Creating Interactive Data-Driven Web Applications by Authoring HTML

by

Lea Verou

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
January 31, 2017

Signature redacted

Certified by ....
David R. Karger
Professor
Thesis Supervisor

Signature redacted

Accepted by ....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Creating Interactive Data-Driven Web Applications by Authoring HTML

by

Lea Verou

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Many people can author static web pages with HTML and CSS but find it hard or impossible to program persistent, interactive web applications. We show that for a broad class of CRUD (Create, Read, Update, Delete) applications, this gap can be bridged. Mavo extends the declarative syntax of HTML to describe Web applications that manage, store and transform data. Using Mavo, authors with basic HTML knowledge define complex data schemas *implicitly* as they design their HTML layout. They need only add a few attributes and expressions to their HTML elements to transform their static design into a persistent, data-driven web application whose data can be edited by direct manipulation of the content in the browser. We evaluated Mavo with 20 users who marked up static designs—some provided by us, some their own creation—to transform them into fully functional web applications. Even users with no programming experience were able to quickly craft Mavo applications.

Thesis Supervisor: David R. Karger
Title: Professor

Acknowledgments

This thesis would not have been possible without the help and support of a number of great people, for which I'm deeply grateful.

To my late mother, **Maria Verou** for being a source of inspiration and encouragement for the 27 years our lives overlapped. From a young age she inspired me to be creative, to never stop learning, to be ambitious. This thesis, and Mavo itself, are dedicated to her.

To my uncle **Stratis Veros** and his lovely wife **Maria Brere**. You are like second parents to me. Thank you for enabling me to study here by spending so much of your time taking care of my estate back in Greece and all the bureaucracy involved.

To my father, **Miltiades Komvoutis** for teaching me so many things early on in my life.

To my advisor, **David Karger**. It has been so much fun working with you for the last 2 years. I have learned so much and achieved things I didn't think I could. I cannot imagine a better mentor. I'm looking forward to the rest of my time here.

To my co-author and fellow Haystacker **Amy Zhang**. My first paper and my first user study would have been immensely more difficult without your help.

To my friend and ex-Haystacker **Eirik Bakke** for his tremendous help on everything related to MIT and academia since my first day here. I hope someday day I can help another newbie as much as you have helped me.

To all my user study participants for their help.

To the **Paris Kanellakis** family and the **Wistron Corporation** for funding this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is a sizeable community of authors creating static web pages with basic HTML and CSS. While it is difficult to pinpoint the size of this community, it is likely large and growing, evidenced by the large number and popularity of WYSIWYG and text-based editors and tools for generating static websites, as well as the number of hosting providers that will only host static webpages (for example, Github Pages[1]). The ACM cites knowledge of HTML and CSS to be at the K-12 level of computer literacy [29].

Far more powerful than static pages are web *applications* that react dynamically to user actions and interface with back-end data and computation. Even a basic application like a to-do list needs to store and recall data from a local or remote source, provide a dynamic interface that supports creation, deletion, and editing of items, and have presentation varying based on what the user checks off. Creating such applications currently requires knowledge of JavaScript and/or other programming languages to provide interaction and to interface with a data management system, as well as understanding of some form of data representation, such as JSON or a relational database.

There are many frameworks and libraries aiming to simplify creation of such Web applications. However, all target programmers and still require writing a considerable amount of code. As shown in Table 1.1, implementing even a simple to-do application similar to the one in Figure 1-1 requires writing hundreds of lines of JavaScript code, even with the state of the art JavaScript frameworks. Other JavaScript frameworks are in the same ballpark.

Many people who are comfortable with HTML and CSS do not possess additional program-

[1]https://pages.github.com/

Framework	SLOC
AngularJS	294
Polymer	246
Backbone.js	297
React	421
Vue.js	137

Table 1.1: Number of lines of JavaScript code required to implement a simple to-do list with four of the most popular JavaScript frameworks. Comments, framework code, and polyfills are excluded. Statistics from *todomvc.com*, a popular framework comparison site.

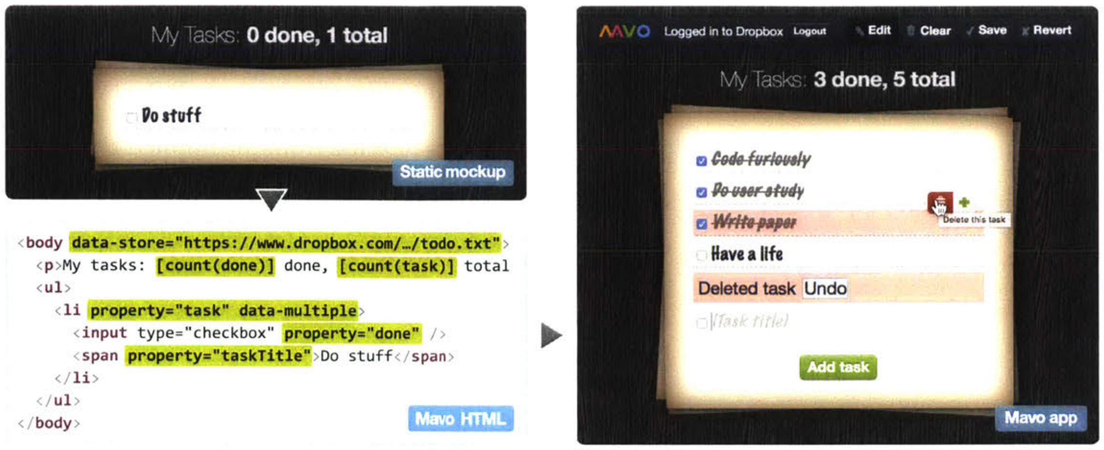
ming skills[2] and have little experience articulating data schemas [28]. For these novice web authors, using a CMS (Content Management System) is often seen as their only solution. However, research indicates that there are high levels of dissatisfaction with CMSs [15]. One reason is that CMSs impose narrow constraints on authors in terms of possible presentation–far narrower than when editing a standalone HTML and CSS document. When an author wishes to go beyond these constraints, they are forced to become a programmer learning and modifying server-side CMS code.

The problem worsens when authors wish to present structured data [6], which CMSs enable via plugins. The interfaces for these plugins are often specialized for specific types of data (e.g. publications) and/or do not allow authors to edit data in place on the page; instead they must fill out forms[3]. This loses the direct manipulation benefits that are a feature of WYSIWYG editors for unstructured content.

Finally, CMSs provide a heavyweight solution when many authors only need to present and edit a small amount of data. They require installation, configuration, a database, and a hosting provider able to run server-side code. For example, out of the over 7,000 CMS templates currently provided in ThemeForest.net, a repository of web templates, 39% are for portfolio sites, while another 31% are for small business sites [3].

[2]We carried out a snowball sample of web designers using a Twitter account followed by 70,000 Web designers and developers. Of 3,578 respondents, 49% reported little or no programming ability.

[3]E.g. https://wordpress.org/plugins/wp-seo-structured-data-schema/



```
<body data-store="https://www.dropbox.com/…/todo.txt">
  <p>My tasks: [count(done)] done, [count(task)] total
  <ul>
    <li property="task" data-multiple>
      <input type="checkbox" property="done" />
      <span property="taskTitle">Do stuff</span>
    </li>
  </ul>
</body>
```

Figure 1-1: A fully-functional To-Do app made with Mavo, shown with its accompanying code and the starting HTML mockup.

1.1 Our Contribution

This paper presents and evaluates a new language called Mavo[4] that augments HTML syntax to empower HTML authors to implicitly define data schemas and add persistence and interactivity. Simply by adding a few HTML attributes, an author can transform any static HTML document into a dynamic data management application. Data becomes editable directly in the page, offering the ability to create, update, and delete data items via a WYSIWYG GUI. Mavo authors never have to articulate a schema separately from their interface or write data binding code. Instead, authors add attributes to describe which HTML elements should be editable and how, unwittingly describing their schema by example in the process. With a few attributes, authors quickly imply complex schemas that would have required multiple tables and foreign keys in a relational database, without having to think beyond the interface they are creating. As an added benefit, Mavo's HTML attributes are part of the HTML RDFa standard [31] and thus contribute to machine-readable data on the Web.

Mavo is inspired by the principle of *direct manipulation* [30] for the creation of the data model underlying an application. Instead of crafting a data model and then deciding how to template and edit it, a Mavo author's manipulation of the visual layout of an application automatically *implies* the data model that drives that application. In addition, Mavo does not require the author to create

[4]Open source implementation & demos available at **http://mavo.io**

a separate data editing interface. Users simply toggle an edit mode in their browser by clicking an edit button that Mavo inserts on their webpage. Mavo then adds affordances to WYSIWYG-edit whatever data is in view, with appropriate editing widgets inferred from the implied types of the elements marked as data. Mavo can persist data locally or outsource storage to any supported cloud service, such as Dropbox or Github. Switching between storage backends is a matter of changing the value of one attribute.

In addition to CRUD functionality, Mavo provides a simple spreadsheet-like expression syntax to place reactive calculations, aggregates, and conditionals on any part of the interface, enabling novices to create the rich reactive interfaces that are expected from today's web applications.

In contrast to the hundreds of lines of code demanded by the popular frameworks, Figure 1-1 shows how an HTML mockup can be transformed into a fully functioning, high fidelity to-do application by adding only 5 lines of Mavo HTML.

We conducted a user study with 20 novice web developers in order to test whether they could use Mavo to turn a static HTML mockup of an application into a fully functional one, both with HTML we provided and with HTML of their own creation. We found that the majority of users were easily able to mark up the editable portions of their mockups to create applications with complex hierarchical schemas.

Our approach constitutes a novel way for end users to transform static webpages to dynamic, data-backed web applications without programming or explicitly defining a separate data schema. From one perspective, this makes Mavo the first *client-side CMS*, where all functionality is configurable from within the HTML page. But it offers more.

In line with the vision of HTML as a *declarative* language for describing content so it can be *presented* effectively [25], Mavo extends HTML with a declarative specification of how the data underlying a presentation is structured and can be *edited*. Fundamentally a language extension rather than a system, Mavo is completely portable, with no dependence on any particular web infrastructure, and can thus integrate with any web system. Similarly, existing WYSIWYG HTML editors can be used to author Mavo applications. We offer Mavo as an argument for the benefits of a future *HTML language standard* that makes structured data on every page editable, persistent and transformable via standard HTML, without dependencies.

The majority of this work has been previously published in a conference publication with co-authors Amy Zhang and David Karger [32].

Chapter 2

Background

This thesis builds upon work from several communities, academic and industrial. This chapter provides an overview of this prior work.

2.1 Main influences

Mavo combines ideas from three prior systems that addressed the downsides of CMSs: Dido [22], Quilt [8], and Gneiss [13, 14].

2.1.1 Dido

Dido [22] built on Exhibit [19], extended HTML with language elements that visualized and stored editable data directly in the browser. This approach allowed a web designer to incorporate Dido into any web design and made Dido independent of any back-end system.

2.1.2 Quilt

Quilt [8] extended HTML with a language for binding an arbitrary web page to a Google spreadsheet "back-end", enabling web authors to gain access to lightweight reactive computation and data management without programming.

2.1.3 Gneiss

Gneiss [13, 14] was a web application within which authors could manage and compute over hierarchical data using an extended spreadsheet metaphor, then use a graphical front end to interact with that data. The data could also originate from Web services, facilitating the creation of mashups.

2.1.4 Relation to Mavo

These three systems introduced powerful ideas: extending HTML to mark editable data in arbitrary web pages, spreadsheet-like light computation, a hierarchical data model, and independence from back-end functionality. But none of these systems provides all of these capabilities simultaneously. Dido had no computational capabilities, could not manage hierarchical data, and was never evaluated. Quilt was dependent on a Google spreadsheet back-end, which left it unable to manage hierarchical data. Gneiss was a monolithic web application that only allowed the user to construct web pages from a specific palette. It did not offer any way (much less a language) to associate an arbitrarily designed web page with the hierarchical data Gneiss was managing, which meant that a web author faced constraints on their design creativity. Gneiss and Quilt both required users to design their data separately from their web pages.

Mavo is a *language* that solves the challenge of combining the distinct positive elements of this prior work, which are in tension with one another. It defines a simple extension to HTML that enables an author to add data management and computation to *any* web page. At the same time, it provides a lightweight, spreadsheet-like expression language that is expressed and evaluated *in the browser*, making Mavo independent of any particular back-end. The editing and expression language operates on *hierarchical data*, avoiding this limitation of traditional spreadsheet computation. While hierarchical data is possible to manage in spreadsheets, it is typically performed in the same way as with relational data models: Multiple worksheets with foreign key references via the `VLOOKUP()` function. Fragmenting hierarchical data to multiple tables is known to be hard for novices [21].

The combination of these ideas yields a novel system that is particularly well-suited to authoring interactive web applications. In Mavo (like Dido), the author focuses entirely on the design of the web page, then annotates that page with markup describing data and computation. The web page *implies* the data model, freeing the author of the need to abstractly model the data, manage a spreadsheet, or describe bindings between the two. At the same time, our expression language provides lightweight computation (Quilt and Gneiss), even on hierarchical data (Gneiss) without relying on any external services (Dido). Because they are part of the document (Dido), Mavo expressions can refer directly to data elements elsewhere in the document, instead of requiring a syntactic detour through references to cells in the associated spreadsheet. Finally, because it is an HTML language extension (Dido and Quilt), Mavo can be applied to *any* web page and authored with any HTML editor, freeing an author from design constraints.

In sum, we believe that the combination of capabilities of Mavo align well with the needs and the preferred workflow of current web authors. In particular, the independence of the Mavo authoring *language* from any back-end system (or even from any particular front-end interpreter) means that Mavo prototypes a future for HTML and the web browser itself, where data interaction becomes as much a basic part of web authoring as paragraphs and colors.

2.2 Content Management Systems

There are many systems that assist novice web developers with building dynamic and data-backed web applications. The drawback to many of these systems, however, is that they often require using their own heavyweight authoring and hosting environments, and they provide pre-made plugins or templates that users can not customize without programming. Examples of such systems include CMSs such as Wordpress, Drupal, or Joomla.

The growing community around static site generators, such as Jekyll [1] is indicative of the dissatisfaction with rigid, heavyweight CMSs [6]. However, these require significant technical expertise to configure, and offer no graphical interfaces for editing data. Furthermore, many uses of CMSs are merely to enable non-technical users to edit website content, a use case that static site generators do not accommodate.

Other types of systems for end user web development include WYSIWYG editors such as CKEditor. These tools are much more lightweight, but also have limited functionality for presenting structured data. Newer editors such as Wix.com allow for structured data, but again only via pre-made snap-in widgets similar to CMSs.

In the previous section, we described three systems—Dido [22], Quilt [8], and Gneiss [13]—from which we draw key insights. However, this work solves challenges in combining those insights into a single system, incorporates additional ideas, and contributes useful evaluation of the resulting system. Most importantly, Mavo demonstrates that the often-hierarchical data model of an application can be incorporated directly into the visual design on which a web author is focused, making the data modeling task an automatic side effect of the creation of the web design. Supporting hierarchical schemas is critical because they occur naturally in many data-driven apps on the web (53% according to [6]). Our evaluation studies users working with such hierarchical schemas.

2.3 Visual application builders

Visual application builders like app2you [23] and AppForge [37] allow authors to specify the design of pages by placing drag-and-drop elements into a WYSIWIG-like environment. However, this approach limits authors to only the building blocks provided by the tools and cannot be used to transform arbitrary HTML. A followup system, FORWARD [17], is more powerful but requires writing SQL queries within HTML.

Early visual programming languages such as Forms/3 [11] and NoPumpG [36] extend the spreadsheet paradigm to graphical interfaces and interactive graphics. However, they do not afford any customization in terms of input UI, and have no concept of a data store. Also, they only target single-user local web applications and do not address the unique challenges that Web applications raise.

There has been some work to better present hierarchical data within spreadsheets [14, 4]. However, the flexibility of these presentations is limited and only slightly deviates from a tabular display.

Additionally, while programmers generally prefer to keep their data schema logic separate from the presentation definition as evidenced by the number and popularity of MVC frameworks, end users may not have the same preferences, and may instead be frustrated by the need to think about data in two separate places. Indeed, with a certain category of applications, including most CRUD applications, how the data is laid out on the page can easily translate to how the data should be organized. For end users who are seeking to build these sorts of apps, it may be easier to define a proper schema in tandem with defining the layout.

2.4 MVC Frameworks

In the last decade, many client-side frameworks have been created promising to simplify the creation of Web applications. Some of them (mainly AngularJS [20] and Vue.js [38]) even offload functionality to HTML, similarly to Mavo.

However, these frameworks use HTML markup as a convenient shortcut for View specification, not as a way to describe the entire application with no additional programming. To create an application utilizing these frameworks, several software engineering concepts and design patterns need to be understood (such as MVC) and, as shown in Table 1.1, they still require a nontrivial amount of JavaScript to be written in addition to the HTML annotations.

2.5 Web Components

Web Components is a recent Web technology consisting of a set of four W3C and WHATWG specifications centered around improving the extensibility of HTML:

- Custom HTML Elements [16]
- HTML Imports [18]
- HTML Templates [35]
- Shadow DOM [33]

On the surface, Web Components and Mavo share many goals: both aim to empower HTML authors to create Web applications. However, in Web Components, HTML is only used to include, call, and configure existing widgets. Developing said widgets still requires programming. While this is a step in the right direction, it severely limits what authors can create to components third-party developers envisioned as useful, not components tailored to their individual use cases.

2.6 The Semantic Web and Web Data Extraction

There has been a great deal of work on both encouraging and extracting structured data on the web [12]. However, automatic scraping techniques often have errors because they must infer structure from unstructured or poorly structured text and HTML markup. Several efforts have been made to define syntaxes and schemas, such as RDFa [31] and Microdata [34], for publishing structured data in web pages to contribute to the *Semantic Web* and *Linked Open Data* [9]. However, novice users have had little incentive to adopt these standards—sharing data rarely provides direct benefit to them—and find them difficult to learn, potentially contributing to their limited adoption on the web. Mavo contributes to this line of work by using a standards-compliant syntax that is machine-readable. Authors typically do not care about theoretical purity and are motivated to add additional markup when they see a tangible benefit. With Mavo, they expend effort because it makes their static website editable or creates a web application. As a side effect, however, they enrich the Semantic Web.

Mavo is not the first system to have attempted to increase HTML author motivation for using Semantic Web technologies. Sync Kit [7] was a client-side caching toolkit which, like Mavo, also used a Semantic Web technology (Microdata [34]) for marking up its HTML templates. However, its utility was to improve performance by caching client-side templates, not to enable HTML authors to create web applications. Its target group was still programmers, as evidenced by the fact that data was provided to it via SQL queries intermixed with the HTML. In addition, all computation had to be performed via the SQL query, as the language did not have computation capabilities.

Chapter 3
Mavo

A description of the Mavo language follows. We first describe its syntax for data specification, editing, and storage, then its expression language for lightweight reactive computation.

3.1 Building data-driven CRUD applications

3.1.1 Declarative, HTML-based Syntax

We chose to use HTML elements, attributes, and classes instead of new syntax for Mavo functionality because our target authors are already familiar with HTML syntax. Whenever possible, we reused concepts from other parts of HTML.

Using HTML5 as the base language also means a WYSIWYG editor for Mavo applications can be easily created by extending any existing WYSIWYG HTML editor. But as discussed previously, we consider it a key contribution of Mavo that it is a system-independent *language*. For example, we expect most Mavo authors to frequently take advantage of the ability to "view source" and work with arbitrary HTML. View source is an essential methodology for learning and adopting new elements of web design. It permits authors to copy and tweak others' designs (even without fully understanding them) without worrying about new or conflicting system dependencies [6]. Source editing is essential to let authors circumvent any limitations imposed by graphical editing tools. Per [27], we want a low *threshold* (cost to get started) while allowing users escape the low *ceiling* (maximum achievable power) of GUI-based tool builders.

As a result of this design, Mavo templates are *endomorphic*, *annotational*, and *idempotent* [5].

The advantage of annotational templates is that they enable the template file to appear as the finished product with regard to design. Idempotency opens up several possibilities for future performance optimizations, such as server-side rendering and client-side caching.

3.1.2 Creating a Mavo application

To enable Mavo functionality on an HTML structure, the author places an `mv-app` attribute on an enclosing element. Its (optional) value gives a name to the application. If no value is provided, the name of the application is taken from the `id` attribute or automatically generated if no `id` attribute is present.

The `mv-app` attribute does not do anything on its own, but enables all other attributes and the expression syntax that is described in 3.2. The element that the `mv-app` attribute is set on will henceforth be referred to as the *Mavo wrapper*.

3.1.3 Storage location

Saving the data produced by the application is as simple as setting an `mv-storage` attribute on that same element. Its value specifies where the data will be stored, through a URI or keyword. If this attribute is not used, or no value is provided, or the value is "none", the data is not stored anywhere, which can be useful for certain types of applications whose purpose is merely to transform transient data, such as calculators and code generators.

Several types of storage backends are supported out of the box, and are discussed below. There is also a flexible API for third-party developers to easily add support for additional storage services.

HTML ELEMENT

If the value of the `mv-storage` attribute is of the form `#id`, the data is stored as JSON inside the designated HTML element. Since such DOM changes are not persisted by default, at first glance this appears to be of limited utility beyond debugging. However, it could be useful in conjunction with another library that monitors the contents of said element and utilizes them in some way. In this case, the other library does not need to be aware of Mavo's existence, and the author combining Mavo with that library does not need to understand programming, so it's a good, decoupled, extension point.

Local storage

If the value of the `mv-storage` attribute is the `local` keyword, the data is persisted in the browser's `localStorage`. This is useful for PIM-type applications where one person is editing their personal data and does not need to share them with others, such as task lists, contact managers, calendars etc.

Github

Github.com is a popular service for hosting git repositories. If the value of the `mv-storage` attribute is a Github URL, the data is persisted in a JSON file on Github. The URL can be less or more specific. If it points to a specific file (e.g. `https://github.com/leaverou/mavo/data.json`), that file is used for storage. If it points to an entire repository, a JSON file with a filename of `[appname].json` is created, and if a repository is also missing from the URL (i.e. it is of the form `http://github.com/[username]`), an `mv-data` repository is created. This allows novices to use Github for storing data without needing to know what a repository even is, or how to create one: They only need to sign up for the service, and they instantly have a usable data store.

If the Github storage backend is used, Mavo also takes care of authentication and prompts users to authenticate before making any edits to the data (or even viewing the data if the repository is private). Figure 3-1

Dropbox

Dropbox is a popular file sharing service. If the value of the `mv-storage` attribute is a Dropbox "share" URL (a publicly viewable URL users can obtain by clicking on the "Share" button through Dropbox's UI), that file is used for storing data.

Akin to the Github storage backend, Mavo takes care of authentication to Dropbox before allowing editing and saving.

Because Dropbox "share" URLs do not follow a predictable format, Mavo cannot create the

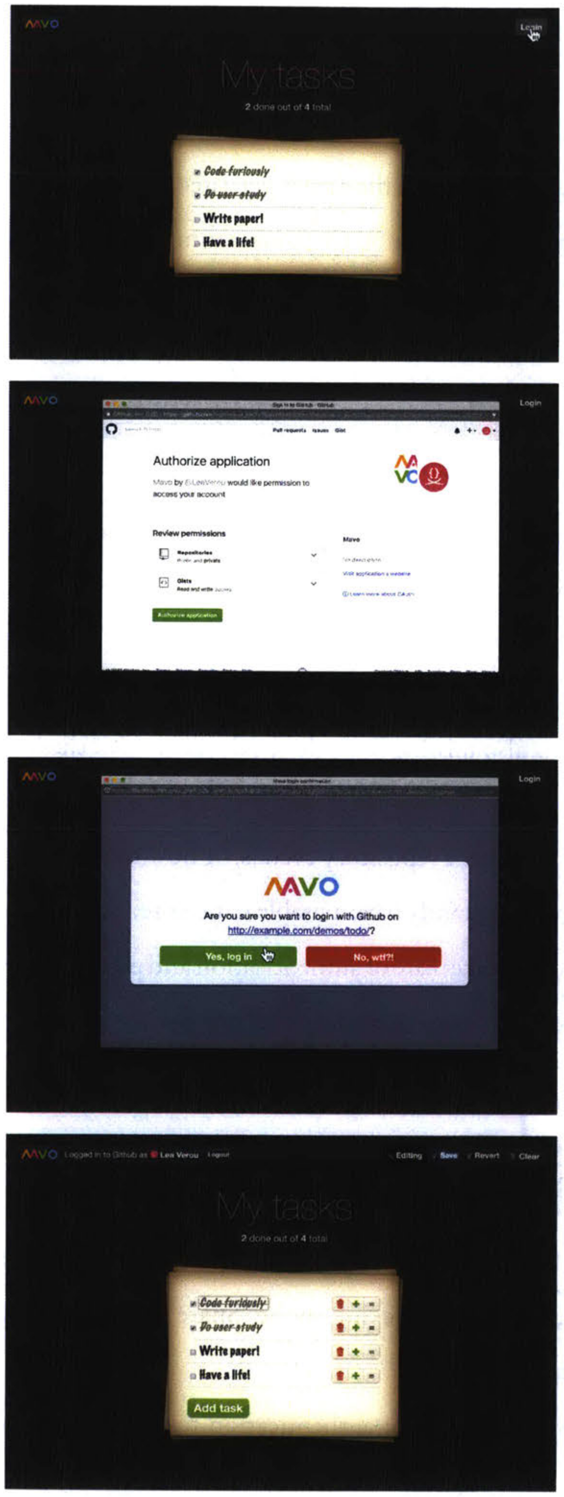


Figure 3-1: Authentication flow when using Github as a storage backend. Note that the data is readable without authenticating (unless the repository is private), but editing the data requires logging in.
The Github confirmation (Step 2) would not be there if the user has ever logged in to any Github-backed Mavo application before, since they all share the same OAuth application on Github's side.
Therefore, the additional confirmation from Mavo's authentication server (Step 3) is necessary, otherwise any malicious JavaScript application could pretend to be a Mavo application, take advantage of the automatic login and then corrupt the user's Github data once it acquires access to the API.

file if it does not exist. Therefore, the author needs to place an empty file in their Dropbox account and obtain a "share" URL to it before they can start using Mavo to store data.

3.1.4 Initialization data

It is often desirable to start with existing data instead of an empty data store. For this purpose, we support an `mv-init` attribute which provides such initialization data. The format of the value follows the same conventions as for the `mv-storage` attribute.

Data provided through `mv-init` is only rendered if the main data store (provided via `mv-storage` is empty.

3.1.5 Overriding storage through the URL

It is possible to temporarily override any Mavo application's storage backend via the URL. The `storage` URL query string parameter overrides the storage backend of the first Mavo application in the page, and `storage_[mavoId]` allows overriding the storage backend of any Mavo application in the page by referencing its name (mavoId).

This enables anyone to use any Mavo application on the Web for storing their own data, in any location they wish, without the application author having to plan for this use case. For example, an invoicing system built by one person can be used by anyone for generating their own invoices by just overriding its storage via the URL. This way, the Mavo application becomes a data editor, instead of bundling a fixed data source.

3.1.6 Data Definition

A core capability of Mavo is to define and materialize data on a web page. Once Mavo is enabled on an HTML structure, it looks for elements with `property` (or `itemprop`) attributes within that structure in order to infer the data schema. These elements are henceforth referred to as simply *properties*. If the HTML author is aware of semantic Web technologies such as RDFa [31] or Microdata, these attributes may be already present in their markup. If not, authors are simply instructed to use a `property` attribute to "name" their element in order to make it editable and persistent. An example of this usage can be found in Figure 1-1.

When an element becomes a property, it is associated with a data value. This value is automatically loaded from and stored to the specified `mv-storage`. For many elements (e.g. `<span>`), the natural place for this value to be "presented" is in the element's contents. In others, such as `<img>` or `<a>`, the natural place for a value is a "primary" attribute (`src` and `href` respectively). These defaults can be overridden. For example, adding `mv-attribute="title"` to a property element means that its data value should be placed in the `title` attribute.

It is worth noting that since Mavo templates are annotational, the example can be filled with real data, which makes the template really look like the output, unlike other templating languages where the template is filled with visible markup. In addition, this example data can easily become default values, by using the `mv-default` attribute without a value.

Mavo vs RDFa

As previously mentioned, Mavo's data definition syntax is based on RDFa [31]. However, usability and flexibility were prioritized over strict specification compliance. Therefore, there are several divergences from RDFa syntax, mainly centered around reducing the amount of metadata required from the author, detailed in Table ??.

3.1.7 Data Editing

Mavo generates UI (user interface) controls for toggling between reading and editing mode on the page, as well as saving and reverting to the last saved state (if applicable), as seen at the top of Figure 1-1 and the last screenshot in Figure 3-1. In editing mode, Mavo presents a WYSIWYG editing widget for any property that is not a form control, which appears only when the user shows intent to interact with it. The generated editing UI depends on the type of the element. For instance, a `<time>` element will be edited via a date or time picker, whereas an `<img>` element will be edited via a popup that allows specifying a URL or uploading an image (Figure 3-2). The assumed data type can be overridden by using the `datatype` attribute (e.g. `datatype="number"`).

Mavo leverages available semantics to optimize the editing interface. For example, using a `<span>` to display dates would result in editing via a generic textfield. However, a `<time>` element

RDFa	Mavo
The `vocab` attribute is mandatory	The `vocab` attribute is supported, but is optional
`property` attributes must have a value	Value-less `property` attributes are allowed. Their value is computed by looking in other identifying attributes, such as `class`, `id`, `name`, `itemprop`. This helps prevent duplication and minimizes the markup required to be added to existing HTML.
The location (content or primary attribute) of an element's data is not configurable and depends solely on its type.	There are extensible defaults for elements that are selector-based instead of type-based, but they can also be overridden via `mv-attribute`.
Objects are explicitly declared using a `typeof` attribute and a type, e.g. `typeof="Person"`	The `typeof` attribute is supported but the value (object type) is optional, since Mavo does not need it. However, objects are mainly inferred from the nesting of properties.

Table 3.1: Differences of RDFa syntax and the Mavo data specification syntax

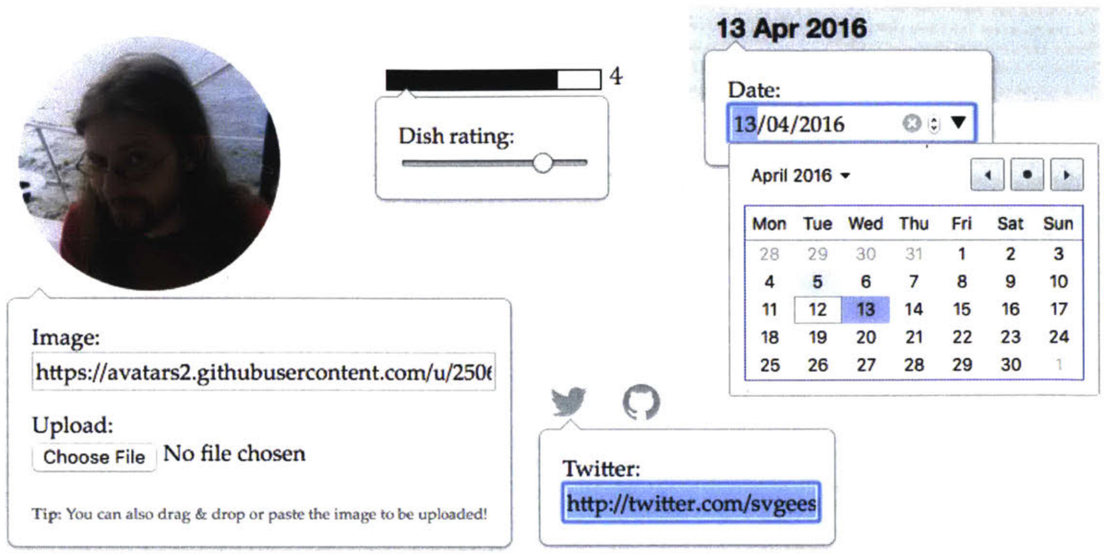


Image: https://avatars2.githubusercontent.com/u/250

Upload:

Tip: You can also drag & drop or paste the image to be uploaded!

Date: 13/04/2016

Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1

Twitter: http://twitter.com/svgees

Figure 3-2: Different types of editing widgets for different types of elements. Clockwise from the top left: `<img>`, `<meter>`, `<time>`, `<a>`

is edited with a time or date picker (depending on the format of its `datetime` attribute). This has the side effect of incentivizing authors to use semantically appropriate HTML.

3.1.8 Customizable Editors

Mavo is designed to be useful to HTML authors across a wide range of skill levels, including web design professionals. Thus, the generated editing GUI is fully customizable:

- Any Mavo UI elements can be fully re-skinned using CSS. In addition, authors can provide their own UI elements by using certain class names (such as `class="mv-add-task"` for a custom "Add task" button).
- The way an element is edited can be customized by nesting a form element inside it. For example, if a property only accepts certain predefined values, authors can express this by putting a `<select>` menu inside the element, essentially declaring it as an enum. An alternative to nesting is referencing a form element anywhere in the page via the `mv-edit` attribute. Any changes to the linked form element are propagated to the property editors. This way, authors can have dynamic editing widgets which could even be Mavo apps themselves, e.g. a dropdown menu with a list of countries populated from remote data and used in multiple Mavo apps.

3.1.9 Objects

Properties that contain other properties (or that contain a `typeof` attribute) become *grouping elements* or *groups*, the Mavo equivalent of objects. This permits authors to define multi-level schemas. For example, an element with a `student` property can contain other elements with `name`, `age`, and `grade` properties, indicating that these properties "belong" to the student.

3.1.10 Collections

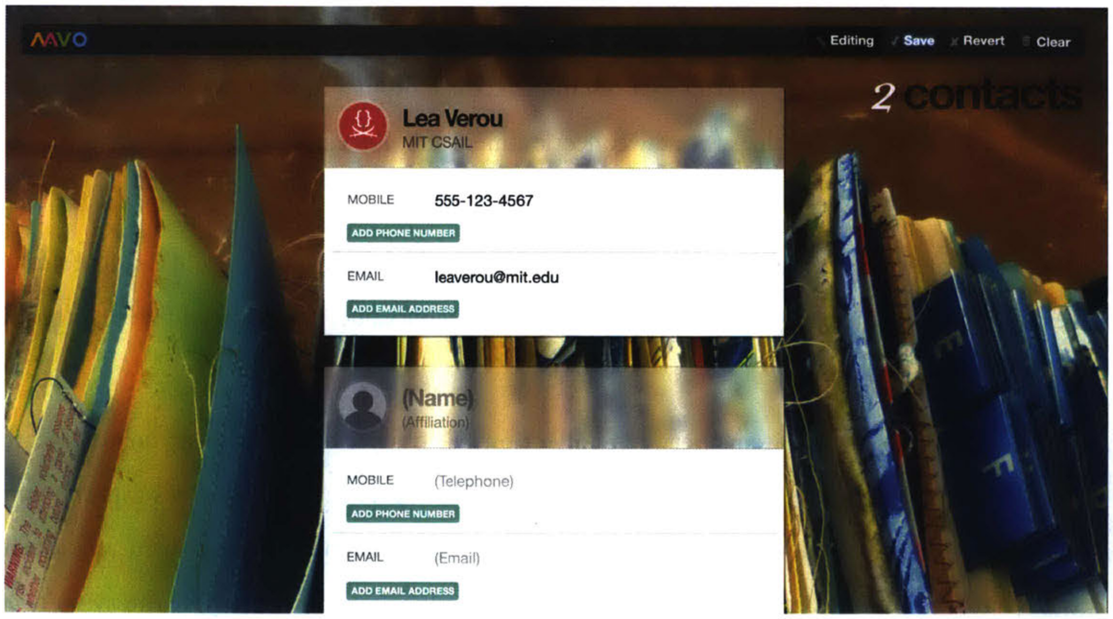
Adding an `mv-multiple` attribute to a property element makes it a collection with that element as the item template. During editing, appropriate controls appear for adding items, deleting items, or reordering items via drag and drop, as seen for the to-do items in Figure 1-1. Collection items can themselves be complex HTML structures consisting of multiple data-carrying elements and even nested collections. This enables the author to visually define schemas with one-to-many relationships.

To author a collection, the author creates *one* representative example of a collection item; Mavo uses this as the archetype for any number of collection elements added later. As discussed earlier, the archetype can contain real data so it resembles actual output and not just a template, and can also provide default data values for new collection members. For example, Figure 4-2 shows what the markup for a contact manager application could look like.

The example phone number and email will only be used on the first phone and email of the first person. Subsequent items added will not use it as a default value, unless we add an `mv-default` attribute to it.

It is possible to **move items between collections** via drag & drop or the keyboard. For this purpose, there is an `mv-accepts` attribute whose value is a space-separated list of property names of collections that are allowed to move items to the current collection. Note that the relationship is not bi-directional by default; to make it so, both collections would need to have the `mv-accepts` attribute, each with the other collection's property name.

```html
<div mv-app="contacts" mv-store="local">
    <details property="contact" mv-multiple open>
        <summary>
            <img property="picture" src="sample.png">
            <h1>
                <span property="name">Lea Verou</span>
                <span property class="affiliation">MIT CSAIL</span>
            </h1>
        </summary>
        <article property="phoneNumber" mv-multiple>
            <span property="name" mv-default>Mobile</span>
            <span property="telephone">555-123-4567</span>
        </article>
        <article property="emailAddress" mv-multiple>
            <span property="name" mv-default>Email</span>
            <span property="email">leaverou@mit.edu</span>
        </article>
    </details>
</div>
```



2 contacts

Lea Verou
MIT CSAIL

MOBILE 555-123-4567

EMAIL leaverou@mit.edu

(Name)
(Affiliation)

MOBILE (Telephone)

EMAIL (Email)

Figure 3-3: Markup for a Mavo contact manager with example rendering.

3.1.11 Direct Schema Manipulation

Our approach to data definition means that end users define their data by defining the way they want their data to look on the page. This is in contrast to many systems which expect their users to define their data model *first* and then map their model into a view. In the spirit of direct manipulation, Mavo users are manipulating their data schema by manipulating the way the data looks. We believe that our approach is more natural for many designers, permitting them to directly specify their ultimate goal: data that looks a certain way.

3.2 Computation

The aforementioned three attributes—`mv-storage`, `property`, and `mv-multiple`—are sufficient for creating any CRUD content-management application with a hierarchical schema and no computation. However, many CRUD applications in the wild benefit from lightweight computation, such as summing certain values or conditionally showing certain text depending on a data value. To accommodate these use cases, Mavo includes a simple expression syntax.

3.2.1 Expressions

Expressions are delimited by square brackets (`[]`) by default and can be placed anywhere inside the Mavo instance, including in HTML attributes. To avoid triggering unrelated uses of brackets on individual elements, authors can use the `mv-expressions` attribute to customize the syntax or disable expressions altogether. The setting is inherited by descendant elements that lack a `mv-expressions` attribute of their own. For example, for the double-brace expressions common in many templating libraries, authors can use `mv-expressions="{{ }}"`. The starting and ending tokens could even be the same, for example `mv-expressions="$ $"` for LaTeX-style dollar sign expressions. To disable expressions, authors can use `mv-expressions="none"`. The value of the `mv-expressions` attribute is inherited by descendant elements, therefore its scope is the entire HTML subtree its element delimits.

In keeping with our goal of leveraging HTML syntax, we also support an HTML-based syntax, via the attribute mv-value which follows property semantics about which attribute (if any) its computation result is placed in. For example, with mv-value, the following examples:

```
<meter value="[average(rating)]"></meter>
<img src="[picture]" />
<span>[foo]</span>
```

can be rewritten as:

```
<meter mv-value="average(rating)"></meter>
<img mv-value="picture" />
<span mv-value="foo">Fallback content</span>
```

A benefit of this approach is that the initial content of the element with the mv-value attribute can be used as a fallback in case of errors. The mv-value attribute is explained in more detail in 5.1.2, as it was one of the features we added based on user study findings.

However, this syntax (and all alternative HTML-based syntaxes we explored) has several drawbacks:

- It is much more verbose than the [expression] microsyntax
- It becomes especially clumsy when the computation result must be placed in an arbitrary attribute. In that case, it requires using mv-attribute and is still disconnected from the markup that will actually be modified.
- Since the entire value of the mv-value attribute is considered an expression, concatenation requires a function call or the & operator. In contrast to that, with the [expression] microsyntax, the non variable parts of expressions can just be placed next to them, e.g. `<p>5 + 5 = [5 + 5]</p>` results in `5 + 5 = 10`.

The choice of brackets for delineating expressions was based on the observation that non-programmers often naturally use this syntax when composing form letters, such as email templates. In addition, many text editors automatically balance brackets.

Our approach to expressions only partially meets the "declarative, direct manipulation" goal we described in our motivation. It is challenging to specify computation, an abstract process, entirely through direct manipulation. The expression language is similar to that in spreadsheets—

fully reactive with no control flow, which nods towards declarative languages. The widespread adoption of spreadsheets provides evidence that this type of computation is within reach of a large population. Furthermore, placing the expression *in the document*, precisely where its value will be presented, as opposed to referencing values computed in a separate model "elsewhere", fits the spirit of direct manipulation in specifying the view. During our user study several subjects volunteered observations that this was effective.

An earlier version of our system used a more spreadsheet-like `=(expression)` syntax, but we found from preliminary user studies that few users realized the spreadsheet connection. Users also found it difficult to determine where an expression terminated due to parentheses being used inside expressions as well.

3.2.2 Named References

Mavo's expression syntax (*MavoScript*) resembles a typical spreadsheet formula syntax. However, instead of referring to cells by grid coordinates, Mavo formulas refer to properties by name. Every property defined in a Mavo instance becomes a (read-only) variable that can be used in expressions **anywhere** in the Mavo instance. These named references are necessary since Mavo has no predefined grid for row/column references. We consider this necessity a virtue. Instead of referencing mysterious row and column coordinates, an expression uses human-understandable property names. We believe this will decrease bugs caused by misdirected references. Indeed, many spreadsheets offer *named ranges* to provide this benefit of understandable references. For spreadsheets, perhaps the main benefit of the row-column references is having formulas with "relative references" (e.g. to adjacent columns) to automatically update as they are copied down into new rows. But Mavo's automatic duplication of templates in collections means copies are never made by the user, obviating the need for this benefit.

A range of common mathematical and aggregate functions is predefined. As with spreadsheets, we also include an `if(condition, iftrue, iffalse)` function that uses the first argument to choose between the remaining two values. Finally, for power users, Mavo expressions can include arbitrary JavaScript, which is executed in a sandbox environment where properties become read-only variables.

3.2.3 Multi-valued Expressions

If a referenced property is inside a collection, then its value in the expression depends on the expression placement:

1. If the expression is on or *inside* the same item that contains the referenced property, its value resolves to the value of the property in (the corresponding copy of) that item.
2. If the expression is *outside* the `mv-multiple` element that contains the property, i.e. outside the collection, it resolves to a *list* (array) of *all* values of that property inside the collection. These lists can be used as arguments to aggregate functions, such as `average(age)` or `count(visit)`.

MavoScript also supports **array arithmetic**: Operations between arrays are applied element-wise. Operations between arrays and primitives are applied on every array element. For example, `rating > 3` compares every item in `rating` with 3 and returns an array of booleans that can then be fed to a `count()` function. While the exact mechanics of how a `count(rating > 3)` expression works might be too complicated for novices to comprehend, the expression itself reads almost like natural language, something several subjects of our user study remarked on.

3.2.4 MavoScript vs JavaScript

We have already discussed one key difference of MavoScript expressions over JavaScript expressions, namely array arithmetic. However, there are a few more differences, aiming to make the syntax looser and more understandable to novices (Table 3.2).

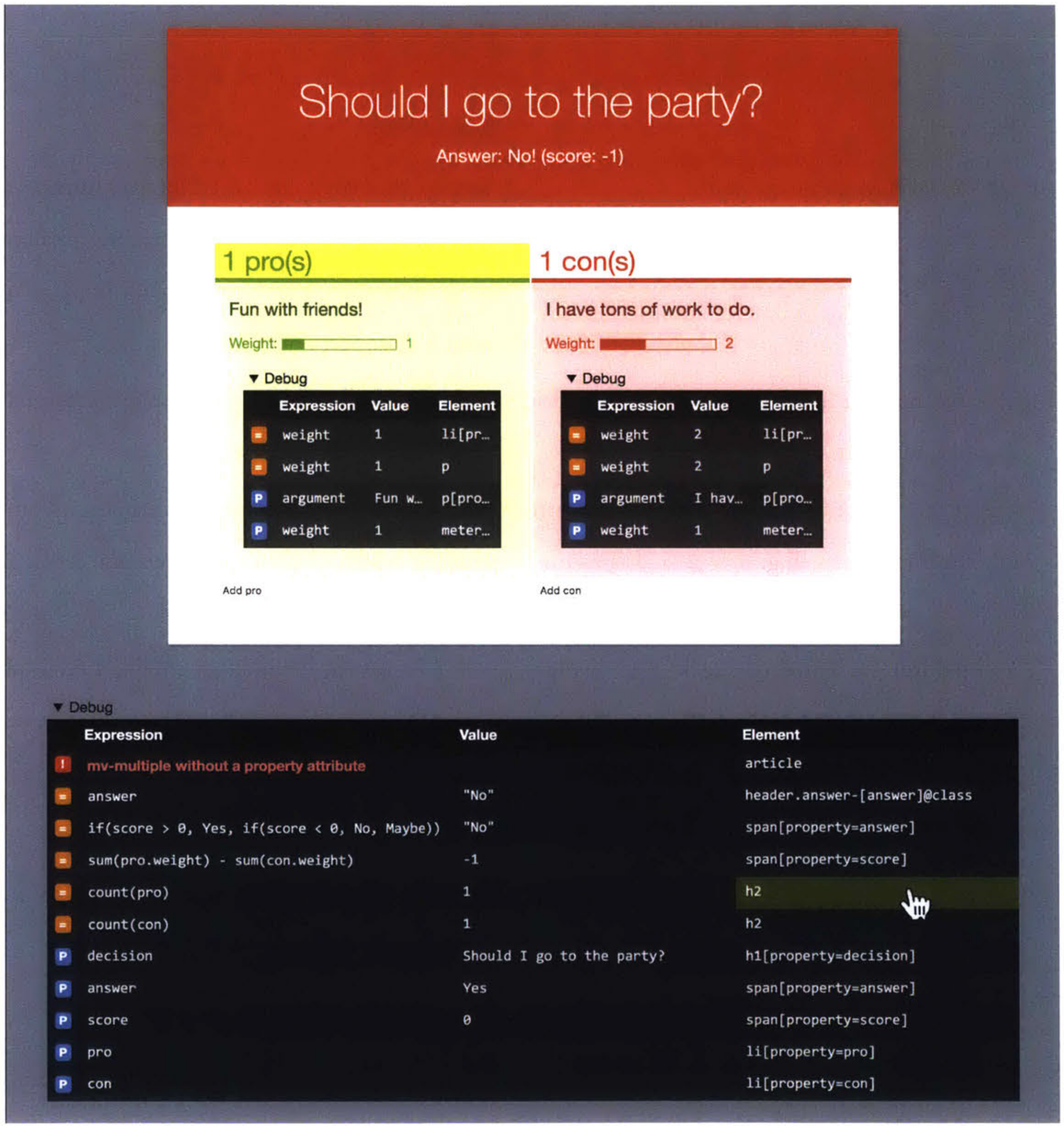
3.3 Debugging

Mavo includes debugging tools that show the current application state, as expandable tables inside each object (Figure 3-4). This is enabled by placing an `mv-debug` class on any ancestor element or using a `debug` URL parameter (i.e. appending `?debug` to the URL, or `&debug` if other query string parameters are present). The latter enables debug tables on every group of every Mavo app on the page, whereas the former is more granular and can enable debug mode only on specific groups.

These tables display a lot of useful information about the current state of each object:

JavaScript	MavoScript
All strings have to be quoted.	Strings that only consist of letters, numbers, and underscores do not need quotes, except to disambiguate them in case they happen to match a property name.
`a + b` can result in addition or concatenation depending on the types of the arguments	`a + b` is always addition, `a & b` is concatenation
Using more than 2 logical operators returns unexpected results. E.g. `3 > 2 > 1` is `false`	Every operator can have multiple operands, including `>`, so `3 > 2 > 1` has the same result as in math (`true`).
== operator for equality check, = for assignment	= or == operator for equality check, there is no assignment.
`&&` operator for logical AND	`and` operator for logical AND
operator for logical OR	`or` operator for logical OR
`if` is a control structure	`if()` is a function. There are no control structures.

Table 3.2: Differences of JavaScript and MavoScript



▼ Debug

	Expression	Value	Element
=	weight	1	li[pr…
=	weight	1	p
P	argument	Fun w…	p[pro…
P	weight	1	meter…

▼ Debug

	Expression	Value	Element
=	weight	2	li[pr…
=	weight	2	p
P	argument	I hav…	p[pro…
P	weight	1	meter…

▼ Debug

	Expression	Value	Element
!	mv-multiple without a property attribute		article
=	answer	"No"	header.answer-[answer]@class
=	if(score > 0, Yes, if(score < 0, No, Maybe))	"No"	span[property=answer]
=	sum(pro.weight) - sum(con.weight)	-1	span[property=score]
=	count(pro)	1	h2
=	count(con)	1	h2
P	decision	Should I go to the party?	h1[property=decision]
P	answer	Yes	span[property=answer]
P	score	0	span[property=score]
P	pro		li[property=pro]
P	con		li[property=con]

Figure 3-4: The debug tools in action, showing local values and warnings.

- Each property and its current value
- Each expression and its current value. The expression is editable and modifying it results in the expression being updated in both the debug tools and the application itself in real time as the user types.
- Warnings about common errors. These include:
  - Using `mv-multiple` without a property attribute
  - Using an invalid property name. This could be either a reserved word, or a name that contains illegal characters.

The third column in the debug table refers to the relevant element. Hovering over that cell will highlight the element in the application, as shown in Figure 3-4.

3.4 Implementation

Mavo is implemented as a JavaScript library that integrates into a web page to simulate native support for Mavo syntax. Therefore, all that is needed to be able to use Mavo on a web page is including Mavo's CSS and JavaScript files in any place in the markup where CSS and JavaScript files are valid, for example in the `<head>` section:

```
<link rel="stylesheet" href="mavo.css"/>
<script src="mavo.js"></script>
```

3.4.1 The Mavo Tree

After the DOM tree of the page is built (`DOMContentLoaded` event), Mavo processes any elements with an `mv-app` attribute and builds an internal *Mavo tree* representation of the schema that their `property` and `mv-multiple` attributes outline (Figure 3-5).

The Mavo tree consists of three types of objects: `Mavo.Group`, `Mavo.Collection`, and `Mavo.Primitive`. All three inherit from the `Mavo.Node` abstract base class. The root of the Mavo tree is always a group and corresponds to the Mavo wrapper element (the element with the `mv-app` attribute).

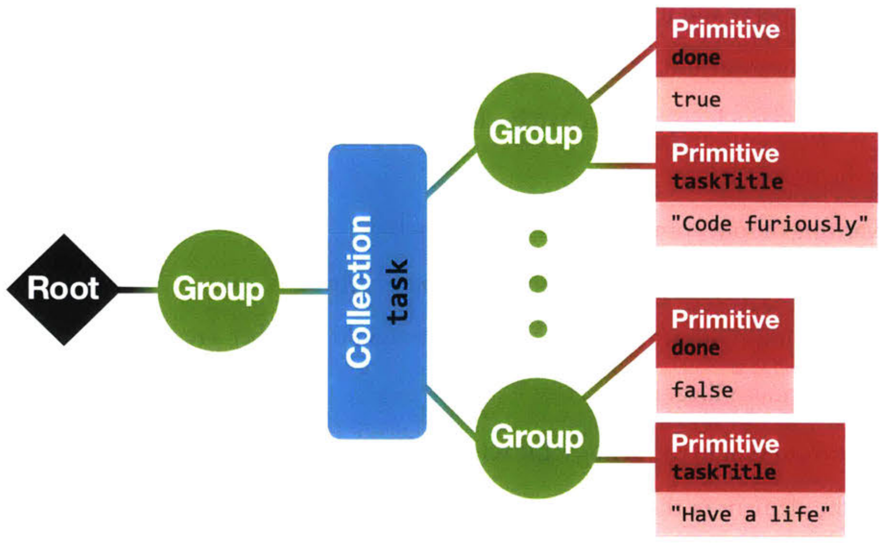


Figure 3-5: The Mavo tree created for the To-Do app shown in Figure 1-1.

Any remote data specified in the `mv-storage` attribute is then fetched asynchronously and recursively rendered.

3.4.2 Expressions

During the DOM traversal that results in the creation of the Mavo tree, Mavo also inspects every text and attribute node on the subtree looking for expressions, and builds corresponding objects. For every expression, a modified JavaScript expressions parser (JSEP [2]) is used to compile MavoScript expressions into JavaScript functions.

Every time the data changes, a `mavo:datachange` event is fired on the related element. This could occur in the following four cases:

id	Operation	Event target
propertychange	A primitive changed value	The property element
add	An item was added to a collection	the new item
delete	An item was deleted from a collection	Collection marker
move	A collection item was moved	All items that changed index

Every data change boils down to one or more of these elementary operations. For example, moving items between collections results in one deletion in the source collection and one addition in the target collection.

These events bubble up the DOM tree, just like every other event. Every time they reach a grouping element, expressions within that grouping element are re-evaluated iff they depend on the changed property.

When the data in an object changes—via rendering, editing or expression evaluation—expressions within it or referring to it are re-evaluated to reflect current values. The execution context is a JavaScript Proxy. Proxies are a relatively new feature (ECMAScript 2015). They enable the creation of objects that allow custom code to be executed on any operation (traps), e.g. when getting or setting a property. This allows Mavo to:

- make MavoScript functions as well as JavaScript's `Math` functions appear to be global scope
- conditionally fetch descendant or ancestor properties only when needed so that the dot notation is only needed for disambiguation
- prevent data modification via expressions
- allow for identifiers to be case insensitive
- allow alphanumeric strings to be unquoted

3.4.3 Performance Optimizations

The most costly Mavo operations are those that involve a large number of DOM operations. These include, but are not limited to, data rendering and expression evaluation. There are a number of performance optimizations in place to accelerate such operations.

Every time a collection item is created during editing or data rendering, its object in the Mavo Tree holds a reference to the first item that was ever created in that collection, which it uses as a template from which it copies property values that do not change across items. Even if the collection is nested inside other collections (so there are multiple instances of the same collection), all of its items use the same template. The collection itself uses its first instance as a template. This allows sharing of certain property values across every collection item without having to recompute them or fetch them from the DOM.

Expression parsing is also a costly operation, since every single text and attribute node in the Mavo application must be inspected. However, once the expressions are found in one collection item, they will be in the same place on any other collection item, albeit on a different DOM node. Therefore, expression objects also store a path from the nearest collection item to the node containing the expression. On new collection items, the element bearing the expression will be different, but the path is the same. This means that new collection items do not have to be traversed for expressions, all we need to do is resolve these paths to create expression objects for them.

Expression evaluation is disabled during data rendering. Given that expressions are re-evaluated on every data change and data change events bubble up the DOM tree, rendering would result in a very large number of expression evaluations, especially on ancestor groups with many deeply nested descendants. Instead, every expression is re-evaluated after data rendering is done, which is $O(N)$ on the size of the dataset.

Another costly operation when performed en masse is editing. Editing properties often involves the creation of several elements. On a collection with several items, creating all these elements at once results in the browser becoming unresponsive for about 0.5 - 1 sec, which is significantly over the 100ms threshold for instantaneous responses [26]. Instead, these elements are created when the user interacts with the property, either via clicking, mousing over for longer than 100ms, or keyboard focus.

3.4.4 Extensibility

Making Mavo extensible has been a priority since the beginning, to enable a plugin ecosystem to significantly extend its scope in the future. JavaScript facilitates this, due to its higher-order functions and mutability of objects.

Hooking is used extensively, as a low-level extension point. Hooks work with two methods: `Mavo.hooks.add(name, callback)` and `Mavo.hooks.run(name, env)`. The former is used by plugins to register callbacks at specific points in Mavo code. The latter is used by Mavo to execute said callbacks. This requires a very good understanding of Mavo's source code by the plugin developer, but affords maximum extensibility, akin to being able to paste code directly in Mavo's source. For example, both Expressions and the Mavo debugging tools are implemented as plugins that utilize hooks and can be removed if not needed, to reduce file size.

In addition to hooks, there are also higher level extensibility points. Certain aspects of Mavo can be extended by merely adding a rule (a property with an object as its value that follows a certain structure) to certain objects or calling certain functions:

Goal	**Object or Function**
New element type	`Mavo.Elements`
New MavoScript function	`Mavo.Functions`
New MavoScript operator	`Mavo.Script.operators`
New storage backend	`Mavo.Backend.register()`

Chapter 4

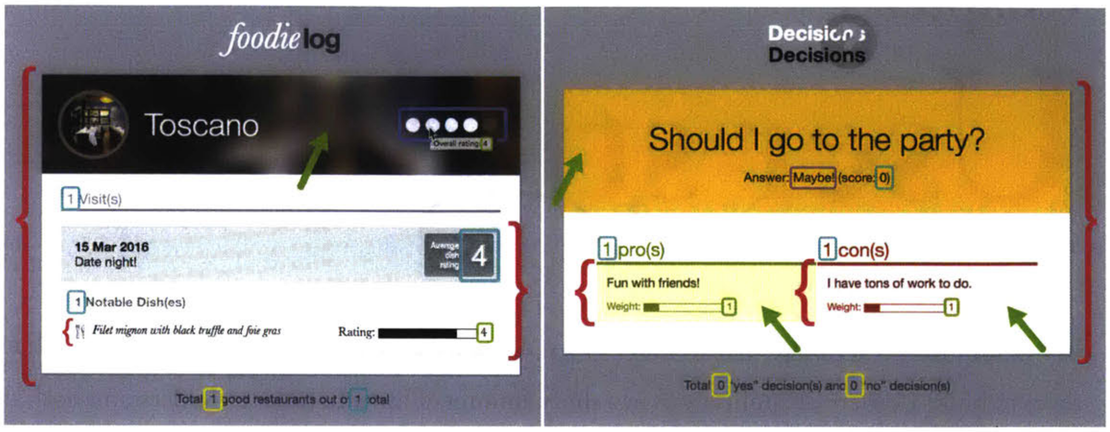
User studies

In our evaluation, we examined whether Mavo could be learned and applied by novice web authors to build a variety of applications in a short amount of time. In order to understand both the usability and flexibility of Mavo, we designed two user studies. For a first STRUCTURED study, we authored static web page mockups of two representative CRUD applications and then gave users a series of Mavo authoring tasks that gradually evolved those mockups into complete applications. This study focused on learnability and usability. For a second FREESTYLE study, *before* telling users about Mavo (so that they would not feel constrained by the capabilities of our system), we asked them to create *their own* mockup of an address book application. Then, during the study, we asked them to use Mavo to convert their mockups into functional applications. This study focused on whether Mavo's capabilities were sufficent to create applications envisioned by users. We carried out the two user studies using three applications. The applications were designed with hierarchical data to test users' ability to generate hierarchical data schemas and perform computations on them.

To facilitate replication of our study, we have published all our study materials online.

4.1 Preparation

We recruited 20 participants (mean age 35.9, SD 10.2; 35% male, 60% female, 5% other) by publishing a call to participation on social media and local web design meetup groups. Of these, 13 performed only the STRUCTURED study, 3 performed only the FREESTYLE study, and 4 performed


foodie log
Toscano
Overall rating 4
1 Visit(s)
15 Mar 2016
Date night!
Average dish rating 4
1 Notable Dish(es)
Filet mignon with black truffle and foie gras
Rating: 4
Total 1 good restaurants out of 1 total
Decisions Decisions
Should I go to the party?
Answer: Maybe! (score: 0)
1 pro(s)
1 con(s)
Fun with friends!
Weight: 1
I have tons of work to do.
Weight: 1
Total 0 "yes" decision(s) and 0 "no" decision(s)

Task category	Example task	Example code	Med. time	Success
Make editable Foodie: 1, Decisions: 1	*"Make the restaurant information editable (name, picture, url, etc)"*	`<h1 property="name">Toscano</h1>`	3:00	100%
🟥 Allow multiple Foodie: 3, Decisions: 2	*"Make it possible to add more pros and cons."*	`<article property="pro" data-multiple>`	1:15	100%
🟩 Simple reference Foodie: 3, Decisions: 3	*"Make the header background dynamic (same image as the restaurant picture)"*	`<header style="background: url([pic])">`	0:43	88%
🟦 Simple aggregate Foodie: 3, Decisions: 2	*"Make the visit rating dynamic (average of dish ratings)"*	`[average(dishRating)]`	0:55	97.5%
🟦 Multi-block aggregate Foodie: 1, Decisions: 0	*"Make the restaurant rating dynamic (average of visit ratings)"*	`<meter value="[average(visitRating)]">`	2:00	77.8%
🟨 Filtered aggregate Foodie: 1, Decisions: 1	*"Show a count of good restaurants"*	`[count(rating > 3)] good restaurants`	6:10	70.9%
🟪 Conditional Foodie: 0, Decisions: 1	*"Show "Yes" if the score is positive, "No" if it's negative, "Maybe" if it's 0."*	`[iff(score>0, Yes, iff(score<0, No, Maybe))]`	5:28	75%

Figure 4-1: User study tasks are shown in the mockups that were given to participants, and results are broken down by task category. The green arrows point to element backgrounds, which participants made dynamic via inline styles or class names. Page elements involved in specific tasks are outlined with color codes shown in the table. "Make editable" tasks are not shown to prevent clutter.

	HTML	CSS	JavaScript
Beginner	0	4	13
Intermediate	8	5	6
Advanced	9	6	1
Expert	3	5	0

Table 4.1: User study participants' familiarity with web development languages.

	JSON	RDFa	Microdata	Microformats	SQL
Never heard of it	0	13	9	10	0
Heard of it	6	6	7	4	5
Can read it	2	0	1	1	3
Can edit it	8	1	2	3	8
Can write it from scratch	4	0	1	1	5

Table 4.2: User study participants' familiarity with data technologies

both. All of our participants marked their HTML skills as intermediate (rich text formatting, basic form elements, tables) or above. However, most (19/20) described themselves as intermediate or below in JavaScript (Table 4.1). When they were asked about programming languages in general, 13/20 described themselves as beginners or worse in *any* programming language, while 7/20 considered themselves intermediate or better. In addition, when we asked participants about their experience with various data concepts, only 4/20 stated they could write JSON, 5/20 could write SQL, and only 1 could write HTML metadata (RDFa, Microdata, Microformats).

Before either study, we gave each user a tutorial on Mavo, interspersed with practice tasks on a simple inventory application. This took 45 minutes on average and covered the `property` attribute (10 minutes), the `mv-multiple` attribute (10 minutes), and expressions using the `[]` syntax, broken down into how to reference properties and perform computations (5 minutes), aggregates such as `count()` (10 minutes), and `if()` syntax and logic (10 minutes).

4.2 The Structured Study

For the STRUCTURED study, we created two applications. 17 subjects were given static HTML and CSS mockups of one of these applications and were asked to carry out a series of tasks by editing the HTML. These tasks tested their ability to use different aspects of Mavo, as shown in Figure 4-1. Eight of these users were given a mockup of a **Decisions app**, a tool for making decisions by summing weighted pros and cons. The application also shows a suggested decision based on the sums of pro and con weights. The other 9 users were given a mockup of a **Foodie log**, a restaurant visit tracker that includes dishes eaten on each visit with individual ratings per dish. The application also computes average ratings for each visit and each restaurant. Both of these applications have a hierarchical data schema.

Each subject was shown a fully functional version of their respective application (but not its HTML source) before being given the static HTML template. While a CSS style file was provided, they did not have to look at it. We provided tasks to the user one at a time, letting them complete one before revealing the next. Tasks were administered in the same order, and we measured the time each subject took to complete the task as well as screen recorded their typing. Participants were asked to speak aloud their thoughts and confusions as they worked. Researchers were silent except to alert subjects to spelling mistakes and to explain HTML and CSS concepts—such as how to set a value on a `<meter>` tag—if subjects were unaware of them. If subjects spent over 15 minutes on a task but were not close to succeeding, the researchers stepped in to offer hints or explain the answer, and marked the task as failed.

4.2.1 Study Tasks

In the case of the Decisions app, users had 10 tasks to complete, while for the Foodie log, users had 12 tasks. The tasks increased in difficulty in order to challenge the users. We grouped the tasks into 7 categories, where each category tests a particular aspect of Mavo. Example tasks, code solutions, and the number of tasks in each category per application is in Figure 4-1. As footnoted earlier, all this task data is available online. A description of each task category follows:

- **Make editable** Adding `property` attributes to different HTML tags to make them editable.

- **Allow multiple** Turn an element into a collection, by adding `property` and `mv-multiple`.
- **Simple reference** Display the value of a property somewhere else, via a `[propertyName]` expression.
- **Simple aggregate** Show the result of a simple aggregate calculation, such as the count or sum of something.
- **Multi-block aggregate** Aggregate calculation on a dynamic property, such as an average of counts.
- **Filtered aggregate** Show how many items satisfy a given condition.
- **Conditional** Show different text depending on a condition.

4.2.2 Results

In the STRUCTURED studies, *before* providing the tasks, we showed users the finished application they were tasked to create and asked them how long they thought it would take them. Of the 17 users, 5 estimated it would take them several hours, 6 estimated days, 3 estimated weeks, and 3 estimated months. Some users said that they would need to learn new skills or that they had no idea where to start.

After going through the tutorial, 6 users went on to complete all the tasks for their application with no failures, 1 user had no failures but had to leave before the last task, and 10 users failed at one or more tasks. A detailed breakdown follows.

Number of failures	Number of Subjects
0	7
1	5
2	2
3	3

All failures were concentrated on expression tasks, usually the most difficult ones. The success rate for basic CRUD functionality was 100%. The 6 users who completed all tasks successfully took on average 17.3 minutes (Decisions, 10 tasks) and 22.5 minutes (Foodie, 12 tasks) to build the entire application. Figure 4-1 shows the median time taken and success rate for each category of task for all 17 users. As can be seen, some task categories were easier for participants to carry out

Application	Before Mavo	After Mavo	Difference
Decisions app	2.875	1.375	-1.5
Foodie log	3.667	1.223	-2.445

Table 4.3: Reported difficulty ratings on a 5-point Likert scale, before and after the study

than others. For instance, all participants quickly learned where to place the `property` and `mv-multiple` attributes, taking a median of 3 minutes to make several elements editable via `property` and a little over a minute to turn single elements into collections. Almost all participants were also able to display simple aggregates, such as showing a count of restaurant visits or a decision score (sum of pro weights - sum of con weights). However, some participants struggled with more complicated expressions, such as conditionals or multi-block aggregates. We explore some of the more common issues next.

We asked these 17 participants who built either the Decisions or Foodie app to rate the difficulty of converting the static page to the fully realized application. They were asked to rate this twice: once after seeing a demo of the final application but before learning about Mavo, and once after going through all the tasks with Mavo. On a 5-point Likert scale, the reported difficulty rating after building the app with Mavo dropped 2.06 points on average from its pre-Mavo rating. Detailed breakdown of pre- and post-study ratings can be found in Table 4.3.

4.2.3 Common Mistakes

The most prevalent error was putting `mv-multiple` on the wrong element—usually the parent container—with 40% of participants stumbling on it at some point. However, as soon as users saw that they were getting copies of the wrong element, they immediately figured out the issue. As the user's *intent* was always clear, a WYSIWYG editor would solve this in the future. Another similarly common and quick-to-fix mistake was forgetting `mv-multiple` (25%). None of these mistakes led to failures on a task.

We noticed that users had a hard time grasping or realizing they could do concatenation. Both the Decisions and Foodie applications included 3 simple reference tasks. We noticed that the

HTML fragment	Success
`</meter> [rating]`	100%
`<meter title="Overall rating: [rating]">`	100%
`</meter> [weight]`	100%
`<header style="background: url([pic])">`	77.8%
`<li class="weight-[weight]">`	75%
`<li class="answer-[answer]">`	75%

Table 4.4: Success rate of simple references.

failure rate was significantly higher (20-25% vs 0%) when the variable part was not separated by whitespace from the static part of the text, as shown in Table 4.4.

Another common mistake was using `sum()` instead of `count()` (20% of participants). This may be because they are thinking of counting in terms of "summing how many items there are", or that they are more familiar with `sum()`, due it being far more common than `count()` in spreadsheets. Interestingly however, there was no correlation between spreadsheet familiarity and occurrence of this mistake. Like the previous mistakes, this was also one that participants typically were able to resolve by themselves, often after another glance at the table of available functions.

We noticed that some participants frequently copied and pasted expressions when they needed the same calculation in different places. A DRY (Don't Repeat Yourself) strategy familiar to programmers would be to create an intermediate variable by surrounding the expression in one place with a tag (such as `<span>` or `<meta>`) that also has a `property`, so that it can be referenced elsewhere. These intermediate properties would reduce clutter and consequently reduce future mistakes down the road; they would also make it easier to modify computations globally. This idea might however be counterintuitive in Mavo as it calls for creating a tag in the HTML that is never intended to be part of the presentation, conflicting with the idea that one authors the application by authoring what they want to see.

The STRUCTURED tasks with the lowest success rate (70.9%) were those that required counting with a filter (`count(rating > 3)`). 25% of participants tried solving these with conditionals, usually of the form `if(rating > 3, count(rating))`, which just printed out the number of

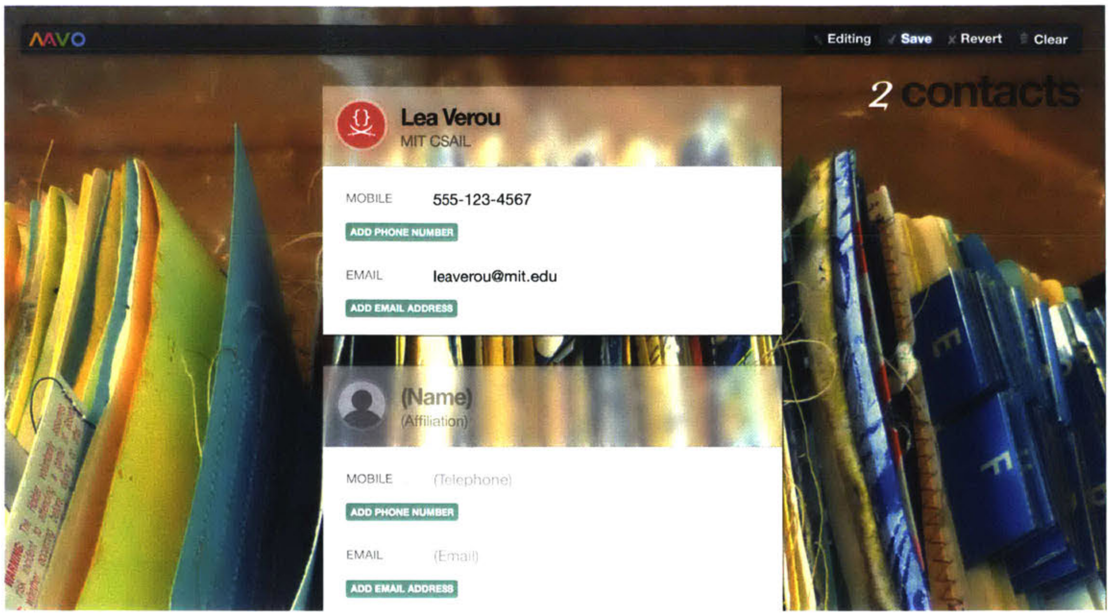


Figure 4-2: A sample of Own Address Book applications created by users.

ratings, since the condition is true if there is at least one rating larger than 3. Most who succeeded remembered or (more often) guessed that they could put a conditional inside `count` and seemed almost surprised when it worked. Another way of completing this task would be to declare intermediate hidden variables computing e.g. `rating > 3` inside each restaurant or decision and then sum or count them outside that scope. Only 10% of participants tried this method, again suggesting that intermediate variables are a foreign concept to this population.

Most participants found `if()` to be one of the hardest concepts to grasp. It is indicative that 40% of subjects tried `if()` when it was not needed, for instance in *simple* reference tasks. 25% of users were unable to successfully complete the conditional task, which required two nested `if()`s or three adjacent `if()` statements, each controlling the appearance of one of the designated words ("Yes", "Maybe", or "No") (Figure 4-3). The latter strategy was only attempted by 37.5% of participants.

In post-study discussions, some users mentioned how conditionals reminded them of what they found hard about programming: "*That's some math and logic which are not my strong points. Just seeing those if statements...I did a little bit of Java and I remember those always screwed me over in that class. No surprise that that also tripped me up here.*" Another user reflected on how having multiple ways of doing something made it more difficult: "*It's hard because there are often multiple*

```
<!-- Solution 1 -->
<span property="answer">
    [if(score > 0, Yes, if(score < 0, No, Maybe))]
</span>
```

```
<!-- Solution 2 -->
<span property="answer">
    [if(score > 0, Yes)][if(score = 0, Maybe)][if(score < 0, No)]
</span>
```

Figure 4-3: The two solutions

ways of doing something. And knowing which one would be the most efficient and best way to do it without making a mistake in the process was hard for me."

4.3 Freestyle Study

Our second FREESTYLE user study involved a third **Own Address Book** application. During recruitment, subjects were asked to create *their own* static mock-up of an address book on their own time prior to meeting us, without being told why. The 7 subjects who complied were assigned to the FREESTYLE study (3 also did the STRUCTURED study first). During our meeting (and after the tutorial), they were asked to add Mavo markup to their own mockup to turn it into a working application.

We added this second study to address several questions. First, we wanted to be sure that our own HTML was not "optimized" for Mavo. Because users were not aware of Mavo at the time they created their application, their decisions were not influenced by perceived strengths and limitations of the Mavo approach. We can therefore posit that these mockups reflected their preferred concept of a contact manager application. Thus, this study served to test whether Mavo is suitable for animating applications that users actually wanted to create. At the same time, it tested whether users could effectively use Mavo to animate "normal" HTML that was written without Mavo in mind.

4.3.1 Study Tasks

Before this FREESTYLE study, we provided no specification of how the application should work or look, except the following basic guidelines:

- Only HTML & CSS should be used, no JavaScript
- If there were lists, they only needed to provide one example in the list
- The mockup needed to contain at least a name, a picture, and a phone number.

Then, during the study session, we asked them to use Mavo to make their mockup fully functional in any way they chose. If the application they envisioned was very simple, after they successfully implemented their application, we encouraged them to consider more complex features, as described in the a section below.

Since what the user worked on depended on their own envisioned implementation, we did not have explicitly defined tasks throughout. However, we did encourage users to try more advanced Mavo capabilities by suggesting the following tasks if they ran out of ideas:

1. Allow phone numbers (or emails) to have a label, such as "Home" or "Work" [Make editable]
2. Allow multiple phone numbers (and/or emails, postal addresses) [Allow multiple]
3. Provide a picture alt text that depends on the person's name (for example, "John Doe's picture") [Simple reference]
4. Show a total count of people (and/or phone numbers, emails) [Simple aggregate]
5. Show "person" vs "people" in the heading, depending on how many contacts there are. [Conditional]

4.3.2 Results of Open-Ended Tasks

Of our participants, 7 brought in their own static mockup of an Address Book app and had time for the FREESTYLE study. We found a variety of implementations of the repeatable contact information portion. One person used a `<table>`, with each row representing a different contact. Three people used `<ul>`, with each contact as a separate list item, and the information about each contact represented inline or as separate `<div>` elements. Two people chose to only use nested `<div>`s, with each contact having their own `<div>`. Finally, one person chose to create a series of 26 `<div>`s, each one a letter of the alphabet, with the intended ability to add contacts within each letter.

When we asked users to use Mavo to improve their mockup in any way, all 7 users chose initially to use the Mavo syntax to make the fields of the app editable and to support multiple contacts, and had no trouble doing so. 4 out of 7 chose, of their own accord, to support multiple phone numbers, emails, or addresses per contact. In all but one case, Mavo was able to accommodate what users envisioned, as well as our extra tasks. In one case (top left in Figure 4-2), the participant wanted grouping and sorting functionality, which Mavo does not support. She was still able to convert her HTML to a web application, but the user had to manually place each contact in the correct one of 26 distinct "first letter" collections. A sample of Own Address Book applications that users created are shown in Figure 4-2.

Five more participants brought Contact Manager mockups, but did not have time to animate them due to participating in the STRUCTURED study first. However, all five mockups were suitable for Mavo and followed the same patterns already observed in the FREESTYLE study.

4.4 General Observations

We conclude this chapter with some general observations applicable to both studies.

4.4.1 Overall Reaction to Data Authoring

The overall reactions to Mavo ranged from positive to enthusiastic. One user who was a programming beginner but used CMSs on a daily basis, said *"Being able to do that...right in the HTML and not have to fool with...a whole other JavaScript file...That is fantastic. I can't say how awesome that is. I'm like, I want this thing now. Can I have a copy please? Please send me an email once it's out."* Along similar lines, another non-programmer said *"When is this going to be available? This is terrific. This is exactly the stuff I have a hard time with"*.

Many participants liked the process of editing the HTML as opposed to editing in a separate file and/or in a separate language. One user said *"It seems much more straightforward, everything is* *right there. You're not referring to some other file somewhere else and have to figure out what connects with what. It's...almost too easy".* Others liked how the Mavo syntax was reminiscent of HTML. One person said *"It didn't seem like a lot of new things had to be learned because naming properties was just the same as giving classes and ids."* Another said *"It's very simple. It's as logical as HTML. You are eliminating one huge step in coding, the need to call the answer at some point, which is really cool...Everything is where it needs to be, not in a different place".*

Other users praised the ability to edit the data from within the browser as opposed to a separate file or data system. One person said, *"I'm convinced it's magic to basically write templating logic and have it show up and be editable. I think there's a lot less cognitive overhead to direct manipulation on the page, especially for a non-technical user".* This unprompted recognition of direct manipulation supports our argument that this approach is natural.

4.4.2 Reaction to Expressions

Many participants were enthused about expressions, even those who failed at a few tasks. One participant said about them: *"It's simpler than I expected it to be. My anxiety expects it to be hard, then I just say 'write what you think' and it turns out to be right. It's very intuitive."* Another user, after learning about filtered aggregates (e.g. `count(age > 5)`) said *"It's so expressive, it tells you exactly what it's doing!"*

Though several subjects struggled with some of the more complicated tasks around expressions, *all* participants easily got the hang of defining a hierarchical data schema within HTML using Mavo. Several participants felt that the Mavo attributes of `property` and `mv-multiple` were powerful even without expressions, and mentioned wanting to use these attributes to replicate functionalities of CMSs that they used. When asked what applications they could see Mavo being useful for, they had many ideas:

- a color palette app
- a movies-watched log
- a basic blog
- an app for tipping
- surveys (two participants)
- contact forms (two participants)

4.4.3 Debugging Behavior

Some participants used the debug table provided to them while others ignored it, instead choosing to look at the visual presentation of the HTML to see where they went wrong. One user even commented out loud that they were not going to look at the debug table at all, then proceeded to fail on a task where a quick glance would have likely prevented this.

A possible explanation is that novices are not used to looking in a separate place for debugging information. The debug tables were visually and spatially disconnected from the rest of their interface, especially on (visually) larger objects. Another possible explanation is that the information density of the table is intimidating to novices. A possible future solution for both of these could be to display values and error messages in a tooltip over the relevant element.

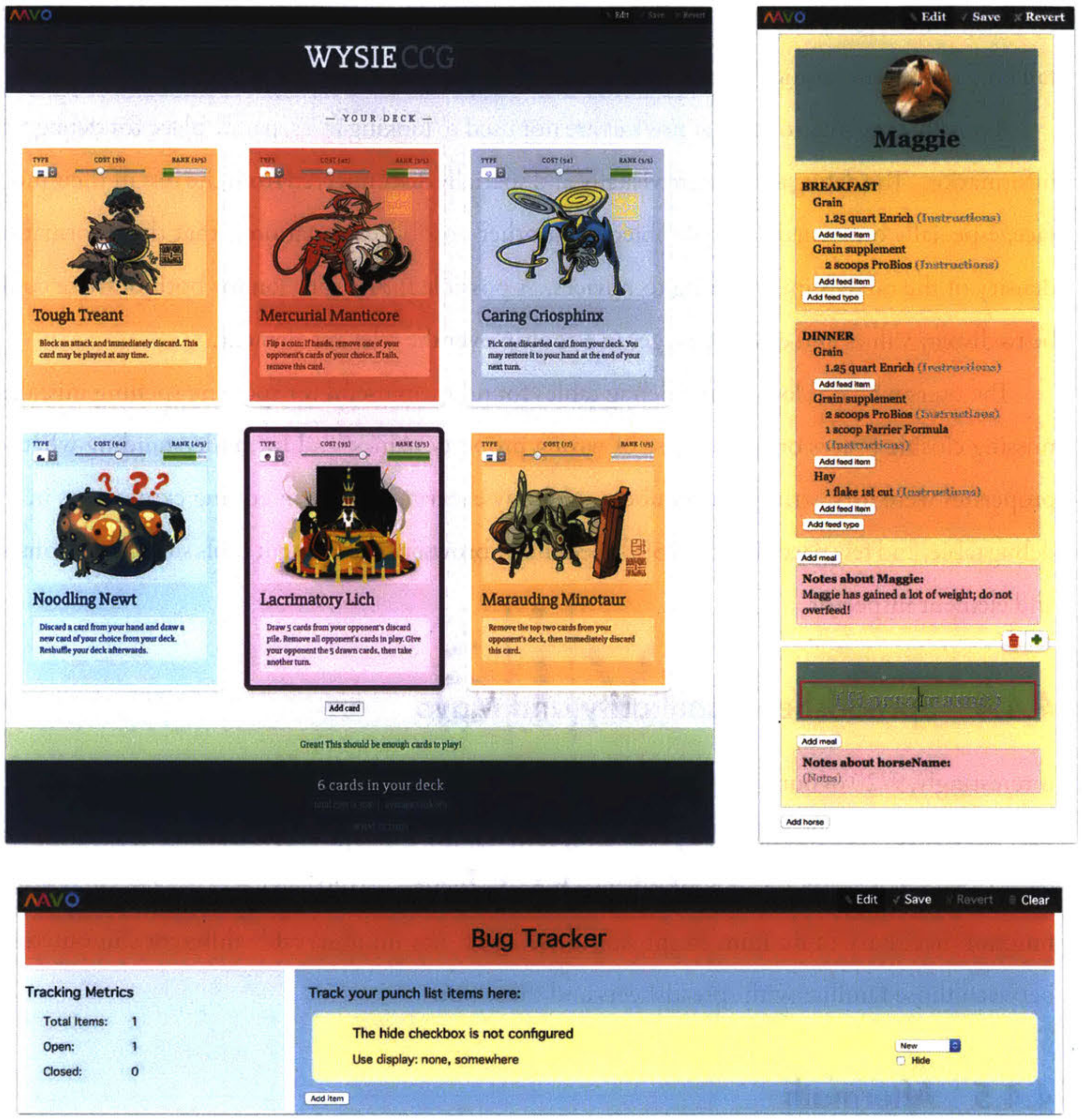
The users who did look at the debug tables found them useful for spotting spelling mistakes, missing closing braces or quotes, use of wrong property names, and for understanding whether properties were lists, strings, or numbers. Nobody experimented with editing expressions in the debug table, and few participants (15%) used the in-browser development tools such as the console and element inspector.

4.4.4 Spreadsheet familiarity and Mavo

Interestingly, 9/20 of our users stated they used spreadsheets rarely or hardly ever while 11/20 said they used them frequently or daily, showing a divergence in usage of spreadsheets. And while all users had used spreadsheets and spreadsheet formulas before, 12/20 had never used the `VLOOKUP()` function necessary to do joins in spreadsheets. There was no observable difference in outcomes between those familiar with spreadsheets and those not.

4.4.5 Aftermath

To further investigate its appeal, we encouraged participants to try out Mavo on their own time after the user study. Three of them went on to create Mavo apps, including a collectible card game, a bug tracker, and a horse feed management application (Figure 4-4). The authors of the first two applications were programming novices, the latter intermediate.



Tough Treant

Block an attack and immediately discard. This card may be played at any time.

Mercurial Manticore

Flip a coin: If heads, remove one of your opponent's cards of your choice. If tails, remove this card.

Caring Criosphinx

Pick one discarded card from your deck. You may restore it to your hand at the end of your next turn.

Noodling Newt

Discard a card from your hand and draw a new card of your choice from your deck. Reshuffle your deck afterwards.

Lacrimatory Lich

Draw 5 cards from your opponent's discard pile. Remove all opponent's cards in play. Give your opponent the 5 drawn cards, then take another turn.

Marauding Minotaur

Remove the top two cards from your opponent's deck, then immediately discard this card.

Great! This should be enough cards to play!

Maggie

BREAKFAST
Grain
1.25 quart Enrich (Instructions)
Grain supplement
2 scoops ProBios (Instructions)

DINNER
Grain
1.25 quart Enrich (Instructions)
Grain supplement
2 scoops ProBios (Instructions)
1 scoop Farrier Formula (Instructions)
Hay
1 flake 1st cut (Instructions)

Notes about Maggie:
Maggie has gained a lot of weight; do not overfeed!

(Horse name)

Notes about horseName:
(Notes)

Bug Tracker

Tracking Metrics

Total Items:	1
Open:	1
Closed:	0

Track your punch list items here:

The hide checkbox is not configured
Use display: none, somewhere

Figure 4-4: Mavo apps independently created by participants. Clockwise: Collectible Card Game, Horse feed management, bug tracker.

Chapter 5

Discussion

In this section we discuss various issues brought up in our design and study of the Mavo language.

5.1 Post-study changes

Our user study showed that Mavo's CRUD capabilities can be easily understood and used by novices, but there is room for improvement on expressions. To rectify this, after the study we implemented a few markup-based expression conveniences, detailed below.

5.1.1 mv-if

Users struggled with conditionals (`if()`) more than anything, possibly due to syntax. To rectify this, we added support for an `mv-if` attribute. This attribute is not merely syntactic sugar over functionality that is already available. Before this attribute, one could only use expressions to conditionally hide an element via inline CSS. The `mv-if` attribute is semantic, not presentational. When its value evaluates to false it removes the element from the DOM entirely. If the element contains properties, their values in expressions become empty, although they are still saved in the data store. If it is inside a property, its content is removed from the property value.

5.1.2 mv-value

In the user study, we noticed that people often lacked the HTML knowledge necessary to use some of the more recent HTML elements without assistance, even though Mavo supported them. For example, both user study applications included HTML5 `<meter>` elements, as inputs in both and as outputs in the Foodie log application (for the restaurant rating).

When `<meter>` elements were inputs, the participants didn't struggle: they knew that putting the `property` attribute on them makes them editable and did not have to know or care about the details, such as which attribute was actually being modified.

However, when they were outputs, participants struggled to place the expression in the right attribute, since they were not familiar with the `<meter>` element. The `mv-value` attribute does for outputs what `property` does for inputs. It follows exactly the same rules and can be overridden with the same `mv-attribute` attribute. With `mv-value`, this markup:

```
<meter value="[average(rating)]"></meter>
<img src="[picture]" />
<span>[foo]</span>
```

becomes:

```
<meter mv-value="average(rating)"></meter>
<img mv-value="picture" />
<span mv-value="foo">Fallback content</span>
```

In addition to attribute-agnostic output, `mv-value` introduces a few other benefits:

- Since the actual data attribute (or element content) does not need to contain the expression, they may contain fallback content that is shown when the expression produces an error. With the bracket system syntax, if an expression produces an error, the output is the expression itself.
- Brackets can be an illegal character in many attributes, e.g. attributes that expect numbers, including many SVG attributes. While browsers do retain illegal values in attributes, there are often errors printed out to the browser developer tools, which could be disconcerting for novices. With `mv-value`, the expression is in a different attribute, with no such restrictions.

However, `mv-value` is not without drawbacks. The main drawback is that there can only be one per element and the expression occupies the entire attribute or element contents. Therefore, static content is now embedded in the expression instead of the expressions being embedded in static content. This inversion of expressions and output means that more extensive changes to the previously static markup are required. With the brackets syntax, the variable part of an attribute is the only part that needs to be edited by the author. However, if the entire attribute is an expression, the entire attribute needs to be edited, for concatenation. An illustrative example is the following:

```
<p>The rating is [rating] out of 100!</p>
```

With `mv-value`, this would become:

```
<p mv-value="'The rating is ' & rating & ' out of 100!'">No data</p>
```

This is much harder to read and feels closer to programming than the first example. Of course, a more realistic `mv-value` solution would be to use an extra element for the variable part:

```
<p>The rating is <span mv-value="rating">?</span> out of 100!</p>
```

5.2 Direct Manipulation of Data Schemas

Mavo's approach of designing schmemas by designing the presentation of the data from those schemas works well because the presentation of data usually reflects its schema. If we have a collection of objects with properties, we generally expect those objects to be shown in a list, with each object's properties presented inside the space allocated to that object. This is understandable, as the visual grouping conveys relationship to the viewer. We are simply inverting this process, arranging for the visual grouping to convey information to the underlying data later. Mavo may not be suitable for creating presentations that conflict strongly with the underlying data schema, should such presentations ever be wanted.

5.3 Target Users

Mavo is aimed at a broad population of users. There is no hard limit to what it can do, since its expressions can also accept arbitrary JavaScript. However, this is not the primary target. Our focus is increasing the power payoff for a given investment of effort/learning that is accessible to novices. Currently, even small web applications require substantial skill and effort to build. Too often, designers of essentially static websites are forced to deploy them inside CMSs, only so that their non-technical clients can update the site content. Mavo frees designers from these CMS constraints by providing an automatic WYSIWYG content management UI for plain HTML. Plain CRUD apps only need `mv-*` attributes "entirely in HTML" without application logic.

For users who want more, expressions add power: lightweight computation for application logic at a conceptual cost similar to spreadsheets. More complex functions provide more power, like advanced spreadsheet ones. Our user study traced out this ease/power curve and showed that most users can work with such expressions.

Although we have focused on Mavo as a tool to support non-programmers, skilled programers can also benefit from the ability to rapidly build dynamic CRUD interfaces. Even for programmers Mavo brings some of the benefits of *data typing* to the construction of the interface: declaring data types enables the system to provide appropriate input and data management without demanding that the developer write special purpose code for the typed content.

5.4 Scalability

Because Mavo is implemented as a pure Javascript library and all computation occurs on the client, serving a Mavo app to any number of users is as easy and scalable as serving static web pages. Scalability issues arise only around access to the *data*, which may be stored locally or outsourced to third-party storage providers such as Dropbox.

Mavo is therefore perfectly suited to so-called *Personal Information Management (PIM)* applications. These applications have a single author and reader, and the amount of data they manage is generally small. For the ultimate in scalability, the Mavo app web page can be stored ("installed") on the user's own machine and data stored locally in the user's browser. While this old fashioned approach sacrifices the access-from-anywhere advantages of cloud-based services, it frees the user of any dependence on the network. Even when operating in the cloud, PIM-oriented Mavo applications scale extremely well because each user's data is isolated. Each user's Mavo simply loads or stores their own small data file, which is the bread-and-butter operation of the popular storage services. A peer-to-peer synchronization service for web storage would allow users to manage information on all their devices while still avoiding dependence on any cloud services.

Mavo is also well suited to "web publishing" applications where an author manages and publishes a moderate-size hierarchical data model and present it to audiences of any size through views enriched by computation of scalar and aggregate functions over those items. This large space spans personal homepages, blogs, portfolios, conference websites, photo albums, color pickers, calculators, and more. Since only the author edits, these applications scale like the PIM applications for editing, while on the consumption side any number of consumers are all simply loading the (static) Mavo application and data file, which again is highly scalable. Conversely, Mavo can be used to supercharge web forms that *collect* information from large numbers of individuals—such as surveys and contact forms—to adapt dynamically to inputs and perform validation computations.

Mavo is not designed to make social or big data apps that present every user with the results of complex queries combining many users' data. This social/computational space is important, but so is the large space of "small data" applications that Mavo can provide. Mavo also does not persist the results of large complex calculations, instead redoing them every time. Again, this is an unimportant issue in small-data applications. Even on big-data applications, Mavo may in the future be a useful component for simplified UI design if powerful back-end servers are used to filter down and deliver only the small amount of data any given user needs in their UI at a given time.

5.5 Multi-User Applications

Mavo can already be used to create basic multi-user applications, since many users can simultaneously visit a Mavo web page and access the underlying data. But access control needs to be implemented by the back-end service and is currently quite coarse. For example, Dropbox only supports read and write access to an entire file. This is adequate for many "small data" applications. However, back-end services with richer access models exist. For instance, DataHub [10] provides row-level access control, where each table row is "owned" by a different user. This would enable apps where users can read others' data but only edit their own. Mavo would need to reflect these permissions in the editing UI it generates. Assuming the backend service provides API methods to determine permissions, this would require few modifications to Mavo.

One planned Mavo capability that would be beneficial for multi-user applications is the ability to combine Mavo instances drawing from different data sources. This would enable uses such as a blog where the posts are stored in Dropbox and can only be edited by the author, with comments that are stored in a service that supports row-level access control. Such functionality would also be very useful for mashups.

Multi-user applications require robust conflict resolution to be able to scale. We plan to support server-sent events to make bidirectional data flows possible, which, in conjunction with auto-saving, should reduce conflicts to a minimum that can be resolved via the UI.

5.6 Handling Large Amounts of Data

The existing Mavo backends save all data in a single JSON file. This is convenient for use cases involving small amounts of data, and allows using any popular file hosting web service as a backend. However, making multi-user apps possible will create a pressing need for handling larger amounts of data with Mavo.

Mavo already supports displaying and editing part of the data, and already keeps track of what data has been modified, to highlight unsaved changes. Therefore, it is easy to implement incremental saving for web services that support it. Implementing a backend adapter for a cloud database service (e.g. Firebase), will also allow for fetching partial data. Such backends usually also support server-sent events, which would enable incremental updates for true bidirectionality.

5.7 Encouraging Semantic Web Best Practices

The Mavo syntax for naming elements is based on a simplified version of RDFa [31] with some divergences discussed in 3.1.6. As a result, at runtime any Mavo instance becomes valid RDFa that can be consumed by any program that needs it. Mavo further incentivizes authors to use good property names by using their identifiers in various places in the generated editing UI: button labels, tooltips, and input placeholders to name a few.

Lastly, in addition to runtime HTML, whenever people edit a website via Mavo, they are also unwittingly producing machine-readable, structured JSON data. If authors are more savvy with Linked Data technologies and use proper RDFa in their markup with a vocabulary and specific types, the JSON produced will also be valid JSON-LD [24].

Chapter 6
Conclusion

This thesis presented Mavo, a language extension of HTML that helps end users convert static webpages to fully-fledged web applications capable of managing, storing and transforming structured data.

Our user studies showed that HTML authors can quickly learn use Mavo attributes to transform static mockups to CRUD applications, and, to a large extent, use Mavo expressions to perform dynamic calculations and data transformations on the existing data.

Mavo represents an architectural argument about what the future of Web authoring should look like. It attempts to make data management and transformation as integral to the Web as paragraphs and colours. We prototyped Mavo as a JavaScript and CSS library to enable HTML authors to experiment with it immediately, but we envision a future where its capabilities are implemented natively by Web browsers.

6.1 Future Work

The Mavo language is still in the beginning of its development. It enables programming novices to do many things they previously were not able to, but there is a lot of room for improvement. In this section, we explore some of the future research and development directions we plan to explore.

6.1.1 Filtering and Sorting

While user study participants were enthusiastic about the potential of building apps with Mavo, there were also a few requested use cases that Mavo cannot presently accommodate. Sorting, searching and filtering were recurring themes. Simple filtering and searching is already possible via expressions and CSS, but not in a straightforward way. We plan to explore more direct ways to declaratively express these operations. Since Mavo makes collections and properties explicit, it doesn't take much more syntax to enable sorting and filtering of a collection on certain properties; however, the more complex question is to develop a sufficiently simple language that can empower users to fully customize any generated sorting and filtering interfaces beyond simple skinning.

One user wanted to filter a list based on web service data (current temperature). Mavo can already incorporate data from any JSON data source, so this will become possible once we support combining data from multiple Mavo instances on the same page.

6.1.2 Dynamic groups and collections

Currently, collections and groups only exist as part of the data schema and there is no way to display dynamic collections whose data have been generated by expressions or resolve expressions in an element based on dynamic data. The ability to define dynamic collections whose data are defined through expressions and are updated when the expression changes would facilitate many use cases, including filtering and dual presentations of the same data, such as a list of concerts and a map displaying their locations.

We are considering two syntax ideas for this functionality:

1. Differentiating the behavior of the `mv-value` attribute depending on whether the result of the expression is a primitive, an object, or an array. The benefit of this approach is its simplicity, as there no new attributes are introduced. The drawback is its unpredictability. The behavior of any expressions inside the element completely changes depending on the return type of the expression. Given that most Mavo users do not have any understanding of data types, this could result in a lot of confusion.
2. Using a new attribute, such as `mv-data`, and requiring `mv-multiple` for collections. This is more explicit and predictable, but requires extending the Mavo language.

For example, with dynamic collections, filtering of a collection of people based on their gender could be implemented as:

```
<ul>
    <li mv-data="unique(gender)" mv-multiple>
        <label>
            <input type="checkbox" property="show" checked />
            [gender] ([count(person.gender = gender)] people)
        </label>
        <meta property="filter" mv-value="if(show, gender)" />
    </li>
</ul>

<article property="person" mv-multiple
   style="display: [if(count(filter = gender) > 0, block, none)]">
    ...
</article>
```

The reason that we are hiding the people with CSS instead if `mv-if` is that since `mv-if` affects expression evaluation, and the filter is dynamic, hiding a gender would cause it to disappear from the filter items so it would be impossible to make it appear again.

While the usability of this markup could certainly be improved, it does produce a fairly high-fidelity filter, with a dynamic display of all values and their counts.

6.1.3 Handling Schema Mutations

Mavo's innovation of inferring schema from HTML presentation might be its Achilles' heel. After Mavo is used to create data, changes to the HTML may result in a mismatch between the schema of the saved data and the new schema inferred from the HTML, which could lead to data loss. Currently Mavo handles only the most basic of such changes, such as:

- When properties are added the schema is automatically extended to include them.
- When properties are *removed*, corresponding data is retained and saved, but not displayed. This protects a user from data loss if they stop displaying a property then bring it back later.

It also enables the creation of multiple Mavo applications operating on different parts of the same dataset.

- When a singleton is made into a `mv-multiple` collection, Mavo converts the single item to a collection of one item.
- When a collection is made into a singleton (by removing the `mv-multiple` attribute), the data is retained so it can be brought back later but anything after the first item is not displayed and cannot be edited or referred to in expressions.
- Property names can be changed by specifying property name aliases using the `mv-alias` attribute.

More complete handling of schema changes is a key open question for Mavo. Our lab study did not explore it because we are not sure what migrations will arise in practice. We plan to release Mavo to the wider public in the coming months and do a field study about how people use it in the wild to create web applications. This will also help identify the types of migrations that are most commonly needed.

The enforced bijection between Mavo schema and data schema may also prevent Mavo from making use of "third party" data that is laid out according to a different schema. We may need to develop language for describing schema mappings to permit incorporation of such data.

Bibliography

[1] Jekyll. https://jekyllrb.com.

[2] JSEP: A tiny JavaScript expression parser. http://jsep.from.so/.

[3] ThemeForest. http://themeforest.net.

[4] Eirik Bakke, David Karger, and Rob Miller. A spreadsheet-based user interface for managing plural relationships in structured data. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2541–2550, New York, NY, USA, 2011. ACM.

[5] Edward Benson. A data aware web architecture. Master's thesis, Massachusetts Institute of Technology, 2010.

[6] Edward Benson and David R Karger. End-users publishing structured information on the web: an observational study of what, why, and how. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 1265–1274. ACM, 2014.

[7] Edward Benson, Adam Marcus, David Karger, and Samuel Madden. Sync kit: a persistent client-side database caching toolkit for data intensive websites. In *Proceedings of the 19th international conference on World wide web*, pages 121–130. ACM, 2010.

[8] Edward Benson, Amy X. Zhang, and David R. Karger. Spreadsheet driven web applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 97–106, New York, NY, USA, 2014. ACM.

[9] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[10] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798*, 2014.

[11] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of functional programming*, 11(02):155–206, 2001.

[12] Michael J Cafarella, Alon Halevy, and Jayant Madhavan. Structured data on the web. *Communications of the ACM*, 54(2):72–79, 2011.

[13] Kerry Shih-Ping Chang and Brad A. Myers. Creating interactive web data applications with spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 87–96, New York, NY, USA, 2014. ACM.

[14] Kerry Shih-Ping Chang and Brad A Myers. Using and exploring hierarchical data in spreadsheets. In *ACM CHI*, 2016.

[15] Ruth Sara Connell. Content management systems: trends in academic libraries. *Information Technology and Libraries (Online)*, 32(2):42, 2013.

[16] Domenic Denicola. Custom Elements. Technical report. https://www.w3.org/TR/custom-elements/.

[17] Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, and Michalis Petropoulos. The sql-based all-declarative forward web application development framework. In *CIDR*, pages 69–78, 2011.

[18] Dimitri Glazkov and Hajime Morrita. HTML Imports. Technical report. https://www.w3.org/TR/html-imports/.

[19] David F Huynh, David R Karger, and Robert C Miller. Exhibit: lightweight structured data publishing. In *Proceedings of the 16th international conference on World Wide Web*, pages 737–746. ACM, 2007.

[20] Google Inc. AngularJS. http://angularjs.org/, 2009.

[21] HV Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 13–24. ACM, 2007.

[22] David R Karger, Scott Ostler, and Ryan Lee. The web page as a wysiwyg end-user customizable database-backed information management application. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 257–260. ACM, 2009.

[23] Keith Kowalzcykowski, Alin Deutsch, Kian Win Ong, Yannis Papakonstantinou, Kevin Keliang Zhao, and Michalis Petropoulos. Do-it-yourself database-driven web applications. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR'09)*. Citeseer, 2009.

[24] Markus Lanthaler, Gregg Kellogg, and Manu Sporny. JSON-LD 1.0. W3c working draft, W3C, January 2014. https://www.w3.org/TR/json-ld.

[25] Chris Lilley. Separation of semantic and presentational markup, to the extent possible, is architecturally sound. Draft TAG finding, W3C, June 2003. https://www.w3.org/2001/tag/doc/contentPresentation-26.html.

[26] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[27] Brad Myers, Scott E Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[28] Mary Beth Rosson, Julie Ballin, and Jochen Rode. Who, what, and how: A survey of informal and professional web developers. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 199–206. IEEE, 2005.

[29] Deborah Seehorn, Stephen Carey, Brian Fuschetto, Irene Lee, Daniel Moix, Dianne O'Grady-Cunniff, Barbara Boucher Owens, Chris Stephenson, and Anita Verno. Csta k–12 computer science standards: Revised 2011, 2011.

[30] Ben Shneiderman. Direct manipulation: a step beyond programming languages. *Sparks of innovation in human-computer interaction*, 17:1993, 1993.

[31] Manu Sporny. W3C HTML+RDFa 1.1 - Second Edition. W3C recommendation, W3C, March 2015. https://www.w3.org/TR/html-rdfa.

[32] Lea Verou, Amy X Zhang, and David R Karger. Mavo: Creating interactive data-driven web applications by authoring html. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 483–496. ACM, 2016.

[33] WHATWG. DOM Living Standard. https://dom.spec.whatwg.org/.

[34] WHATWG. Microdata - HTML Living Standard. https://html.spec.whatwg.org/multipage/microdata.html.

[35] WHATWG. The template element - HTML Living Standard. https://html.spec.whatwg.org/multipage/scripting.html#the-template-element.

[36] Nicholas Wilde and Clayton Lewis. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 153–160. ACM, 1990.

[37] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. Wysiwyg development of data driven web applications. *Proceedings of the VLDB Endowment*, 1(1):163–175, 2008.

[38] Evan You. Vue.js. https://vuejs.org/, 2014.