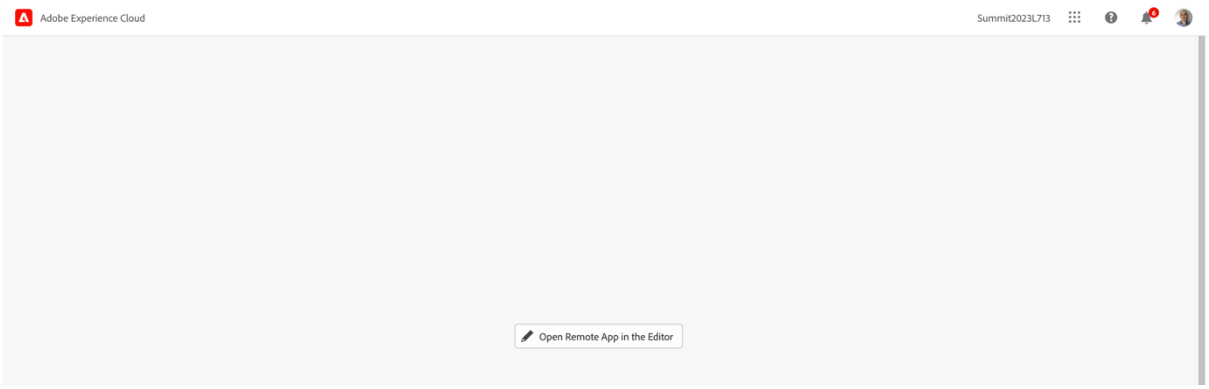
Make sure have saved everything, and reload the app in your browser, and you should be able to see all the changes we made.



Exercise 3.3 – Allow the application to be edited in-context

Understanding Universal Editor concepts

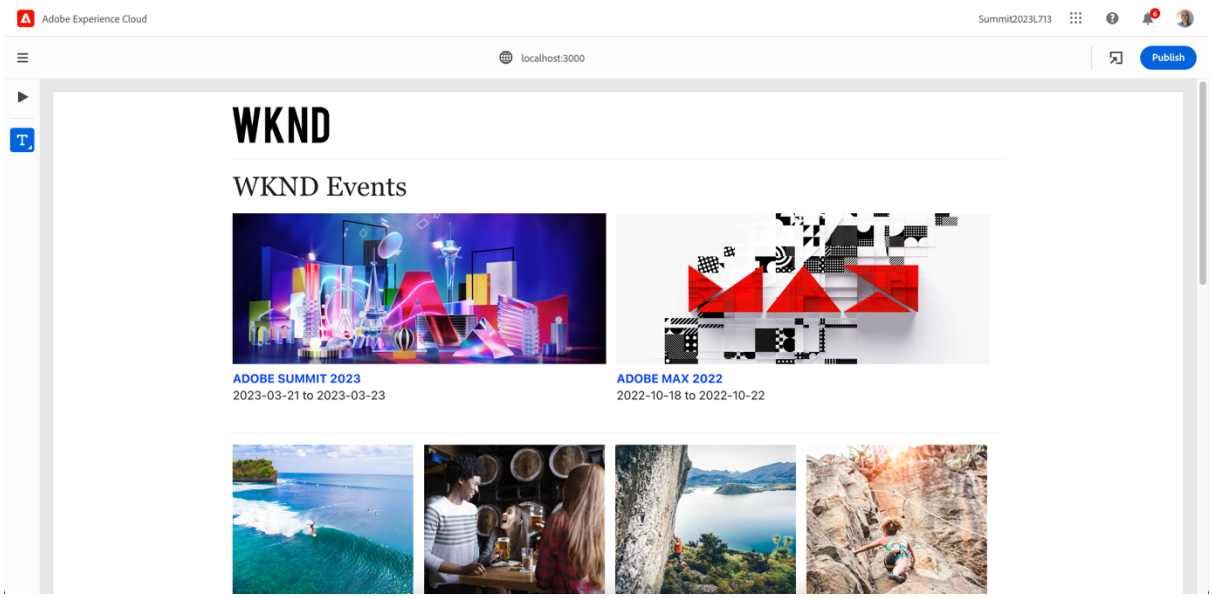
1. Open *Universal Editor* bookmark, this is loading the frame.



2. Click on **Open Remote App in the Editor**, it will load a default remote application.

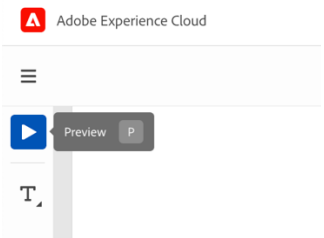
3. Click on the *Globe* icon to open the location bar, and change the location to *https://localhost:3000*

4. The frame is now loading our Summit lab application:



5. Scroll down and hover the *Bali Surf Camp* adventure, it will detect it as a text that can be edited.
   A click on the text is going to open an editor to adjust the property.
   A click on the image does not do anything.

6. In the left panel, select the preview option, change mode to *Preview*

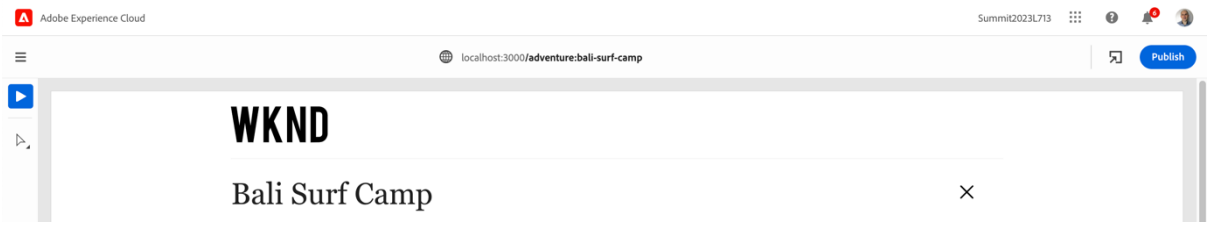


---

*Tip: You can also use keyboard shortcuts to navigate between modes: **T** for Text, **P** for Preview, etc.*

---

7. Click on the image belonging to *Bali Surf Camp*, the location is changing to the adventure detail page.

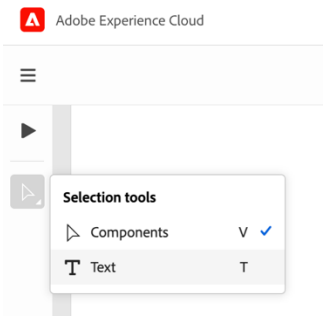


8. Click on the **X** to go back to home page.

Connecting the application to Universal Editor

We saw a text editor was shown when clicking on the *Bali Surf Camp* field.

1. Switch back to *Text* mode by doing a long press on the second icon or via shortcut: *T*



2. Click again in the *Bali Surf Camp* text, and start editing, then press *Enter* key.

3. Open the *Sample App* bookmark (eventually make a hard refresh of the page if already opened).

The app does not display the updated content. This is because it needs to be instrumented to understand where content is coming from.

4. Switch back to *Visual Studio Code* and open file react-app -> public -> index.html

5. Add a new line after theme-color meta information (line 7) with the following entry. Make sure you replace the program and environment IDs with the ones of your sandbox.

```
<meta name="theme-color" content="#000000" />
<meta name="urn:auecon:aemconnection" content="aem:https://author-p12345-
e67890.adobeaemcloud.com">
```

6. Go back to *Universal Editor* and reload the app.

7. Try again an edit of *Bali Surf Camp* text, then press *Enter* key.

8. Switch to *Sample App* tab and refresh the page.
The text has been updated to the one you edited using Universal Editor.

Instrumenting the application

1. In *Universal Editor* tab, hover the *Adobe Summit 2023* even text.
Despite being in *Text* mode, this is not showing any option to edit the text.

The application needs to be instrumented to provide editing capabilities in Universal Editor.

2. Open *Events.*jsx in *Visual Studio Code*. First, we need to define where the content is coming from, by defining the URN connection:

- *props* represents the object representing the properties of the event
- *props._path* is the path of the content fragment in the repository
- */jcr:content/data/master* is the relative path to that content fragment in the repository for the variation holding the content
In this example, we are just hardcoding to *master* but that information could be part of the query results (using the *_variation* field)

```
function EventItem(props) {
  const editorProps = useMemo(() => true && { itemID: "urn:aemconnection:" +
props?._path + "/jcr:content/data/master"}, [props._path]);

  // Rest of the function
  // ...
}
```

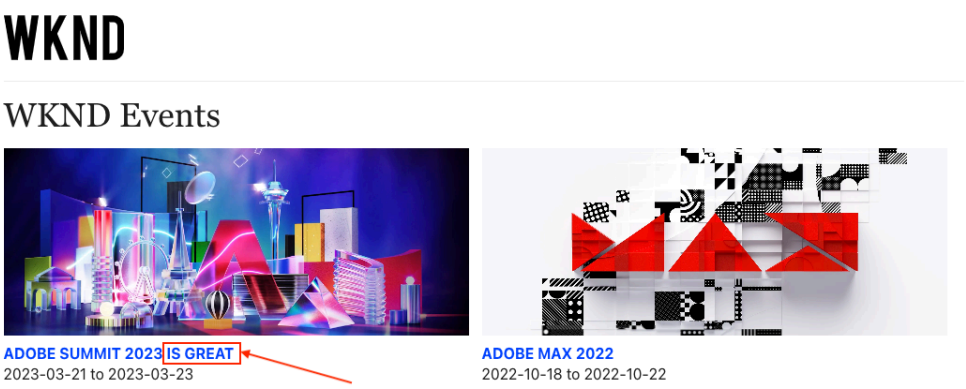
3. Adjust the render function:

- Update the list item element by adding *itemScope* and *editorProps* attributes to list item to connect the specific HTML element with the right URN data
- Add attributes to *Title* div to specify which property it uses
    - *itemProp* defines what is the property to update
    - *itemType* defines what is the type of the property, in this case a text

```
  return (
    <li className="event-item" itemScope {...editorProps}>
      <Link to={`/event:${props.slug}`}>
        <img className="event-item-image" src={`${props.teasingImage._publishUrl}`}
                alt={props.title} itemProp="teasingImage" />
        <div className="event-item-title" itemProp="eventName"
itemType="text">{props.eventName}</div>
      </Link>
      <div className="event-item-details">
        <span className="event-item-date">{props.startDate}</span>
        <span> to </span>
        <span className="event-item-date">{props.endDate}</span>
      </div>
    </li>
  );
```

4. Go back to *Universal Editor*, and click on *Adobe Summit 2023* text, then make an adjustment and press *Enter* key.

5. Open the *Sample App* and see the change has been persisted.



Connecting to Content Fragment Editor

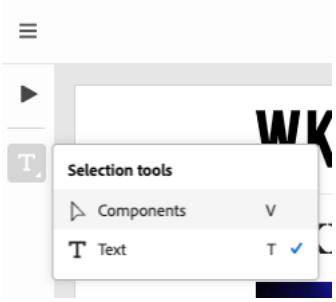
Only a subset of the properties might be editable in context. However, other fields might also influence the content of the application. To make it easy to identify which content fragment to edit, we can connect an action to open the content fragment editor for that particular field.

1. Switch back to *Visual Studio Code*, in the *Events.jsx* file, and update the *editorProps* object to specify the item type, and mention it is a content fragment (*cf*)

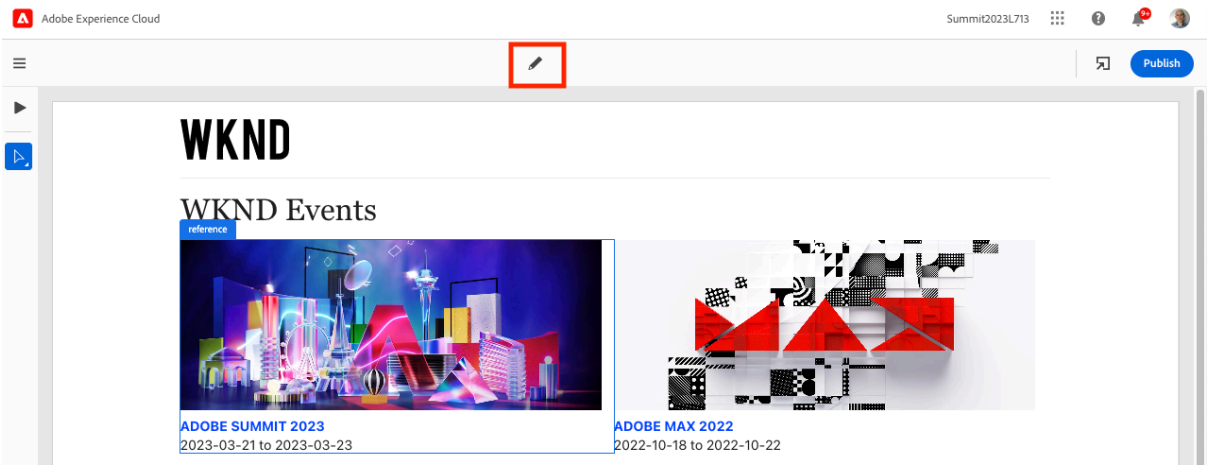
```
function EventItem(props) {
  const editorProps = useMemo(() => true && { itemID: "urn:aemconnection:" +
props?._path + "/jcr:content/data/master", itemType: "reference", itemfilter:
"cf"}, [props._path]);

  // ...
}
```

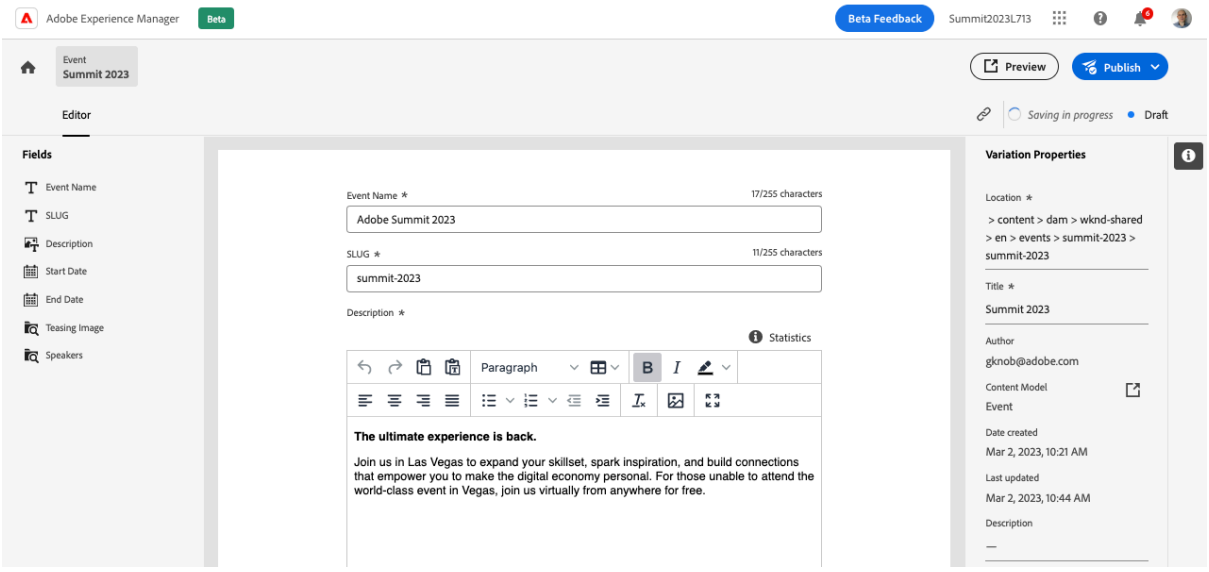
2. Go back to *Universal Editor* and switch to *Components* view by doing a long press on the second icon on the left (or via shortcut *V*)



3. Select the *Summit 2023* event and observe it now displays a reference frame around it and an *Edit* icon.



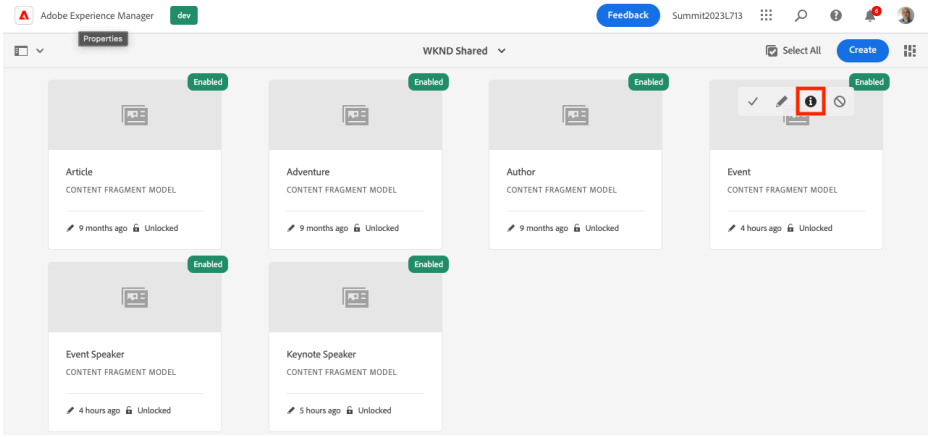
4. Click on the *Edit* icon, which opens that specific content fragment in the form-based editor.



Connecting Universal Editor as preview application in Content Fragment Editor

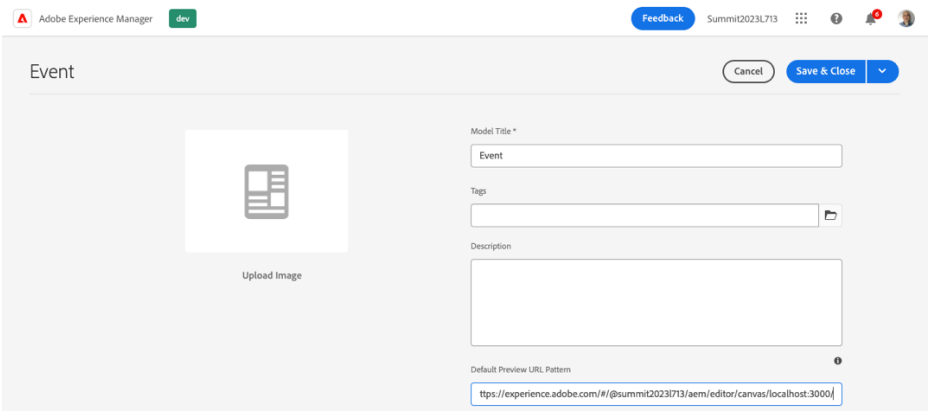
Getting a way to easily preview content created in form-based authoring is very practical for understanding impact of some changes.

1. Open the *Models* bookmark and open the *Properties* for the *Event* model



2. Specify a preview URL, pointing to the Universal Editor URL, then **Save & Close**

```
https://experience.adobe.com/#/@summit2023l713/aem/editor/canvas/localhost:3000/
```



3. Go back to the tab that had the *Summit 2023* event opened in *Content Fragment Editor*

4. Click on *Preview* button, which takes you back to the *Universal Editor*.

Next steps

Thank you

Thank you for participating! Please rate this lab in the Summit 2023 mobile app survey!

Additional resources

https://github.com/Adobe-Marketing-Cloud/Summit-2023-L713