

**Adding Sorting and Grouping to the Mavo Framework for End User Web Application Authoring**

by Daniel Sanchez

S.B., C.S. M.I.T., 2016

Submitted to the  
Department of Electrical Engineering and Computer Science  
In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

February 2018

© 2018 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

**Signature redacted**

Author:

Department of Electrical Engineering and Computer Science  
February 2, 2018

**Signature redacted**

Certified by:

David R. Karger, Professor of Computer Science and Engineering, Thesis Supervisor  
February 2, 2018

**Signature redacted**

Accepted by:

Christopher Terman, Chairman, Masters of Engineering Thesis Committee





Adding Sorting and Grouping to the Mavo Framework for End User Web Application  
Authoring

by Daniel Sanchez

Certified by David Karger

Submitted to the Department of Electrical Engineering and Computer Science

February 2, 2018

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in  
Electrical Engineering and Computer Science

## **ABSTRACT**

Sorting and grouping in Mavo gives developers further control as to how the data shown on their webpage will be displayed. Given a collection of elements with the same underlying data schema, users can choose a property of these elements whose value should be used as a sorting criterion for what order the elements appear in. Expanding on this, Mavo users can also view a collection of data in groups, where group headings appear above a subset of the items denoting a shared characteristic of the items. With this additional functionality, Mavo developers can now author more powerful web applications that react dynamically to user input to display their data in a more meaningful way. All this new functionality comes with the existing ease-of-use of Mavo, requiring only an HTML attribute to specify how the data should be displayed, and how the view should update should changes in the data occur.



# ACKNOWLEDGEMENTS

I would like to extend a note of thanks to Lea Verou for her constant help and advice throughout the development of these features. Lea is one of the original creators of Mavo and by far the largest contributor to the Mavo codebase. With her expertise of Mavo, she helped tremendously in building off of Mavo's existing functionality. In addition to her behind the scenes help, Lea also gave a great deal of insight that affected the final user experience of these features.

Additionally, I would like to thank my advisor, David Karger, for his input and help at various times throughout the development of these features. We had our fair share of whiteboard and online discussions with Lea, which ultimately helped greatly in shaping the final outcome of this thesis.



# TABLE OF CONTENTS

<b>ABSTRACT</b>	<b>3</b>
<b>ACKNOWLEDGEMENTS</b>	<b>5</b>
<b>TABLE OF CONTENTS</b>	<b>7</b>
<b>LIST OF FIGURES</b>	<b>9</b>
<b>INTRODUCTION</b>	<b>11</b>
<b>RELATED WORK</b>	<b>18</b>
Mavo	18
Similar Solutions	18
AngularJS	18
Exhibit	21
<b>USER EXPERIENCE</b>	<b>24</b>
Sorting and Grouping Using Attributes	24
Syntax	25
Dynamic Properties	26
Grouping	26
Sorting and Grouping Using Functions	29
Syntax	29
Use Cases	31
Triggers	31
<b>IMPLEMENTATION</b>	<b>35</b>
Sorting as a Plugin	35
MavoScript Functions	35
Sorting	36
Arguments and Data Processing	36
User Defined Comparison Function	37
Sorting Order Specification	38
Null Prefixed Sorting	38
Grouping	39

DOM Manipulation Functions	40
Setup	40
Sorting the DOM	41
Grouping the DOM	41
Cached Criteria	44
Triggers Implementation	44
<b>FUTURE WORK</b>	<b>48</b>
Nested Grouping	48
Automatic Group Headings for Specific Elements	48
Support for More Powerful Grouping Criteria	49
Persisting the Sorting Order	49
<b>CONCLUSION</b>	<b>52</b>
<b>BIBLIOGRAPHY</b>	<b>53</b>



# LIST OF FIGURES

Figure 1.	Syntax for statically sorting a collection by multiple properties	20
Figure 2.	Syntax for dynamically sorting a collection by a single property	20
Figure 3.	Using mv-sort with multiple properties	24
Figure 4.	Automatic group heading elements, with developer code at the top, and resulting HTML at the bottom	28
Figure 5.	Using sort function in mv-value	30
Figure 6.	Control flow while traversing output of groupBy function in groupDOM	43



# INTRODUCTION

Processing and displaying data is largely an inherent need of web applications today. In the development process of these applications, developers use various strategies and tools to better manage this data while maintaining a clean user experience for those interacting with their application. This is where a variety of web application frameworks come into play, each seeking to ease the web development process in their own way. One common data pattern that developers need to deal with is the processing of repeated instances of a data schema. In the view of the application, these will typically be seen as various types of lists, with each item of the list displaying the characteristics of one such instance. For example, perhaps we are developing a restaurant search application, and need to display the relevant restaurants as a list of items. Each item in this list would perhaps display the name of the restaurant, its location, and its phone number, among a variety of other restaurant data. Creating each of the associated DOM elements with vanilla JavaScript would quickly get tedious for developers. As such, many of the aforementioned frameworks give users an easier way to create these lists, requiring developers to simply specify the template of each element in the list, as well as where the corresponding data should be displayed in that element.

From the user standpoint, these lists will always be presented to the user in some default order. However, users oftentimes will have the need to view these items in a different order, giving them a better context of the presented data. This is where sorting is a necessity in many web applications. The ability to sort data can solve a variety of use cases on the user side. Sorting can give users the power to quickly see more relevant data at the top of the list, can give a better comparative context of the data being presented, and can also be used as an assistant to searching should they choose to scroll through their data. To meet these needs, developers typically sort the data on

the JavaScript side, before taking this sorted data and presenting it to the user. Some web frameworks also provide constructs that can simplify the sorting process.

However, as it stands, creating these interfaces generally requires some programming knowledge, namely in JavaScript, and a basic understanding of how to process data. This can act as a barrier to entry for web designers with experience creating interfaces in HTML and CSS, but little to no experience with programming languages and data processing. Mavo [1] is a web framework that seeks to bridge this gap, by empowering developers to create data-driven web applications without writing any JavaScript. Mavo also enforces a straightforward user experience, by permitting users to edit data right where they see that data displayed. Mavo has proven to be a viable solution for this audience of developers in many use cases, but previously, Mavo did not have a way to sort data without writing JavaScript. This thesis presents sorting in Mavo, through the use of new Mavo constructs that give developers a concise yet powerful way to specify how a collection of data should be ordered. Building on top of sorting, we also present grouping, a common added convenience to many list interfaces. Grouping gives Mavo developers the ability to specify a criterion based on an item data schema that should present a visual grouped representation of the items when displayed in the web application. This visual representation is done in the form of group headers, which Mavo can automatically insert into the DOM at the beginning of any group.

We have provided the following constructs to developers for sorting and grouping functionality in Mavo: a `sort` function, a `groupBy` function, an `mv-sort` attribute, and an `mv-groupby` attribute. Each of these constructs has its own use case, which we will describe in detail in the following sections.

As these features are built on top of Mavo, we outline various facets of the Mavo framework in order to provide a better context for work described in this paper:

## Property

While properties can be used as a term somewhat generally within software, with respect to Mavo, properties are elements designated to contain important data that should be savable, editable, and/or used in expressions. This is done by attaching a `property` attribute to the element where the data should be displayed and edited. The value of this attribute designates a key for data to be stored, while the contents of the element designate a value in a key-value mapping that is saved. We will refer to these elements as property elements, and refer to the key-value mapping of data that this element represents as the property. More specifically, the key will be referred to as the property name, and the value as the property value.

## Expression

Expressions in Mavo are portions of the HTML wrapped in square brackets (e.g. `[ . . . ]`), with the exception of any string in an `mv-value` attribute, which is automatically considered an expression without the need for brackets. They are used as ways to pass Mavo variables and evaluate small bits of code in places where a static string would not convey the necessary information. The language used within expressions is known as MavoScript, which provides some built-in functions and variables, access to properties as variables, and has small syntactic differences from JavaScript. In addition, vanilla JavaScript can also be used in expressions for advanced users. While expressions are commonly used within HTML attributes, they can be used to populate HTML content as well. Expressions work well with the sorting and grouping features presented here, by allowing a clean way to create sorted or grouped interfaces where the criterion for sorting and grouping can change.

## Mavo Node

Mavo Nodes are Mavo's base representation of a variety of Mavo elements. They are always associated with a DOM element, typically property elements, but also can represent the root of the Mavo application. They are represented in the codebase with the `Mavo.Node` class.

## Collection

Collections are Mavo's way of representing repeated instances of a data schema. On the frontend, collections are specified with the `mv-multiple` attribute, and the data for the collection is dependent on the structure of the `mv-multiple` element. The `mv-multiple` element denotes the template element of the collection: the element which each respective data item of the collection clones before inserting the appropriate data into the cloned elements and getting appended into the DOM. The structure of each data item is determined by the structure of the collection's template element, using nested property elements to denote nested properties in the data schema for each individual item. Collections are represented in the codebase with the `Mavo.Collection` class, and are subclasses of `Mavo.Node`.

## Using mv-value

The `mv-value` attribute is used in Mavo to populate an element's data with the result of an expression. The populated data is typically some primitive that the expression has evaluated to. However, `mv-value` can also be used to process arrays of data. Doing this requires the structure of the property elements in the `mv-value` element to match the data schema of the items in the expression's resulting array. An example of this is seen in Figure 5 later in the paper.

The basic format of using each of the sorting and grouping constructs follows a similar pattern: the collection being operated on is specified (either explicitly or implicitly), followed by a specification of the sorting or grouping criterion. At this point, the

developer's job is complete, and the sorting/grouping constructs wait for the appropriate trigger to activate. Choosing these triggers was an important step in the design process, as we wanted the interface to update in response to a number of different scenarios, while also making sure that these updates were timed appropriately in a way that keeps the user experience smooth and free of unexpected jumps. These jumps can happen, for example, if a collection were to sort itself when a user is directly editing the data in that collection. The sorting and grouping constructs then process the data presented to them, by first figuring out the data type of each item in the collection, and then discerning how to apply the given criterion to these items. In the case of primitive items, the only piece of information we need from the criterion is the ordering, either increasing or decreasing. If we have a Mavo Node or JavaScript object, we also need the attributes of these items from which we can retrieve the value that we would like to base our sorting comparisons on. This is done by specifying one or more properties as the criteria, with the order direction (increasing or decreasing) specified for each individual property. In the case of multiple properties, a subsequent property becomes the new criterion when a former property resulted in a tie when comparing two items.

With grouping, some additional steps are taken to discern the individual groups, and to create group heading elements in the DOM. At its current state, groups are divided based on which items have equal values in a certain property. However, future iterations can easily support more complex grouping criteria as well by applying an operation on the resolved property. Thus, rather than checking for equal property values, we can instead simply check for equal transformed property values (e.g. transform all names into the first letter of the name, then group by people with the same first letter of their name). The template for the group heading elements can be specified by the user, otherwise we will automatically create a default heading element.

The remainder of the paper proceeds as follows: we first discuss various works related to the work presented in this thesis. Then we present the user experience of sorting

and grouping in Mavo, followed by the implementation details of these features. Next we describe future work that can be done to build off of these features, followed by a conclusion summarizing the extent and impact of our work.





# RELATED WORK

This chapter outlines the various works related to this thesis. This includes an overview of Mavo, the framework upon which the work in this thesis builds upon, as well as other frameworks that offer similar solutions to what we sought to accomplish.

## Mavo

The work in this thesis builds upon the existing work in Mavo, a framework that augments HTML syntax, giving authors the ability to implicitly define data schemas, and author a functional data-driven web application without writing a line of JavaScript. The existing work in Mavo supports a great deal of the preliminary work needed for sorting and grouping to function properly, including the creation of collections, the notion of implicit data schemas through properties, and the use of expressions to pass dynamic property values. Upon the existing Mavo codebase, we implemented sorting and grouping as a plugin, which takes advantage of the extensibility of Mavo to easily allow developers to add this additional functionality to Mavo.

## Similar Solutions

Some existing frameworks offer similar solutions to sorting and grouping, which we outline below. Table 1 outlines the features in each of the frameworks, and Figures 1 and 2 shows examples of accomplishing a given task in each of the frameworks.

### *AngularJS*

AngularJS<sup>1</sup> is a client side framework that has many similar approaches to Mavo. It operates similarly to Mavo by extending the HTML vocabulary, thus leaving most of the

---

<sup>1</sup> <https://angularjs.org/>

developer's interactions on the HTML side. However, AngularJS itself does require JavaScript to process and manipulate data, such as data related to a collection. Once this collection is set up, it can be referenced in an `ng-repeat` attribute, and sorted using an `orderBy` construct that can be referenced within this attribute.

The `orderBy` construct expects the properties of the items that we would like to have the items sorted by. When using multiple properties, AngularJS uses an array-like format, comma-separating the property strings and having the set of property names wrapped in square brackets. An example of this can be seen in Figure 1. Each property is prefixed with `+` or `-` to control the sorting direction, ascending or descending order respectively. The developer can also simply specify `+` or `-` without property names, which will use the collection items themselves in comparisons. This can be useful when sorting an array of primitives.

In addition, developers can choose to sort by a static or dynamic property, by providing either a string, or AngularJS expression respectively for the property (AngularJS uses its own version of expressions that operate similarly to Mavo, permitting developers to reference variables that can update and trigger changes in the UI).

Developers can also specify a defined comparator function, as a means to determine what makes an element "greater than" or "less than" another element. This comparator function is a JavaScript function defined by the developer within an Angular scope, such that its variable name can be referenced in an Angular expression in the HTML. AngularJS does not currently have an out-of-the-box grouping solution.

	Mavo	AngularJS	Exhibit
Sorting by multiple properties	✓	✓	✓
Control over sorting direction	✓	✓	✓
Dynamic sorting properties	✓	✓	✓ <sup>a</sup>
No JavaScript necessary	✓	✗ <sup>b</sup>	✓
Comparator function	✗	✓	✗
Grouping	✓	✗	✓

<sup>a</sup> Users cannot specify a dynamic property, but dynamic sorting is supported by default using an automatically inserted dropdown

<sup>b</sup> JavaScript is necessary to set up Angular and the data variables, but the sorting action itself does not require JavaScript

Table 1: Comparison of sorting and grouping features in various frameworks

```

<!-- Mavo -->
<div mv-multiple="people" mv-sort="+name -age"></div>

<!-- AngularJS -->
<div ng-repeat="person in people | orderBy:['+name', '-age']"></div>

<!-- Exhibit -->
<div ex:role="view" ex:orders=".name, .age" ex:directions="ascending, descending"></div>

```

Figure 1: Syntax for statically sorting a collection by multiple properties

```

<!-- Mavo -->
<div mv-multiple="people" mv-sort="+[propName]"></div>

<!-- AngularJS -->
<div ng-repeat="person in people | orderBy:propName:reverse"></div>

```

\* Note that Exhibit was excluded from this example, as its syntax does not support a variable property name

Figure 2: Syntax for dynamically sorting a collection by a single property

## ***Exhibit***

Exhibit<sup>2</sup> is a publishing framework that seeks to simplify the development of data-rich interactive web pages. Exhibit, unlike existing frameworks, expects data to be loaded at the start of the application. Thus, given that you have a preexisting data source, you can develop a data-rich interface in Exhibit with no extra JavaScript code. However, this also means that Exhibit solves a different use case from Mavo as a whole, since Mavo expects its users to be able to edit and save the data within the application, and see changes to the data immediately. With Exhibit, we only operate on one collection of data per application, so this collection is automatically inferred by Exhibit given that the data is passed in the correct format. Thus, the only parameters of sorting left to specify are the sorting criterion, which is specified through the use of an `ex:orders` attribute.

Similarly to `ng-repeat` in AngularJS, this attribute can accept one or more property names of the items, whose property values will be what the items are sorted by. Each of these properties must also be dot-prefixed, and comma separated (e.g. `ex:orders=".prop1, .prop2"`). Unlike AngularJS, the `ex:orders` attribute is only used to specify the sorting criterion upon page load, and updating the sorting order is an inherent feature in Exhibit that requires no additional code. Exhibit automatically inserts an element into the DOM that users can interact with to choose a new sorting criteria. Specifying the sorting direction per property is done through the use of the `ex:directions` attribute. For each listed property in `ex:orders`, the developer must specify one of either "ascending" or "descending". These values must also be comma separated.

By default, Exhibit does offer grouping functionality in sorted collections. Exhibit will automatically place header elements above subsections of the items with the same value specified by the sort criterion. Thus, grouping is built into the `ex:orders`

---

<sup>2</sup> <http://simile-widgets.org/exhibit/>

attribute by default, and can be turned off by setting the `ex:grouped` attribute to `false`.



# USER EXPERIENCE

The direct interaction that Mavo developers have with the features presented in this thesis is in their use of the various provided constructs. In this chapter, we will outline how sorting and grouping in Mavo is used by Mavo developers. We present two new constructs into the Mavo language to enable sorting: an `mv-sort` attribute, and a `sort` function. Similarly, we also present an `mv-groupby` attribute and a `groupBy` function for grouping functionality.

## Sorting and Grouping Using Attributes

The provided `mv-sort` and `mv-groupby` attributes provide a way for developers to specify sorting or grouping behavior on any element that defines or refers to a collection. This is the case with any element with an `mv-multiple` attribute, or any element with an `mv-value` attribute whose expression resolves to an array. Currently, both attributes operate on the same syntax, with only the output differing.

```
<ul>
  <li property="simpleGroup" mv-multiple mv-sort="+prop2 -prop1">
    <div>Prop 1: <span property="prop1"></span></div>
    <div>Prop 2: <span property="prop2"></span></div>
    <br>
  </li>
</ul>
```

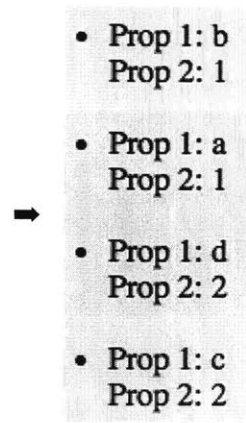
- 
- Prop 1: b  
Prop 2: 1
  - Prop 1: a  
Prop 2: 1
  - Prop 1: d  
Prop 2: 2
  - Prop 1: c  
Prop 2: 2

Figure 3: Using `mv-sort` with multiple properties



## ***Syntax***

Both the `mv-sort` and `mv-groupby` attributes expect space or comma separated values, where each value falls under one of the following formats:

- *Order character*
  - Simply a + or a – character, referring to increasing or decreasing order respectively.
  - e.g. +
- *Property name*
  - The name of an existing property of the items in the collection. The value of this property is what is compared when determining what is "greater than" or "less than" a value from a separate item in the collection.
  - e.g. `propName`
- *Order character followed by property name*
  - e.g. `-propName`

Simply using an order character will compare the items in the collection directly to each other. This is useful when sorting a collection of primitive values, such as numbers and strings. As such, simply using an order character as the criteria is typically done without any subsequent values.

When using a property name without prefixing with an order character, this will sort the collection by this property's value in a default order. If the user wants to specify increasing or decreasing order for a specific property, they need only prefix the property name with either a + or – respectively.

Sorting by multiple properties is done by simply space or comma separating these values. This specifies how to sub-sort items that resulted in a tie in a previous property. An example of this can be seen in Figure 3. Here we specified that first and foremost,

sort by `prop2` in increasing order. We notice in the output that two items had a value of 1 for `prop2`, and two items had a value of 2 for `prop2`. For these tied cases, the sorting function then referred to the next mentioned value in `mv-sort`, which was `-prop1`. Thus, in these subgroups, we sorted the previously tied items in decreasing order of their value for `prop1`.

### ***Dynamic Properties***

Often times, a Mavo developer may want the specified properties to change based on the state of the application. Using expressions in Mavo, this is a very easy task to accomplish. Rather than typing the name of a property of the collection items, developers can instead specify in an expression the value of an external property. This external property's value can be used to choose one of the property names in the collection items as the sorting/grouping criterion. Doing this requires the use of expressions, where the developer simply wraps the name of the external property in square braces, for example, `mv-sort="-[propName]"`, where `propName` is the name of an external property. Whatever the value of `propName` is in the application will populate `mv-sort` in replacement of the bracketed expression.

### ***Grouping***

Grouping is an extension of sorting, and thus, using `mv-groupby` in the same way as `mv-sort` will create a grouped collection in the same manner. The differentiating factor is the addition of heading elements in grouping. The group heading elements are displayed above a subset of the items in the sorted collection, and are used to divide the collection into subsets called groups, where each item in the group has a shared characteristic. In the current iteration, the criterion for grouping items together is that they have equal value in the property specified in the `mv-groupby` attribute. Using multiple properties in `mv-groupby` is currently incomplete, but is expected to result in a nested grouping structure, with latter properties appearing as groups within groups created by previous properties.

The group heading elements can be created either explicitly, or automatically inserted. In the explicit case, the user must specify an element with an `mv-group-heading` class directly above the collection template element. This element will be cloned, and the group values will be inserted inside each clone as the heading for each group. Otherwise, if the user does not specify a group heading element, Mavo will default to creating its own HTML section heading elements (one of `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`). First, Mavo decides on a template group heading element. By default, this will be an `<h1>` element, unless an HTML section heading element exists at some point in the DOM prior to the collection. In this case, if the previous section heading element is `<h(n)>`, then the group heading template element will be `<h(n+1)>`. If the previous section heading element was an `<h6>` element, the group heading template element will also be an `<h6>` element. In addition, any `aria-level` attributes that existed on a previous section heading element will have the incremented value as the `aria-level` of the group heading template element. After determining the template element, Mavo will clone this element for each group, and insert the group heading with its respective value above each group. Heading elements inserted automatically will also automatically have an `mv-group-heading` class added to it. An example of this can be seen in Figure 4. At the top, we see the setup of the application, which passes the `groupProp` property in an expression to the `mv-groupby` attribute. The `select` menu is used to select the value of `groupProp`. At the bottom, we observe the resulting HTML. For our example, we selected "company" as our grouping property. We notice that the resulting HTML inserted `<h2>` elements for each group heading. This is because there existed an `<h1>` element earlier in the document, which displayed "Students" at the top of the application. In addition, the `aria-level` of the `<h1>` element was incremented in the `<h2>` element from 1 to 2.

```

<h1 aria-level="1">Students</h1>
<p>Group by: </p>
<select property="groupProp">
  <option value="">None</option>
  <option value="degree">Degree</option>
  <option value="company">Company</option>
  <option value="year">Graduation Year</option>
</select>

<div mv-multiple="students" mv-groupby="+[groupProp]">
  <div><b property="name"></b></div>
  <div property="degree"></div>
  <div property="company"></div>
  <div property="year"></div>
  <br>
</div>

```

```

.mv-group-heading {
  border-top: 1px solid #efefef;
  border-bottom: 1px solid #ccc;
  background: #fafafa;
  border-left: 1px solid #eee;
  border-right: 1px solid #eee;
  padding: 4px;
}

```



# Students

Group by:



---

## Akamai

**Matt Levine**  
 S.M  
 Akamai  
 1997

---

## Google

**April Rasala**  
 S.M  
 Google  
 2001

**Jon Feldman**  
 Ph.D  
 Google  
 2003

---

## IBM

**Dennis Quan**  
 Ph.D  
 IBM  
 2003

```

<h1 aria-level="1">Students</h1>
<p>Group by: </p>
<select property="groupProp" mv-mode="edit" value aria-label="
<h2 aria-level="2" class="mv-group-heading mv-ui">Akamai</h2>
<div mv-multiple="students" mv-groupby="+company" typeof="Iten

```



Figure 4: Automatic group heading elements, with developer code at the top, and resulting HTML at the bottom

## Sorting and Grouping Using Functions

As we have seen, expressions in Mavo have proven to be vastly useful in their ability to pass variables within HTML attributes. An additional benefit of expressions is the ability to manipulate this data. Mavo provides an expression language, which expressions use to perform computations on data. This expression language is called MavoScript, and provides predefined special properties and functions which developers can use to manipulate data. The additions of the `sort` and `groupBy` functions provides extra functionality to this language, giving users more control over the collections used within expressions.

### ***Syntax***

Both functions expect an array to sort or group as the first argument. In MavoScript this can easily be done by passing the name of the collection. The remaining arguments are a variable number of property strings in the same format as provided to the `mv-sort` or `mv-groupby` attribute. This would be either an order character, a property name, or an order character prefixing a property name. However, the property arguments can also come in the form of arrays. This can happen when the developer references the item's property without wrapping it in string quotations, and an example can be seen in Figure 5. The reason for this is that by default, `mv-value` expects an expression without the explicit use of square brackets, and thus any properties referenced without string quotations are referenced as variables. When a property of an item of a collection is referenced in an expression, the expression resolves this value to an array of the property values of that property in the original collection. So in Figure 5, the `prop1` variable resolves to an array of `[2, -1, 3, 0, 5]`, as these are the corresponding values of `prop1` in unsorted order in the `simpleCollection` collection. This case is handled as a convenience to the developer, since switching between using string quotations and not using string quotations between the

```
<h4>List 1</h4>
<ul>
  <li property="simpleCollection" mv-multiple>
    <div>Prop 1: <span property="prop1"></span></div>
    <div>Prop 2: <span property="prop2"></span></div>
    <br>
  </li>
</ul>
<h4>List 2</h4>
<ul>
  <li mv-multiple mv-value="sort(simpleCollection, prop1)">
    <div>Prop 1: <span mv-value="prop1"></span></div>
    <div>Prop 2: <span mv-value="prop2"></span></div>
    <br>
  </li>
</ul>
```



- List 1**
- Prop 1: 2  
Prop 2: 0
  - Prop 1: -1  
Prop 2: 1
  - Prop 1: 3  
Prop 2: 2
  - Prop 1: 0  
Prop 2: 3
  - Prop 1: 5  
Prop 2: 4
- List 2**
- Prop 1: 5  
Prop 2: 4
  - Prop 1: 3  
Prop 2: 2
  - Prop 1: 2  
Prop 2: 0
  - Prop 1: 0  
Prop 2: 3
  - Prop 1: -1  
Prop 2: 1

Figure 5: Using sort function in mv-value

sorting/grouping attribute and the sorting/grouping function may have been a point of confusion.

## ***Use Cases***

As it stands, the most common use case of collections in expressions is within the `mv-value` attribute, and is typically used to create a synced copy of a preexisting collection. If the value of the `mv-value` expression resolves to an array and the element also has an `mv-multiple` attribute, then `mv-value` will operate similarly to creating an original collection, creating repeated DOM instances of the data schema of each element in the array. However, as this is a synced collection, the resulting data displayed to the user cannot be edited here, and can only be edited in the original collection. With the addition of the sorting and grouping functions, this synced collection can now be the sorted or grouped version of the original collection, should the user choose to display both. An example of this can be seen in Figure 5. Here list 1 is displaying the collection in its original unsorted order, whereas list 2 has the `sort` function operating on this collection with the value of `prop1` as its criterion for sorting.

The developer can also use `mv-sort` to accomplish the same result, by simply using `mv-value="deepGroup" mv-sort="prop1"`, and for Mavo's target audience, the usage of the function is currently more of a convenience for developers who would prefer to operate within MavoScript. However, for Mavo's advanced users, developers can perform array operations on the resulting sorted array from within the expression, as MavoScript also supports vanilla JavaScript. For example, a developer could take the sorted output and call `.slice(0, 3)` to display the top 3 elements in the sorted array.

## **Triggers**

One carefully considered point when implementing sorting and grouping was figuring out the right triggers under which the view should sort or group itself. For clarity, we will

describe here the triggers in the context of sorting. However, the same triggers that trigger sorting also trigger grouping, where each mentioned sorting construct would be associated with the corresponding grouping construct (`mv-sort` corresponds to `mv-groupby`, `sort` function corresponds to the `groupBy` function).

First, when all Mavo elements are rendered, any collection whose template element has an `mv-sort` attribute has a sort triggered on it given that the value of `mv-sort` is a valid sorting criterion. It was decided that should `mv-sort` have a null value or an empty string, the collection will be displayed in its original, unsorted order. The reasoning behind this decision was so that with the use of expressions, `mv-sort` can be used to show a collection sorted by various options of sorting criteria, or displayed in its original order should no criterion be selected. We found this functionality to be more useful than alternatively having an empty `mv-sort` attribute trigger some default sorting behavior.

Sorting is also triggered in response to updates to the state of the application. For example, a sort is triggered any time the sorting criterion is updated. This can happen when an expression is used in `mv-sort`, and the expression updates based on user interaction with another part of the application. In addition, sorting also occurs when any data related to the collection is edited. This includes direct edits to the collection items, or edits to other elements that indirectly change the collection data through expressions. However, an additional concern for this case is the jarring user experience that could occur should the shifting of the sorted elements occur as the user is editing a property value. To address this concern, the sort is only triggered when the property that is being changed is in read mode. This can happen in the latter aforementioned case using expressions and edits external to the collection. If the property being changed is in edit mode, the sort will only occur when the user exits edit mode. This allows the user to make any edits to the collection data without worrying about collection



items shifting locations, and will update its display once the user declares that they have completed their edits.

With regards to the `sort` function, it has essentially the same triggers for sorting as `mv-sort` from the user's point of view. However, note when using the `sort` function with `mv-value`, we are simply displaying the same data as another collection on the page. This means that the copy cannot be edited, and to change its values you must change the values from the collection it is based on. So any changes to a property value in the original collection, or changes to sorting criterion will cause the copied `mv-value` collection to evaluate itself for a sort.



# IMPLEMENTATION

The implementation of sorting and grouping in Mavo is done in the form of a Mavo plugin, `mavo-sort.js`, which users can easily include by including the associated JavaScript file, or loading the file through `mv-plugins="sort"`. This plugin adds a number of new functions to the Mavo codebase, and makes sure to call these functions at the appropriate times. Namely, we add `Mavo.Functions.sort`, `Mavo.Functions.groupBy`, `Mavo.Collection.prototype.sortDOM`, and `Mavo.Collection.prototype.groupDOM`, each of which we will explain in detail.

## Sorting as a Plugin

We chose to abstract the sorting functionality into a Mavo plugin. This keeps the code separate from the main Mavo codebase, and allows developers to include this code easily if they choose to do so. All of the sorting code is self-contained, and keeping this code in a plugin keeps Mavo modular, allowing developers only to load the code that they need for their application, which can keep page load times quick upon loading the application. After extended use, if we observe via feedback that sorting is an essential tool for most use cases, adding the abstracted plugin code to the main codebase is a simple task as well.

## MavoScript Functions

In the Mavo codebase, all MavoScript functions that are available to Mavo users are defined in `Mavo.Functions`. As such, our implementations of `Mavo.Functions.sort` and `Mavo.Functions.groupBy` directly add these functions for use in expressions through MavoScript.

## **Sorting**

Sorting in MavoScript is implemented through `Mavo.Functions.sort`. In the User Experience chapter, we talked about the various parameters that the developer can pass to this function, and how that function is used to sort in the DOM. Here we will talk about how these parameters are processed, and the various cases considered when formulating the final output.

### **Arguments and Data Processing**

The first argument to `Mavo.Functions.sort` is the array that we would like to sort. In MavoScript, passing the variable name of a collection will get its array representation. When implementing this function, we wanted to make sure it was able to handle item data of various data types. Namely, the cases we handle are `Mavo.Node` instances, primitives, and JavaScript objects. `Mavo.Node` instances will occur anytime a collection is referenced in a Mavo expression. From these instances, we can get their data in the format of a JavaScript object or a primitive by calling the `getData` method on the instance. If the collection items themselves have properties, `getData` will return JavaScript objects, otherwise, it will return either a number or a string primitive. Now we've equalized the cases between getting an array of `Mavo.Node` instances, and an array of primitives or JavaScript objects, and we can handle each case appropriately. Supporting arrays of primitives or JavaScript objects as input keeps MavoScript generalizable, such that advanced users can pass their own data should they choose to.

The remaining arguments represent a variable number of properties, and the order they would like to be sorted in. The order can only be specified if the arguments come in the form of strings, where the first character will either be a `+` or a `-`, denoting increasing and decreasing order respectively. Using the remainder of the string, if the items in the collection represent JavaScript objects (either directly or after `getData`), we can use this property name to search for the appropriate property value in the object. In the

case where the JavaScript object represents nested properties, the nested property can either be specified by its direct name, or through dot notation (e.g. if `prop2` is nested inside of `prop1` in a collection item, a user can either simply specify `prop2`, or `prop1.prop2`).

The property arguments can also come in the form of arrays, representing the respective values of the property in the unsorted collection. In these cases, in order to retrieve the appropriate property value, we need the index of the item in the unsorted array. We do this by creating a copy of our array using JavaScript's `map`<sup>3</sup> function, that maps each item to an `[item, index]` tuple, that gives us access to both pieces of information for a given item. This tuple also comes in handy for stable sorting, where we compare indices in the case that all property comparisons resulted in a tie. The use of these arrays as property arguments does not have any way of specifying sorting direction, and in these cases, we default to sorting the items in the default sorting direction, which is currently in decreasing order.

Now that we have defined a way to retrieve the values used for comparison, we segue into where these comparisons happen. We use JavaScript's native `sort`<sup>4</sup> function, where all of the processing mentioned above happens in the comparison function that we've defined. The comparison function gets the aforementioned `[item, index]` tuples as arguments, and use the property arguments to retrieve the value of comparison from `item`. In the case that all properties result in ties, we resort to comparing the indices, resulting in a stable sort, since stable sort is not natively supported by JavaScript's `sort` function.

### **User Defined Comparison Function**

In its current state, sorting in Mavo does not support the use of a user-defined comparison function. The main reason this was not prioritized in the current iteration of

---

<sup>3</sup> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

<sup>4</sup> [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array/sort](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort)

Mavo sorting is that defining a complex comparison function lies outside the expectations of Mavo's target audience. Mavo seeks to be a viable solution for those with little to no programming experience, and the addition of such a feature would likely not be used by this audience. In addition, as Mavo is a language used entirely within HTML, figuring out the right method for developers to define such a function is a challenge within itself. Though Mavo's target audience may be unfamiliar with programming, there are more advanced users that do use Mavo, and supporting the use of user-defined comparison functions could be a useful addition that can be explored in future iterations of sort.

### **Sorting Order Specification**

One point of contention was deciding the best way for a user to specify the order that a property should be sorted in. While we ended up choosing the `+/-` prefix format, we also thought about the idea of using comma separated word pairs. The first word in the pair would be the property name, and the second word would be either `asc` specifying ascending order, or `desc` specifying descending order (e.g. `mv-sort="prop1 asc, prop2 desc.."`). The main advantage to this format was increased readability at the expense of more characters. In addition, users must comma separate the word pairs, whereas they can choose to comma or space separate with the existing implementation. We chose the `+/-` prefix format as we believed it to be concise, and helped stick with the HTML attribute standard of space separating separate values within an attribute, while also providing the freedom to comma separate if a developer is more comfortable with that format. In addition, we found it to be only slightly less readable than the alternative such that the tradeoff was worth it.

### **Null Prefixed Sorting**

Occasionally, users may save their data without inputting values for the relevant property fields in a collection item. This results in null values for that property, and we needed to decide how to deal with these cases when sorting by a property that has a null value for some items. For these items with null properties, we decided to always

place these items at the end of the collection. Our reasoning behind this decision is that users will likely want to see relevant data first, and such relevant data is more likely to be non-null values. However, there may be instances where users want to specify that they want to see the null items first, and while this is not supported in the current iteration of `sort`, it is something that can be explored in future iterations.

## ***Grouping***

As grouping is an extension of sorting, the parameters that `Mavo.Functions.groupBy` accepts are exactly the same as `Mavo.Functions.sort`. In fact, the first thing that the `groupBy` function does is call `Mavo.Functions.sort` on the input array with the same parameters to get a sorted copy that the function uses to create grouped structures.

These group structures come in the form of a JavaScript object, and use three object properties to convey the information related to a group: `id`, `property`, and `items`. The `id` property specifies the value of the group headings, `property` specifies which property name the grouping function is grouping on, and `items` is an array of the items from the original array that this group contains, in sorted order. In order to determine the value of `id`, `groupBy` parses the items in the same way that the `sort` function does to retrieve the elements being compared. The `items` property may also contain nested group structures in the case of grouping by multiple properties. Thus, the final output of the `groupBy` function is an array of group structure objects, each of which may recursively contain an array of group structure objects in `items`, or may simply contain the grouped items in sorted order.

Something to note is that in order to use the `groupBy` function in `mv-value`, developers must be sure that the structure of the property elements matches that of the group structure objects output by `groupBy`. As such, if the developer wants `groupBy` to work in the same way as `mv-groupby`, they need to be sure that they include an `id`

property element, as well as an `items` property element within the `mv-value` element. The value of the group header will be displayed in the `id` property element, whereas the respective items will be displayed in the `items` property element.

## DOM Manipulation Functions

The `Mavo.Collection.prototype.sortDOM`, and `Mavo.Collection.prototype.groupDOM` functions are used by the plugin internally to interact with the DOM when the `mv-sort` or `mv-groupby` attributes are used. When Mavo finds an element that should be interpreted and saved as a collection (the element must have an `mv-multiple` attribute and a property name), Mavo first uses the data associated with the property to create a `Mavo.Collection` instance. Mavo looks at the associated data and the template element to which `mv-multiple` is attached, and creates children elements that are clones of the template element, with each child populated with item data in the appropriate locations. Mavo appends these clones to the DOM in place of the template element, and uses these elements to create `Mavo.Node` instances which are stored in the `children` property of the `Mavo.Collection` instance.

### *Setup*

Since `sortDOM` and `groupDOM` are prototype methods that can be called from a collection instance, any collection now has the ability to change its DOM representation. In addition, the collection instance gives `sortDOM` and `groupDOM` access to the elements that need reordering, where they are currently placed in the DOM, and the `Mavo.Node` instances associated with the data. Since these DOM methods already have access to this data as a result of being called from a collection instance, the remaining pieces of information needed to sort or group in the DOM are the properties that the collection should be sorted or grouped by, which `sortDOM` and `groupDOM` both take as arguments in the form of a list of values. Each value is formatted in the same way that the `sort` or `groupBy` functions expect them, either as a string representing



the property and its order direction, or an array representing the property's unsorted values.

### ***Sorting the DOM***

Sorting the DOM is then simply done by first retrieving the array data associated with the collection, which we can simply use `this.children` for (where `this` refers to the `Mavo.Collection` instance). Using this array and the properties passed to `sortDOM`, we can call `Mavo.Functions.sort` to get an array with the Mavo nodes in sorted order. Finally, we create a `DocumentFragment`<sup>5</sup>, append the elements associated with the ordered Mavo Nodes to this fragment, and then append this fragment to the DOM in place of the template element. Appending elements to the `DocumentFragment` rather than the DOM minimizes the amount of restructuring incurred on the DOM, which is expected to be a much larger and more complex `Document`<sup>6</sup>, resulting in better expected performance of a sorting or grouping action.

### ***Grouping the DOM***

In order to group the DOM, `groupDOM` first gets the data it needs for grouping, by calling `Mavo.Functions.groupBy` with the appropriate parameters. Next, `groupDOM` determines the heading template element, which will later be cloned and populated with heading data. If the collection has a heading template element stored in `this.headingTemplate`, this element will be used. This element is stored after the collection determines its heading template element for the first time, since the template is removed from the DOM after grouping. This allows users to group and ungroup a collection, while still giving the collection access to the template. If a stored template heading element doesn't exist, `groupDOM` proceeds to determine it in the same way described in the User Experience chapter, by using the previous `mv-group-heading` element if it exists, otherwise creating a section heading element based on the closest section heading element that appears above the collection template element when

---

<sup>5</sup> <https://developer.mozilla.org/en-US/docs/Web/API/DocumentFragment>

<sup>6</sup> <https://developer.mozilla.org/en-US/docs/Web/API/Document>

viewing the HTML source code. We determine the existence of such an HTML section heading element by using a simple selector to find all section heading elements that exist in the Mavo application, and using `Node.compareDocumentPosition`<sup>7</sup> to determine the closest one existing prior to the collection template element. After determining the appropriate heading template, this element gets saved in `this.headingTemplate` for future use.

Now we move to inserting the collection elements and the heading elements in the correct spots in the DOM, and with the correct data. This is done by traversing the output of the `groupBy` function. This traversal starts by looking at the first group structure object. As this structure represents a group, we clone the heading template element, populate it with the `id` value of the group structure object, and append this to a `DocumentFragment`. Then we recursively look at the `items` property of this element, appending heading elements to the DOM for every nested group structure we find, until we come across a collection item. Upon reaching an array of collection items in an `items` property, we append the elements representing these items in their sorted order to the fragment, placing them beneath the appropriate heading element. The control flow of this recursive process is outlined in Figure 6. After the fragment is populated with the appropriate headings and elements, it is simply appended to the DOM in place of the collection template element.

---

<sup>7</sup> <https://developer.mozilla.org/en-US/docs/Web/API/Node/compareDocumentPosition>

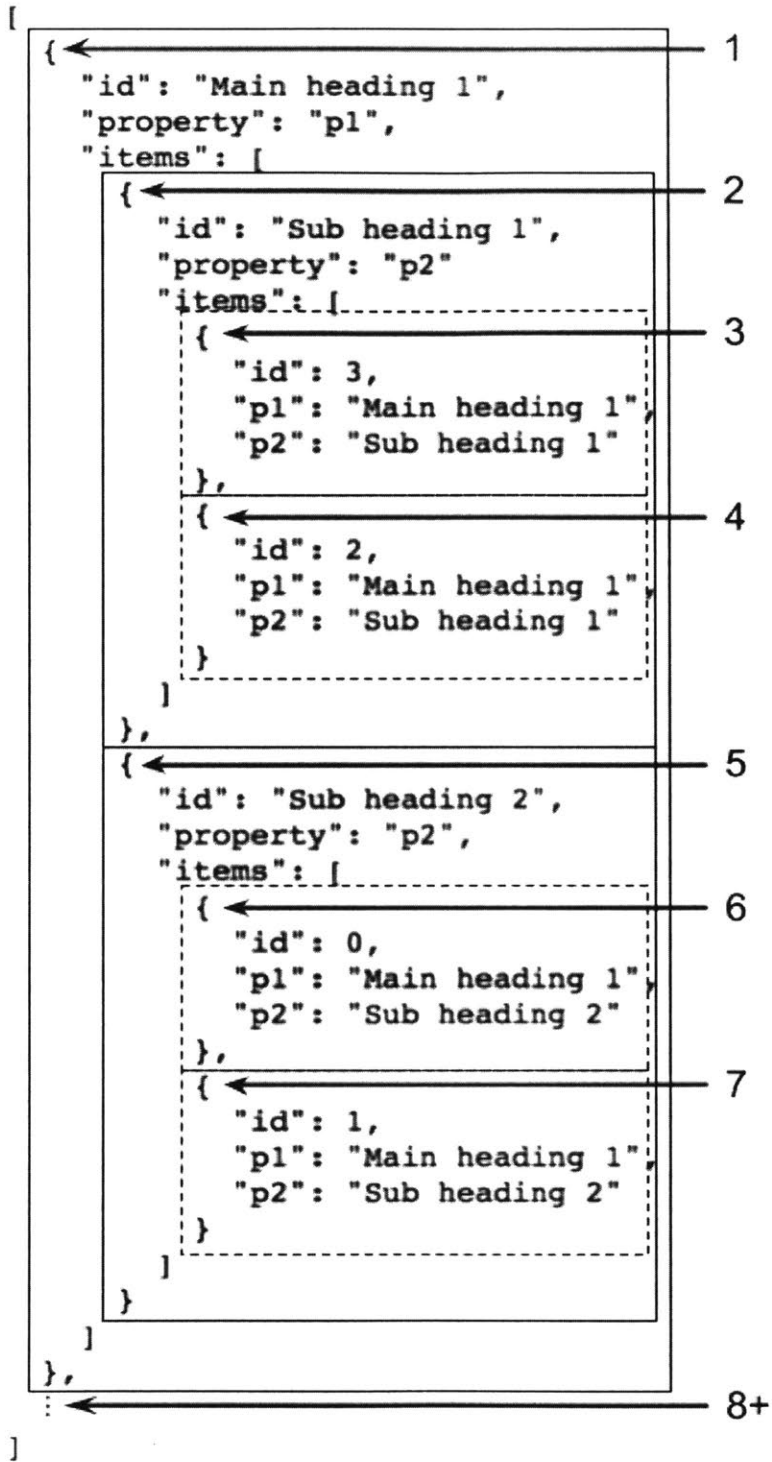


Figure 6: Control flow while traversing output of `groupBy` function in `groupDOM`. Solid boxes outline group structure objects, dashed boxes outline collection item data. Numbers on the side displays the order the elements are traversed in.

## ***Cached Criteria***

Both DOM manipulation functions also cache the criterion by which they were last sorted. This criterion is stored in the collection instance as a property in the form of a formatted string, that is unique per criterion. In certain instances of sorting and grouping, before the associated DOM manipulation function executes, we compare the given criterion to what is stored. If they do not match, the sorting or grouping action continues, otherwise, this action is skipped. This can help eliminate unnecessary sorting or grouping, which can occur in some cases of triggers described below.

## **Triggers Implementation**

Earlier we described the different scenarios that causes the view to sort or group itself from the user's perspective. Here we will delve into how we account for all of these various cases in the implementation of our plugin. We will again explain our implementation in the context of sorting, under the assumption that the same logic applies for grouping. We recall that we account for three general cases of sorting: when Mavo loads all relevant elements, when the sorting criterion changes, and when the collection data changes.

Many of these cases are handled through the use of Mavo hooks. Mavo hooking is an extensibility practice used by Mavo to give Mavo plugin developers the ability to execute bits of plugin code at various portions of the Mavo codebase, without them directly editing the Mavo codebase. In the Mavo codebase, at various key instances of code execution, a `Mavo.hooks.run(name, env)` method is called. Plugin developers can then run their own code at this point of execution by defining a callback function in `Mavo.hooks.add(name, callback)`.

To trigger sorting when Mavo loads all relevant elements, we make use of the "node-render-end" hook, which is executed when a `Mavo.Node` instance has its

element rendered in the DOM. At this point, we simply check if the associated element has an `mv-sort` attribute, determine the associated `Mavo.Collection` instance, and pass the associated parameters to `sortDOM`. We must note that this is triggered for each element in a collection, and as such we take advantage of caching the sort criterion to make sure that the collection is only sorted once per criterion.

Next we consider the case of sorting when the sorting criterion changes. To handle this, we make use of hooks, and `MutationObserver`<sup>8</sup> instances. First, we use the `"init-end"` hook as a spot to set up our `MutationObserver` instances. This hook is run when Mavo has created all internal Mavo objects, but has not rendered all the Mavo-generated elements. When this hook is run we create a special `MutationObserver` for every element with an `mv-sort` attribute, which will listen for any changes to this attribute, which happens when an expression used in this attribute is updated. When these changes occur, we can call `sortDOM` with the appropriate parameters. We opted to use the `"init-end"` hook because at this point, the elements of a collection aren't cloned yet, with only the template element existing in the DOM. Thus, we create one `MutationObserver` instance per collection.

Finally, we describe how we trigger a sort upon changes to collection data. To handle this case, we use the `"render-end"` hook, which is run when all elements associated with `Mavo.Node` instances are rendered in the DOM. At this hook's callback, we listen for `"mv-change"` and `"mv-done"` events on these elements. These are events created by Mavo that are broadcasted in certain scenarios. `"mv-change"` is triggered upon any data changes to the data of a `Mavo.Node` instance, where the event is fired from the associated element. Upon this event, we check if the changed node is in read mode, meaning it must have been changed from an external interaction, likely through the use of an expression. If so, we find the collection containing this `Node`, make sure that the sort criterion of the collection includes the property that this `Node` represents,

---

<sup>8</sup> <https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>

and if so, we trigger a sort on this collection. We don't trigger sorting in write mode, as this would result in the jumping behavior of elements moving in the DOM as a user is attempting to edit them. Instead, to handle direct edits to a collection, we use "mv-done". This event is triggered when a user exits edit mode on a Mavo application, and is triggered for every `Mavo.Node` in the application. As such, when triggered, we trigger a sort on every collection in the Mavo application.



# FUTURE WORK

## Nested Grouping

While sorting by multiple properties is currently supported, grouping by multiple properties is currently incomplete. When grouping by multiple properties, the heading elements will appear above each nested group, but the concept of a nested structure is not currently apparent. The group heading element is the same for every level of the group, even when using the automatically inserted `<h1>-<h6>` HTML section heading elements. In the future, we instead would like to increment `n` for each `<h (n) >` element at a deeper level than the previous, such that the section heading elements themselves could convey this nested structure. In addition, the use of indentation is a commonly used mechanism to convey nested information, and would be a useful visual aid to Mavo grouping.

## Automatic Group Headings for Specific Elements

In the case where the user does not specify a heading element with `mv-group-heading`, while using HTML section heading elements is a reasonable general case heading element, there are cases where there are better choices for heading elements depending on what element the underlying collection is operating on. For example, in the case where the collection is operating on an `option` element within a select menu, the best candidate for a heading element here would be an `optgroup` element. While the user has the power to specify this themselves, it would be desirable to have a default action that makes sense in as many conceivable cases as possible.



## Support for More Powerful Grouping Criteria

Currently with grouping, we group by the same criterion with which we sort a collection. Given that grouping is a feature built on top of sorting, this works and makes sense. However, there are cases where users will want more control over what defines a group heading. For example, consider a contact list that simply displays names. Given the current implementation of grouping, only people with the same name will be grouped together. Assuming that most people have different names, this grouping criterion would not be effective in helping to organize the data. The grouping functionality provided by Mavo should give users the ability to specify a more meaningful grouping criterion based on the data that they expect in their application. One way this could be done is through the use of an expression that maps a property value to a new heading value. With this heading value, we can now check for equivalence as we currently do with the base property value. In the context of contact list example, consider an expression that maps a name to the first letter of that name, thus producing a contact list grouped by letter. Something left to consider is how such an expression can work easily with the existing `mv-groupby` and `group` function constructs in a way that is simple to use and easily understood by a web developer.

## Persisting the Sorting Order

As it stands, sorting is a view action rather than a data action. This means that the sorted order is not persisted along with the rest of the collection data when the user saves their Mavo application. Even so, most use cases currently give the appearance that the order is persisted. If a static property is used as the sorting criteria, that static property runs the sort on the view every time the page is loaded. If a dynamic property is used as the sorting criteria (e.g. through the use of expressions), then if the expression is based on the value of an external property, that property can be saved. Thus the sorting criteria is persisted in that manner, resulting in the same sort upon

page load. However, it may be worth exploring whether there are use cases not covered here that would be solved by saving the sort order.



# CONCLUSION

Mavo currently gives developers a great deal of power over how their web applications will display data, with very little effort on their part. When implementing sorting and grouping functionality into Mavo, keeping this preface in mind was an important aspect that guided the efforts in maintaining a quality developer experience with the addition of new functionality. Sorting and grouping gives developers even more control over their web applications, with very little code required to maintain a dynamically updating data-driven web application that Mavo seeks to provide its users. With even more control over the display of their data, we expect more users to find Mavo to be a top choice of framework when developing data-driven web applications.

# BIBLIOGRAPHY

[1] Verou, Lea, Amy X. Zhang, and David R. Karger. "Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML." *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 2016.