Create Your Own Datalayer

Use Velocity data structures to streamline data management and simplify your code

Winston Churchill-Joell
Cascade User Conference 2025

Welcome, and thank you for joining this session on creating your own datalayer in Velocity. My name is Winston Churchill-Joell, and I'm the Director of Digital Services in the office of marketing and communications at Sarah Lawrence College. Over the next 30 minutes, we'll dive into how Velocity data structures can help you organize and maintain your code.

What we'll cover

The concepts

- **Create a data model** to sit between the rich interfaces and objects Cascade provides and your code
- **Reduce complexity in your macros** by moving more logic into the datalayer
- **Consolidate variables** to manage state

Some examples

- **Migrating existing code** to use a datalayer
- **Using JSON output in Velocity** to port an API-powered gallery to Cascade
- **Versatile applications for HashMap** data structures

We'll cover the concepts of modeling incoming data to take full advantage of the rich set of tools in Velocity that provide access to Cascade's internal APIs. We'll talk about how a data model or datalayer can reduce complexity and mitigate state management issues. We'll also take a look at few examples of this approach in action.

About your presenter(s)

- Winston Churchill-Joell, Director of Digital Services at Sarah Lawrence College since 2001
- Cascade Server for over 15 years
- Responsible for all things Cascade
- Passionate about digital accessibility, semantic markup, data structures, CSS, vanilla JS, and dogs🐾
- Leo is the cute one with the slightly bigger ears; he's passionate about anything that squeaks

A little more about me. I'm responsible for Cascade Server administration, development and user support. We've been in Cascade for over 15 years, using the platform to manage our public website and our course catalogue, among numerous other projects. I love working on the web and I care about making experiences that are accessible to everyone. I also have a more-than-passing interest in data structures, which inspired the work discussed in this talk. The cute guy perched on my shoulder there is Leo. If you hear general mayhem in the background, it's probably him. If there's an Amazon delivery, pandemonium will ensue; you've all been warned.

Project Files and Resources

- Github repo with code examples
- Download Cascade Server site archive
- Reference materials for this talk
- Don't worry, this slide will show again at the end of the presentation

SLC.EDU/CUC25

The QR code and URL here link through to a Github repo with all the files used for this talk. This includes the Cascade Server site archive that has the data definitions and velocity formats used to build the examples. And don't worry, I'll put up the QR code again at the end of the presentation.

So, what do I mean by *datalayer*?

Usually, in an application

- **Broker** the data connection (API)
- **Model the data**
- **Normalize** read / write operations

In the context of Cascade and Velocity

- **API** in the velocity context is handled by *Query* and *Locate* APIs, as well as the `$currentPage` object
- **Use data structures such as lists and hashMaps** to model the data from the API for consistency and repeat access
- **Normalize incoming data** to absorb changes at the source

So I should clarify what I mean by datalayer in the context of Cascade and Velocity. In an application context, the datalayer might handle API and database connections and model the data coming back from these sources to abstract away some complexity all with the purpose of normalizing the interface to the data for application logic.

From an API and data source perspective Velocity already functions in many ways like a datalayer, with tools such as the Query and Locate APIs and the $currentPage object facilitating the connection to the underlying data in Cascade.

So, what do I mean by *datalayer*?

```
Object type: com.hannonhill.cascade.api.adapters.PageAPIAdapter
Properties:
- allPermissionLevel: PermissionLevel
- assetId: String
- assetType: String
- class: Class
- contentType: ContentType
- createdBy: String
- createdOn: Date
- currentUserCanRead: boolean
- currentUserCanWrite: boolean
- dataDefinition: StructuredDataDefinition
- dataDefinitionId: String
- dataDefinitionPath: String
- folderOrder: int
- getStructuredDataNode(String): StructuredDataNode
- getStructuredDataNodeWithFieldId(String): StructuredDataNode
- getStructuredDataNodes(String): StructuredDataNode[]
- getStructuredDataNodesWithFieldId(String): List
- hideSystemName: boolean
- identifer: Identifier
- identifier: PathIdentifier
- isUserCanRead(String): boolean
- isUserCanWrite(String): boolean
```

```
- isUserCanWrite(String): boolean
- label: String
- lastModified: Date
- lastModifiedBy: String
- lastPublishedBy: String
- lastPublishedOn: Date
- link: String
- linkingAssets: List
- metadata: Metadata
- metadataSet: MetadataSet
- metadataSetId: String
- name: String
- parentFolder: Folder
- parentFolderIdentifier: PathIdentifier
- path: String
- persisted: boolean
- shouldBeIndexed: boolean
- shouldBePublished: boolean
- site: Site
- siteId: String
- siteName: String
- structuredData: StructuredDataNode[]
- tags: List
- xHTML: String
- xHTMLAsXMLElement: Element
```

These tools provide a generous interface for accessing any information about, or content from, any asset or collection of assets in Cascade. It's almost an embarrassment of riches, providing properties, methods, and complex structures for information retrieval and hooks for querying related assets and collections. Illustrated here is the output from the property tool's outputProperties() method executed on a page object. The property tool is your best friend when learning the available methods and properties of an object in the velocity context, whether that's a data structure or an API object. If you're not familiar yet, get acquainted with the propertytool; it's essentially interactive documentation and it will save you some frustration! If this output looks a little daunting, the point of this talk is to use the wealth of options here and from Cascade's other APIs to make something more tailored to your workflow.

So, what do I mean by *datalayer*?

Usually, in an application

- **Broker** the data connection (API)
- **Model the data**
- **Normalize** read / write operations

In the context of Cascade and Velocity

- **API** in the velocity context is handled by *Query* and *Locate* APIs, as well as the `$currentPage` object
- **Use data structures such as lists and hashMaps** to model the data from the API for consistency and repeat access
- **Normalize incoming data** to absorb changes at the source

So think of the datalayer as a means to abstract away some of the complexity of the available data we just looked at by creating your own data model. If the objects and collections returned by Cascade's internal APIs can be thought of as raw materials in their most versatile state, what I'm proposing could be considered a refining step that tailors these resources for your purposes. That refining step can be as unique to your workflow as your code is to your work. This approach can front load some of the decision making and reasoning about user-defined content, metadata, and settings, which keeps things simpler downstream using macros to focus on straightforward tasks instead of complex data transformations.

So, what do I mean by *datalayer*?

Usually, in an application

- **Broker** the data connection (API)
- **Model the data**
- **Normalize** read / write operations

In the context of Cascade and Velocity

- **API** in the velocity context is handled by *Query* and *Locate* APIs, as well as the `$currentPage` object
- **Use data structures such as lists and hashMaps** to model the data from the API for consistency and repeat access
- **Normalize incoming data** to absorb changes at the source

A datalayer can also act as a buffer between your macros and upstream changes in data definitions or other future changes, providing your code with a consistent interface, even if changes occur at the source, such us updates to Data Definition field names, or group structure. Part of our job as developers is to build things that can withstand the unexpected, but the other part of the gig is to build things that are resilient enough to enable change and support new development

Isn't this just more complexity?

The Challenge

- **Wired and and dynamic metadata** are accessed differently in the API and the syntax can get cumbersome
- **Structured data** can have a lot of complexity from the data def and implementations can vary over time
- **Temporary values and managing state** can get out of hand in a hurry
- **Passing data to macros** can get unwieldy with too many parameters
- **HTML** doesn't work for everything

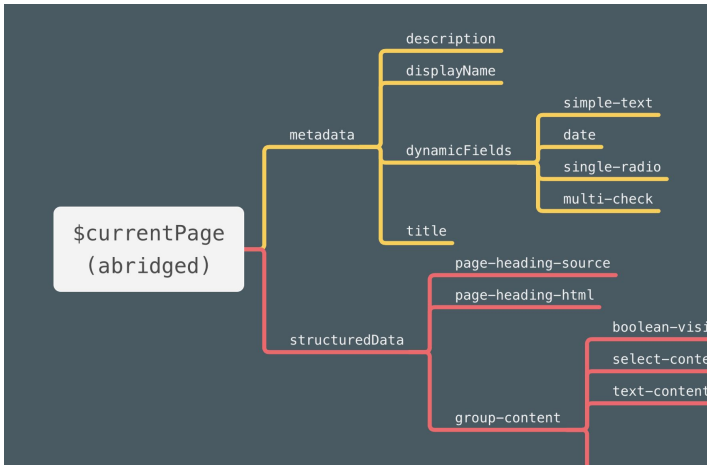
The Opportunity

- **Flatten wired and custom fields** to same depth and store multi-value fields as lists
- **Prep and organize structured data** for easier output and hand-off; normalize variations and changes
- **Structure temporary values** for better readability and state management
- **Use a single HashMap** to stand in for a growing number of **parameters in macros**
- **Make data portable** for hand off to other outputs such as JSON or server-side languages

So you might be thinking, isn't this just more complexity? For simple use cases, this approach is probably overkill, but in my experience, formats only grow in functionality and complexity over time. The number of macros in your formats will grow. Dynamic metadata fields are super useful and fast, so more of those are likely to be added. Data definitions pick up increasing complexity to handle new content requirements or provide content contributors with more options and better guardrails. A datalayer can scale with these changes and actually make them easier to plan out and execute.

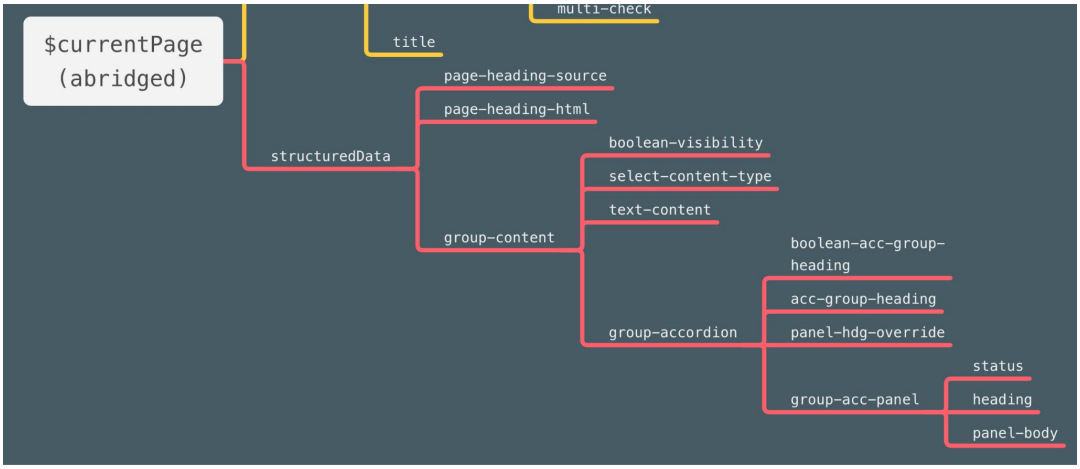
As for scale, I'm still using this approach in targeted cases, ramping up to a larger project. I'm certainly not the first person to promote this idea and I'd love to hear from anyone who's using anything similar in their work. Maybe you'll use a datalayer to clean up a few macros, or you might build out a complete implementation of the page object to hand off as JSON for a third party site search indexing tool. This latter scenario is what I'm working on and part of the inspiration for this idea.

Working with metadata



As a visual learner I started using a datalayer to help simplify my mental model of the page object. I began with dynamic metadata, looking to surface values as collections where needed and making them as easy to access as wired metadata fields. We use dynamic metadata heavily on our site and they're surfaced as content in our catalogue, news releases, and events calendars to name a few content types. They're also used as hooks for powering server-side queries to pull tagged items in as related content. As a result I found myself writing a lot of the same code repeatedly or handing off to macros to access and process these values which was getting harder to maintain.

Isn't this just more complexity?



Meanwhile on the structured data side, I wanted a unified model that was simpler to navigate with less syntax. As I started building out this model, I saw opportunities to reduce and apply certain fields that were used for conditional logic in the data definition. These fields made the user experience better for content contributors using our data definitions, but they added layers of logic and overhead in velocity. This was an opportunity to minimize some of that complexity. What we're looking at here is diagram of the data definition I created for my 2023 talk on accessible accordions. We'll dig into this further in the coming slides and then in the examples section.

Isn't this just more complexity?

The Challenge

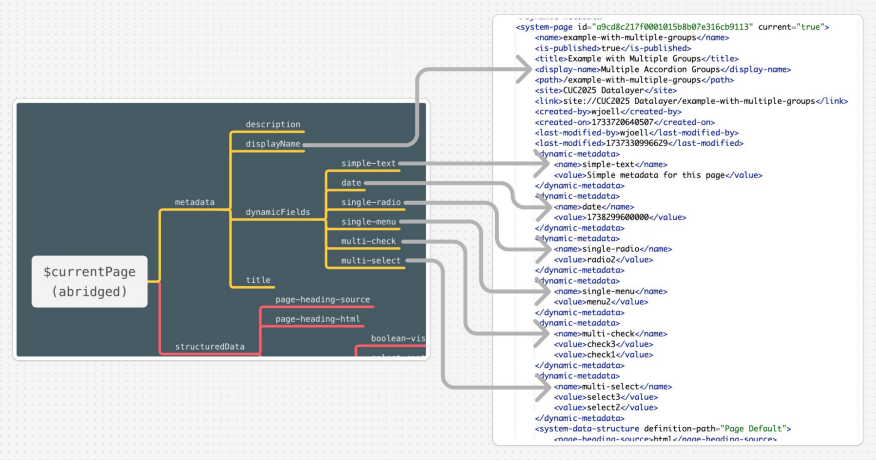
- **Wired and and dynamic metadata** are accessed differently in the API and the syntax can get cumbersome
- **Structured data** can have a lot of complexity from the data def and implementations can vary over time
- **Temporary values and managing state** can get out of hand in a hurry
- **Passing data to macros** can get unwieldy with too many parameters
- **HTML** doesn't work for everything

The Opportunity

- **Flatten wired and custom fields** to same depth and store multi-value fields as lists
- **Prep and organize structured data** for easier output and hand-off; normalize variations and changes
- **Structure temporary values** for better readability and state management
- **Use a single HashMap** to stand in for a growing number of **parameters in macros**
- **Make data portable** for hand off to other outputs such as JSON or server-side languages

So, let's take a look at a few opportunities where a datalayer can make a difference starting with metadata. As I mentioned, in our case we make extensive use of metadata field types with dozens of values that might be a bit overwhelming for this example.

Wired and custom metadata



So instead, I’m using some mock fields with truly uninspired names that represent several field types such as date, radio, menu, checkbox, and multiselect.

Isn't this just more complexity?

The Challenge

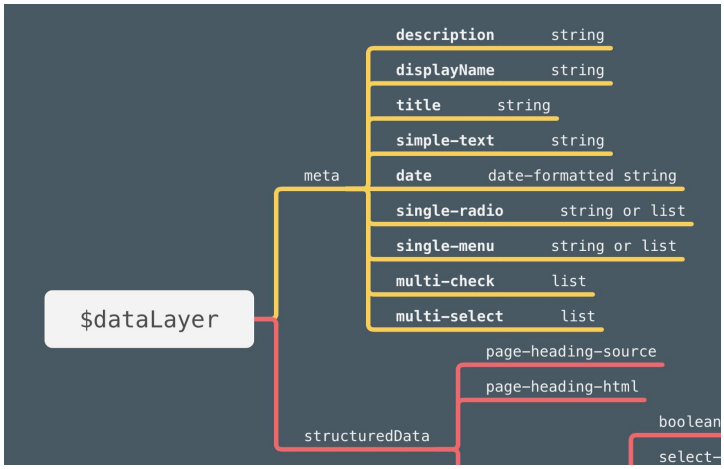
- **Wired and and dynamic metadata** are accessed differently in the API and the syntax can get cumbersome
- **Structured data** can have a lot of complexity from the data def and implementations can vary over time
- **Temporary values and managing state** can get out of hand in a hurry
- **Passing data to macros** can get unwieldy with too many parameters
- **HTML** doesn't work for everything

The Opportunity

- **Flatten wired and custom fields** to same depth and store multi-value fields as lists
- **Prep and organize structured data** for easier output and hand-off; normalize variations and changes
- **Structure temporary values** for better readability and state management
- **Use a single HashMap** to stand in for a growing number of **parameters in macros**
- **Make data portable** for hand off to other outputs such as JSON or server-side languages

As I mentioned, I wanted dynamic fields as easily accessible as wired fields, so I created a meta node and stored everything at the same level, choosing data types that best suited each field

Wired and custom metadata



In this example, all meta data fields are now a direct child of meta. The single-radio and single-menu fields could be strings, but for the sake of future-proofing, it might be smarter to make them lists, even if only a single value is possible for the time being. It takes a little doing, but metadata field types can be altered with a database query if necessary, and this is something Hannon Hill support can help you out with. For example, I have converted checkboxes to multi-select menus, but that conversion could be made to a radio field.

Isn't this just more complexity?

The Challenge

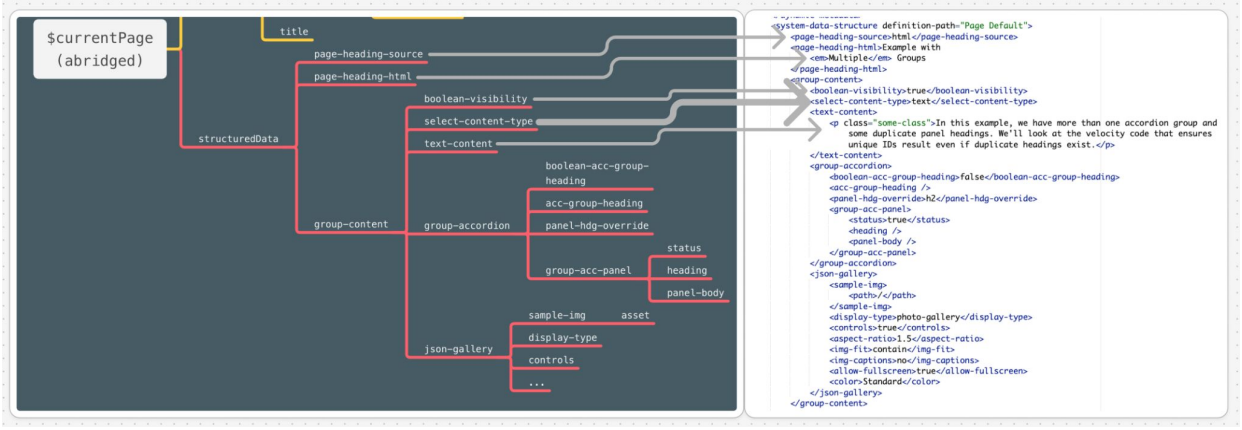
- **Wired and and dynamic metadata** are accessed differently in the API and the syntax can get cumbersome
- **Structured data** can have a lot of complexity from the data def and implementations can vary over time
- **Temporary values and managing state** can get out of hand in a hurry
- **Passing data to macros** can get unwieldy with too many parameters
- **HTML** doesn't work for everything

The Opportunity

- **Flatten wired and custom fields** to same depth and store multi-value fields as lists
- **Prep and organize structured data** for easier output and hand-off; normalize variations and changes
- **Structure temporary values** for better readability and state management
- **Use a single HashMap** to stand in for a growing number of **parameters in macros**
- **Make data portable** for hand off to other outputs such as JSON or server-side languages

As I mentioned, we've been in Cascade for quite some time and our data definitions have evolved over the years. With the introduction of field IDs, it's much more feasible to move and even rename data definition fields, but the thought of hunting down and updating all the references to these fields in our code is daunting. Once again, I started to see how an abstraction layer that manages these references in one place, and one place only, could make a huge difference.

Structured data: logic and output



This diagram illustrates an example structured data hierarchy with a blend of content fields and control fields that inform display logic in the data definition and signal to the developer which content fields are relevant for output. Keeping track of these control fields and what they mean can get tricky, especially as more are added and the complexity of the data definition grows.

Isn't this just more complexity?

The Challenge

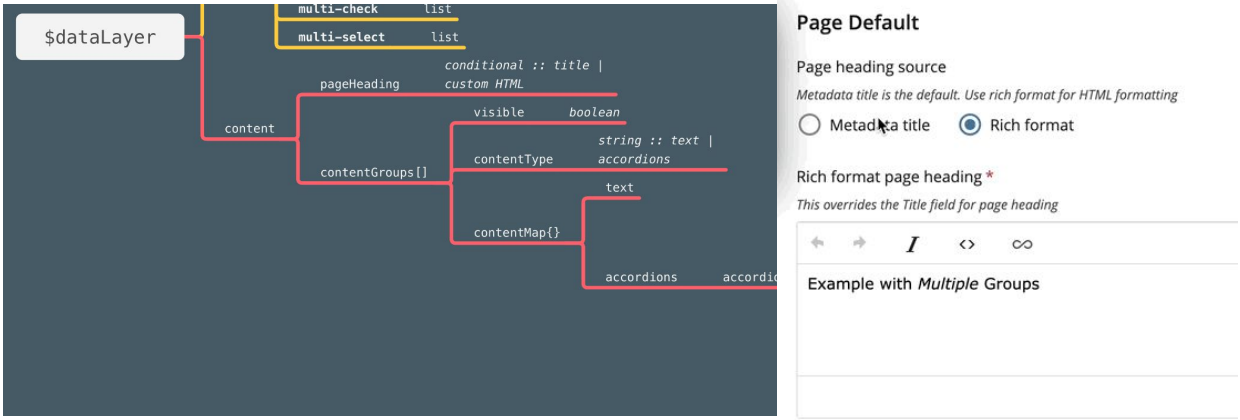
- **Wired and and dynamic metadata** are accessed differently in the API and the syntax can get cumbersome
- **Structured data** can have a lot of complexity from the data def and implementations can vary over time
- **Temporary values and managing state** can get out of hand in a hurry
- **Passing data to macros** can get unwieldy with too many parameters
- **HTML** doesn't work for everything

The Opportunity

- **Flatten wired and custom fields** to same depth and store multi-value fields as lists
- **Prep and organize structured data** for easier output and hand-off; normalize variations and changes
- **Structure temporary values** for better readability and state management
- **Use a single HashMap** to stand in for a growing number of **parameters in macros**
- **Make data portable** for hand off to other outputs such as JSON or server-side languages

Modeling the structured data had very different considerations than the metadata. Metadata fields have different types, but their values aren't interconnected in the way that data definition fields can be. Those dependencies are visually obvious in the UI especially when control logic is applied. In the case of structured data, I wanted to limit the number of fields coming into the data layer and pre-process as much logic as possible to avoid passing that work down to all the macros downstream using this data, where the same processing and logical operations would be repeated over and over again.

Structured data: prep and organize



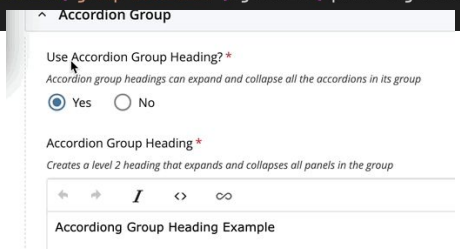
Starting with a root key of content for my structured data I started mapping out the content that I wanted easy access to. In this example, instead of storing all the data, I quickly realized that it made more sense to have a single field for pageHeading that was conditionally set in the datalayer. There was no point in bringing over the control field and the nested structure when all I needed was the relevant version of the content. Again, this is more obvious in the data definition UI animation shown here.

Structured data: prep and organize

Digging into more complex nested structures such as the accordion group, once again, the relationship between the groupHeading and panelHeadingLevel fields were better resolved in the datalayer rather than in macros. The relationship was obvious in the data definition UI but less so when dealing with the values in velocity.

Structured data: prep and organize

```
#elseif ( $content.type == "accordion" )
    #set ( $content.groupAccordion = {} )
    #if ( $group.getChild("group-accordion").getChild("boolean-acc-group-heading").value == "true"
        && !$_PropertyTool.isEmpty($group.getChild("group-accordion").getChild("acc-group-heading").value))
        #set ( $content.groupAccordion.groupHeading = $group.getChild("group-accordion").getChild("acc-group-heading").value )
        #set ( $content.groupAccordion.panelHeadingLevel = "h3" )
    #else
        #set ( $content.groupAccordion.groupHeading = false )
        #set ( $content.groupAccordion.panelHeadingLevel = $group.getChild("group-accordion").getChild("panel-hdg-override").value )
    #end
```



This made the case for doing the logic once in the datalayer and passing along the result to everything that uses it, rather than relying on every macro to correctly apply that logic. Should a change ever be required, it's also much easier to change the logic in one place.

We'll dig into the structure and syntax of the datalayer in a few minutes, but I wanted to illustrate the dual role that the groupHeading node of the datalayer is playing here. If the conditions are met, it takes on the value of the accordion group heading from the page, otherwise it's false. This is important because a boolean value is unambiguous and simpler to test for, which means less clutter in your macro. We're also tying the panelHeadingLevel value to the groupHeading conditions. If the group heading is present, the panelHeadingLevel must be h3 to follow the level 2 group heading. If we ever decide to change that rule, we only have to change it here once. Conversely, If the conditions aren't met and groupheading is set to false, we're then looking to the settings chosen by the content manager, which are h2 by default thanks to the data definition.

Structured data: logic and output

```
#****
    @param ::   $apiNode            :: structured data node
    @param ::   $classes            :: string - class name(s) for container attribute
    *#
#macro ( renderAccordionGroup $apiNode $classes )
<div class="${classes}">
    #if ( $apiNode.getChild('boolean-acc-group-heading').textValue == 'true' )
        <h2 class="accordion-heading" id="#stringToID($apiNode.getChild('acc-group-heading').textValue)"><span>$apiNode.getChild
            ('acc-group-heading').textValue</span></h2>
        #set ( $panelHeadingLevel = 'h3' )
    #else
        #set ( $panelHeadingLevel = $apiNode.getChild('panel-hdg-override').textValue )
    #end
    #foreach ( $item in $apiNode.getChildren('group-acc-panel') )
        #if ( $item.getChild('status').textValue != 'off' && !$_PropertyTool.isNull($item.getCh
            !$_PropertyTool.isNull($item.getChild('panel-body').textValue) )
        <div class="accordion-item" id="acc-${foreach.parent.index}${foreach.index}-#stringToID
            ).textValue)">
            <${panelHeadingLevel} class="panel-heading" data-accordion-collapsed="${item.getChil
                -${foreach.parent.index}${foreach.index}-#stringToID($item.getChild('heading').t
                ('heading').textValue</span></${panelHeadingLevel}>
            <div aria-labelledby="hdg-${foreach.parent.index}${foreach.index}-#stringToID($item
                class="accordion-content" role="region">
                $item.getChild('panel-body').textValue
            </div>
        </div>
        #else
            <!-- skip panel -->
        #end
    #end
</div>
#end
```

```
#****
    @param ::   $map            :: HashMap
    @param ::   $classes            :: string - class name(s) for container attribute
    *#
#macro ( renderAccordionGroup $map $classes )
<div class="${classes}">
    #if ( $map.groupAccordion.groupHeading )
        <h2 class="accordion-heading" id="#stringToID($map.groupAccordion.groupHeading)"><span>$map.groupAccordion.groupHeading
            </span></h2>
    #end
    #foreach ( $item in $map.groupAccordion.panels )
        #if ( $item.status != 'off' && !$_PropertyTool.isEmpty($item.heading) && !$_PropertyTool.isEmpty($item.panelBody) )
        <div class="accordion-item" id="acc-${foreach.parent.index}${foreach.index}-#stringToID($item.heading)">
            <${map.groupAccordion.panelHeadingLevel} class="panel-heading" data-accordion-collapsed="${item.status}" id="hdg
                -${foreach.parent.index}${foreach.index}-#stringToID($item.heading)"><span>$item.heading</span></${map
                .groupAccordion.panelHeadingLevel}>
            <div aria-labelledby="hdg-${foreach.parent.index}${foreach.index}-#stringToID($item.heading)" class="accordion
                -content" role="region">
                $item.panelBody
            </div>
        </div>
        #else
            <!-- skip panel -->
        #end
    #end
</div>
#end
```

Looking at both versions of the macro, I realize the difference between the earlier version on the left and later version on the right may not look dramatic, but there are key differences that we'll have a chance to review together in the examples later in this talk. In the new version the test for groupHeading is simplified. If it's false it's skipped and we move on. If there's content, we use it and we're no longer setting any values for panel heading in that test. As discussed, the panel heading level for the accordions in this example are set in the datalayer. All of the dependency between these values is also resolved in the datalayer. Throughout the rest of the macro we get to take advantage of a bit of syntactic sugar from the hashmap dot syntax, which cleans up some of the clutter and makes the code more readable and easier to reason about.

Isn't this just more complexity?

The Challenge

- **Wired and and dynamic metadata** are accessed differently in the API and the syntax can get cumbersome
- **Structured data** can have a lot of complexity from the data def and implementations can vary over time
- **Temporary values and managing state** can get out of hand in a hurry
- **Passing data to macros** can get unwieldy with too many parameters
- **HTML** doesn't work for everything

The Opportunity

- **Flatten wired and custom fields** to same depth and store multi-value fields as lists
- **Prep and organize structured data** for easier output and hand-off; normalize variations and changes
- **Structure temporary values** for better readability and state management
- **Use a single HashMap** to stand in for a growing number of **parameters in macros**
- **Make data portable** for hand off to other outputs such as JSON or server-side languages

Although I try to avoid setting values in macros, it's sometimes useful to set temporary values for ease of passing data to child macros. The reason I try to avoid setting values or transforming data in macros has to do with state management across the velocity script. All variables are global in the velocity context, so it's critical to reset them in looping operations and in general to limit the number of places variables are instantiated. Let's take a look at a hypothetical example.

Passing around data and managing state

```
#**
    @param $collection :: nodeset
    *#
#macro processCollectionWithParams($collection)
<div class="collection">
    #foreach($item in $collection)
        ## reset parameters so that values are not carried over from previous iteration
        #set($parameter1 = "")
        #set($parameter2 = "")
        #set($parameter3 = "")
        #set($parameter4 = "")
        ## if any field is null, $parameter will not update, hence the reset
        #set($parameter1 = $item.getChild("field1").value)
        #set($parameter2 = $item.getChild("field2").value)
        #set($parameter3 = $item.getChild("field3").value)
        #set($parameter4 = $item.getChild("field4").value)
        #processNode($parameter1 $parameter2 $parameter3 $parameter4)
            ##do something with parameters
            ##macro signature is rigidly defined
        #end
    #end
</div>
#end
```

```
aram $collection :: nodeset

 processCollectionWithMap($collection)
lass="collection">
oreach($item in $collection)
  ## reset parameters map
  #set($parameters = {})
  ## if any field is null, key will not be added to map
  ## use $parameters.getOrDefault("key", "default") to avoid errors
  #set($parameters.parameter1 = $item.getChild("field1").value)
  #set($parameters.parameter2 = $item.getChild("field2").value)
  #set($parameters.parameter3 = $item.getChild("field3").value)
  #set($parameters.parameter4 = $item.getChild("field4").value)
  #processNode($parameters)
      ##do something with parameters map
      ##macro signature is flexible
  #end
nd
```

Looking at these two mock examples, it's easy to see how the left hand version is getting harder to maintain.

We're resetting each parameter to avoid data leaking through from a prior iteration of the loop.

Then we're passing these parameters to a child macro to process each iteration. The signature for this macro is capped at 4 parameters and would need to be updated in all the places it's used if more parameters need to be added.

Passing around data and managing state

```
#**
    @param $collection :: nodeset
    *#
#macro processCollectionWithParams($collection)
<div class="collection">
    #foreach($item in $collection)
        ## reset parameters so that values are not carried over from previous iter
        #set($parameter1 = "")
        #set($parameter2 = "")
        #set($parameter3 = "")
        #set($parameter4 = "")
        ## if any field is null, $parameter will not update, hence the reset
        #set($parameter1 = $item.getChild("field1").value)
        #set($parameter2 = $item.getChild("field2").value)
        #set($parameter3 = $item.getChild("field3").value)
        #set($parameter4 = $item.getChild("field4").value)
        #processNode($parameter1 $parameter2 $parameter3 $parameter4)
            ##do something with parameters
            ##macro signature is rigidly defined
        #end
    #end
</div>
#end
```

```
#**
    @param $collection :: nodeset
    *#
#macro processCollectionWithMap($collection)
<div class="collection">
    #foreach($item in $collection)
        ## reset parameters map
        #set($parameters = {})
        ## if any field is null, key will not be added to map
        ## use $parameters.getOrDefault("key", "default") to avoid errors
        #set($parameters.parameter1 = $item.getChild("field1").value)
        #set($parameters.parameter2 = $item.getChild("field2").value)
        #set($parameters.parameter3 = $item.getChild("field3").value)
        #set($parameters.parameter4 = $item.getChild("field4").value)
        #processNode($parameters)
            ##do something with parameters map
            ##macro signature is flexible
        #end
    #end
</div>
#end
```

By contrast using a hashmap, we have one object to reset. The potential pitfall here is that null assignments simply don't get added, so the key won't exist. In my opinion this is a feature, because instead of testing the field for empty with if/else, you can use the .getOrDefault method to supply a default value for a missing key, even if you want to use an empty string as the fallback. We also have flexibility to add or remove parameters now as properties of a single object without having to update all the calls to the child macro.

Isn't this just more complexity?

The Challenge

- **Wired and and dynamic metadata** are accessed differently in the API and the syntax can get cumbersome
- **Structured data** can have a lot of complexity from the data def and implementations can vary over time
- **Temporary values and managing state** can get out of hand in a hurry
- **Passing data to macros** can get unwieldy with too many parameters
- **HTML** doesn't work for everything

The Opportunity

- **Flatten wired and custom fields** to same depth and store multi-value fields as lists
- **Prep and organize structured data** for easier output and hand-off; normalize variations and changes
- **Structure temporary values** for better readability and state management
- **Use a single HashMap** to stand in for a growing number of **parameters in macros**
- **Make data portable** for hand off to other outputs such as JSON or server-side languages

Much of what we do is geared toward HTML output. However, there are other applications for the content we manage in Cascade. Often those applications require content to be structured in a portable data format, and arguably the most popular format these days is JSON, or JavaScript Object Notation. One great advantage of modeling your data in a HashMap is that we have the means of serializing that structure as JSON, making your work portable for hand off to client-side JavaScript or server-side languages.

What is a HashMap?

- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON

```
#set($myMap = {})

## or initialize with values
#set($myMap = {
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
})

## use temp var to suppress output from put method
#set($_void = $myMap.put("key1", "value1"))

## alternative dot syntax
#set($myMap.key2 = "value2")

## alternative bracket syntax
#set($myMap["key3"] = "value3")
```

So we've been talking about hashmaps and before I show a few examples, I should quickly review what a HashMap is for anyone not familiar with them.

HashMaps are data structures modeled with key/value pairs. If you're familiar with Objects in JavaScript, or dictionaries in Python, or associative arrays in PHP, these are all similar to hashmaps.

Maps enforce unique keys which is why they are so performant even at large sizes.

What is a HashMap?

- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON

```
#set($myMap = {})

## or initialize with values
#set($myMap = {
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
})

## use temp var to suppress output from put method
#set($_void = $myMap.put("key1", "value1"))

## alternative dot syntax
#set($myMap.key2 = "value2")

## alternative bracket syntax
#set($myMap["key3"] = "value3")
```

As you can see in the these examples, the syntax for setting values is very flexible. You can declare an empty map or immediately populate an entire map upon declaration. There is a put method which is perfectly valid to use, but it does have the side effect of returning a value to the output. The easiest way to avoid that side effect is to assign the put operation to a temporary value. In my opinion the dot syntax is the most intuitive and easy to use, but requires that your key names don't have any spaces and ideally are only alphaNumeric. If you don't have control over the names of your keys, you can use the bracket syntax as an alternative to the put() method.

What is a HashMap?

- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON

```
18
19  ## access values
20  $myMap.get("key1")
21
22  ## alternative dot syntax
23  $myMap.key2
24
25  ## alternative bracket syntax
26  $myMap["key3"]
27
28  ## default value
29  $myMap.getOrDefault("key1", "default")
30
31  ## remove key
32  #set($_void = $myMap.remove("key1"))
33
34
35
36
```

The syntax for retrieving values is equally flexible, including the getOrDefault method for setting a fallback value in the event that a key doesn't exist.

As I mentioned, this method functions like a shorthand if/else, so it's actually useful to remove a key rather than set it to an empty string.

What is a HashMap?

- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON

```
36
37  ## check if key exists
38  $myMap.containsKey("key1")
39
40  ## check if value exists
41  $myMap.containsValue("value1")
42
43  ## get all keys
44  $myMap.keySet()
45
46  ## get all values
47  $myMap.values()
48
49  ## get all entries
50  $myMap.entrySet()
51
52
53
54
```

As you can see HashMaps have a rich API with a number of ways to query and iterate through them. One interesting thing to note is that the .keySet() method returns a Set, which is another useful data structure that is essentially a list that enforces unique values. If you need to create a set object for some use case, you simply create a hashmap of keys with empty strings as values.

What is a HashMap?

- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON

```
54
55  ## Values can be a variety of types such as booleans,
56  ## strings, numbers, lists, and HashMaps
57  #set($myMap = {
58      "key1": true,
59      "key2": "value2",
60      "key3": 3,
61      "key4": [1, 2, 3],
62      "key5": {
63          "nestedKey1": "nestedValue1",
64          "nestedKey2": "nestedValue2"
65      }
66  })
```

HashMaps and lists can store any data type or object including Cascade API objects. Using maps and lists together you can model robust structures that can provide a rich interface for your macros to use.

What is a HashMap?

- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON

```
54
55  ## Values can be a variety of types such as booleans,
56  ## strings, numbers, lists, and HashMaps
57  #set($myMap = {
58      "key1": true,
59      "key2": "value2",
60      "key3": 3,
61      "key4": [1, 2, 3],
62      "key5": {
63          "nestedKey1": "nestedValue1",
64          "nestedKey2": "nestedValue2"
65      }
66  })
```

Because values can be retrieved by keys, even large and deeply nested maps are easy to navigate and fast to work with.

What is a HashMap?

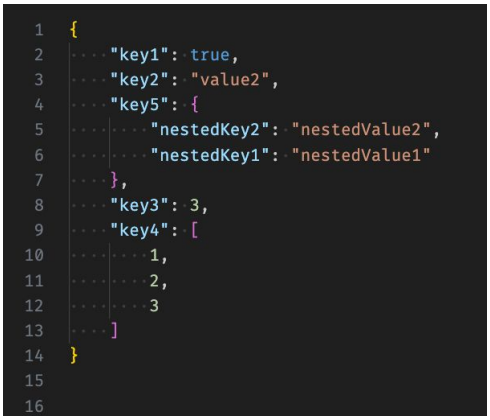
- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON

```
54
55  ## Values can be a variety of types such as booleans,
56  ## strings, numbers, lists, and HashMaps
57  #set($myMap = {
58      "key1": true,
59      "key2": "value2",
60      "key3": 3,
61      "key4": [1, 2, 3],
62      "key5": {
63          "nestedKey1": "nestedValue1",
64          "nestedKey2": "nestedValue2"
65      }
66  })
67
68  $_SerializerTool.toJson($myMap)
69
70
```

With the Serializer Tool's json output method

What is a HashMap?

- A HashMap is a collection of one or more key/value pairs
- Keys must be unique
- Flexible syntax, powerful methods
- Values can be a variety of types such as booleans, strings, numbers, lists, and HashMaps
- Can scale to large data sets yet remain fast for storage and retrieval
- Can be output to JSON



```
{
    "key1": true,
    "key2": "value2",
    "key5": {
        "nestedKey2": "nestedValue2",
        "nestedKey1": "nestedValue1"
    },
    "key3": 3,
    "key4": [
        1,
        2,
        3
    ]
}
```

you can output your map to JSON for use in other applications

Some examples...

- Revisit the accordions project
- Ported JSON gallery
- Table sorting
- SVG library

Let's look at a few examples. The first is an update to the project from my 2023 Cascade talk about accessible accordions, this time using a datalayer.

The second is a proof of concept, porting a JSON-powered gallery to a local implementation inside Cascade

The last two examples show the versatility of maps in somewhat surprising ways.

Accordions Project

- Datalayer
- Updated macros
- Let's break some stuff

Now that we all know what hashmaps look like and we have some familiarity with the approach, let's look at an example datalayer setup and how that impacts some of the macros downstream, comparing before and after. We'll also break some stuff in the data definition on purpose to see how the datalayer can absorb those changes.

Ported JSON gallery

- Proof of concept
- JSON data output
- Uses srcset and MIME for sizes and formats
- CDN-delivered assets

I have to admit this was mostly an experiment to see if it was doable. I don't know how practical this particular port is, but there are definitely other gallery libraries that work with JSON, so it's a worthwhile proof of concept. This gallery library is a home grown library that I know well, which is why I chose it. The library relies on CDN -delivered assets and I decided to use the same model here, but upload smaller files to Cascade to manage alt text and image order. Let's take a look.

Fun with HashMaps

- Sorting tables
- SVG icon library lookup

The table sorting project is a couple of years old at this point, but it taught me something interesting about the sort tool, which is that it can be used on a list of hashmaps, if you're careful about managing the data types of corresponding values. It's tempting to think of this as sorting a number of rows in a table, but it's more accurate to say that we're sorting a number of single-row tables that all have the same columns. From there it's easy to output a table. This example incorporates user-managed settings for sort criteria. After that, we'll take a quick look at using a hashmap to manage a library of svg icons. Let's take a look.

Thank you

And once again, project files and resources…

- Github repo with code examples
- Download Cascade Server site archive
- Reference materials for this talk

SLC.EDU/CUC25

Thanks for your time and sticking with me. I know this is a bit of an odd topic, but I hope it was useful. Ping me in the Hannon Hill Slack to tell me if you use these concepts or do something similar. I'd love to hear about it. Once again, here's the QR code or the URL to take you to the repo for this talk. You'll find code examples and a full Cascade site archive for download. I'm happy to take any questions!