Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML

Lea Verou
MIT CSAIL
leaverou@mit.edu

Amy X. Zhang
MIT CSAIL
axz@mit.edu

David R. Karger
MIT CSAIL
karger@mit.edu

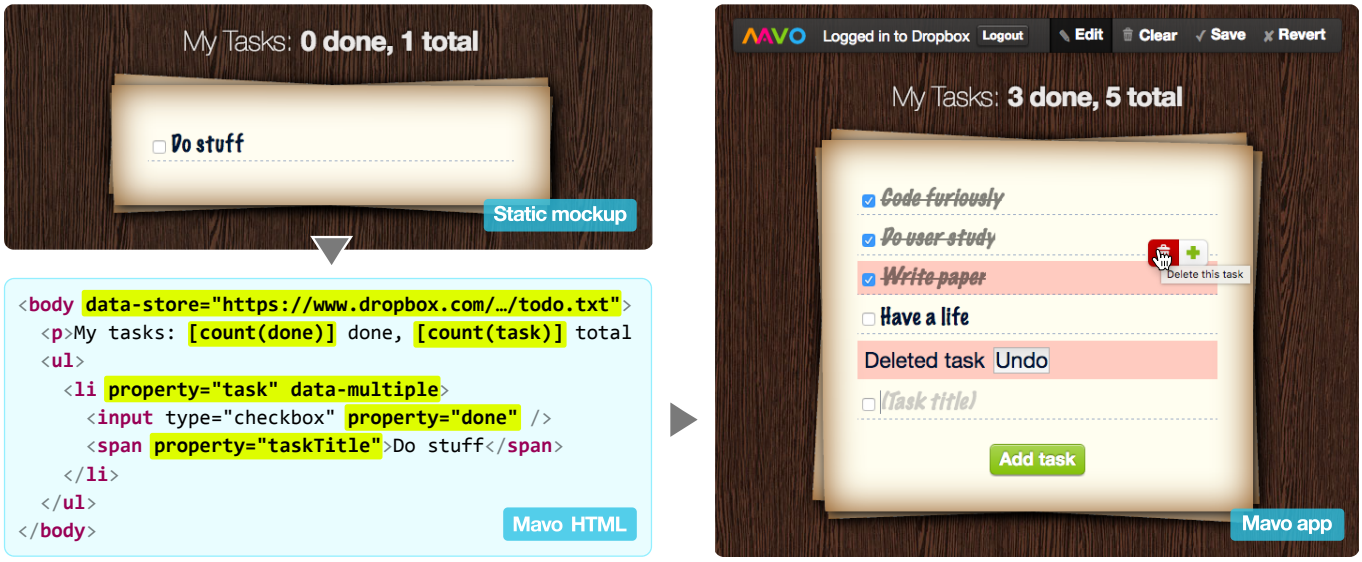

Figure 1: A fully-functional To-Do app made with Mavo, shown with its accompanying code and the starting HTML mockup.

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UIST 2016, October 16–19, 2016, Tokyo, Japan.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4189-9/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2984511.2984551

Author Keywords

Web design; End-user programming; Information architecture; Semantic publishing; Dynamic Media; Web.

ACM Classification Keywords

H.5.4. Information Interfaces and Presentation (e.g. HCI): Hypertext/Hypermedia- User Issues

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. The ACM cites knowledge of HTML and CSS to be at the K-12 level of computer literacy [19].

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 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. It is indicative that even implementing a simple to-do application similar to the one in Figure 1 needs **294**[1] lines of JavaScript (not including comments) with ANGULARJS, **246** with POLYMER, **297** with BACKBONE.JS, and **421** with REACT. Other JavaScript frameworks are in the same ballpark.

Many people who are comfortable with HTML and CSS do not possess additional programming skills[2] and have little experience articulating data schemas [18]. 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 [12]. 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 [4], which CMSs enable via plugins. The interfaces for these plugins do not allow authors to edit data in place on the page; instead they must fill out forms. 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. 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 [2].

Our Contribution

This paper presents and evaluates a new language called Mavo[3] 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 [3] and thus contribute to machine-readable data on the Web.

Mavo is inspired by the principle of *direct manipulation* [20] 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 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 shows how an HTML mockup can be transformed into a fully functioning to-do application by adding only 5 lines of Mavo HTML.

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, 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.

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.

RELATED WORK

Mavo combines ideas from three prior systems that addressed the downsides of CMSs. Dido [15] built on Exhibit [14], extending 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. Quilt [5] 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 computation

[1] Statistics from *todomvc.com*

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

without programming. Gneiss [10, 11] 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.

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.

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.

End User Web Development

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 [4]. However, these require significant technical expertise to configure and offer no graphical interfaces for editing data.

In the previous section, we described three systems—Dido [15], Quilt [5], and Gneiss [10]—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 [4]). Our evaluation studies users working with such hierarchical schemas.

Visual application builders like app2you [16] and AppForge [23] 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 [13], is more powerful but requires writing SQL queries within HTML. Visual programming languages such as Forms/3 [8] and NoPumpG [22] extend the spreadsheet paradigm to graphical interfaces and interactive graphics. However, they do not afford any customization in terms of input UI, have no concept of a separate data store. Also, they only target single-user local web applications and do not address the unique challenges that Web applications raise.

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 [9]. 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 [3] and Microdata [21], for publishing structured data in web pages to contribute to the *Semantic Web* and *Linked Open Data* [6]. 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

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.

Building data-driven CRUD applications

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. [4] 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 [4]. Source editing is essential to let authors circumvent any limitations imposed by graphical editing tools. Per [17], we want a low *threshold* (cost to get started) while allowing users escape the low *ceiling* (maximum achievable power) of GUI-based tool builders.

Storage location

To specify Mavo functionality on an HTML structure, the author places a `data-store` attribute on an enclosing element. Its (optional) value specifies where the data will be stored, through a URI or keyword. If no value is provided, data is not stored anywhere, which can be useful for calculator-type applications. Mavo can store data locally in the page (`data-store="#elementid"`), in the browser's local storage (`data-store="local"`), in an uploaded / downloaded file (`data-store="file"`) or on one of the supported persistent storage services by providing a URL to a file on them. We currently support Dropbox and Github for remote storage, but we provide a flexible API for third-party developers to add support for more services.

For example, to save data in Dropbox, the author provides a Dropbox "share" URL as the value of the `data-store` attribute. Mavo then prompts any user to log in to Dropbox before editing and saving. For storage services with predictable URLs, such as Github, Mavo can even create the file if it does not exist.

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.

[4] To demonstrate, we have prototyped a Mavo WYSIWYG editor, which can be found at **mavo.io/play**

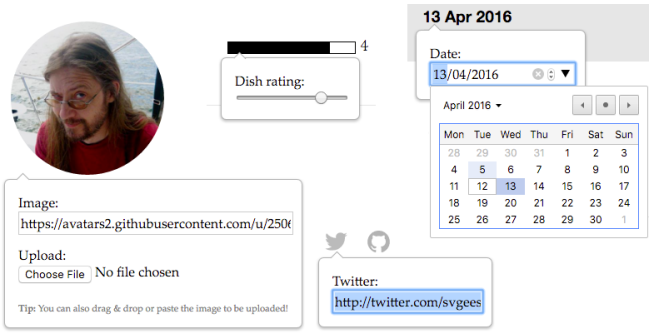

Figure 2: Different types of editing widgets for different types of elements. Clockwise from the top left: `<img>`, `<meter>`, `<time>`, `<a>`

These elements are henceforth referred to as simply *properties*. If the HTML author is aware of semantic Web technologies such as RDFa 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.

When an element becomes a property, it is associated with a data value. This value is automatically loaded from and stored to the specified `data-store`. 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 `data-attribute="title"` to an element with a property says that the value should be placed in the `title` attribute.

It is worth noting that in Mavo, 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 `data-default` attribute without a value.

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. 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 2). The assumed data type can be overridden by using certain class names (e.g. `class="date"`).

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 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.

Customizable Editors

We designed Mavo 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 `data-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.

Objects

Properties that contain other properties become *grouping elements* (objects in programming terminology); this permits a user 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. Inferring objects from the structure of properties instead of requiring an explicit `typeof` attribute is Mavo's main divergence from RDFa. (Explicitly declaring objects via `typeof` attributes is also supported).)

Collections

Adding a `data-multiple` attribute to a property makes it a collection. During editing, appropriate controls appear for adding and deleting new elements in the collection, as seen for the to-do items in Figure 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.

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.

Computation

The aforementioned three attributes—`data-store`, `property`, and `data-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.

One of Mavo's guiding design principles is to reuse existing HTML syntax as much as possible while minimizing the introduction of "programming-like" concepts such as assignment and sequential execution in favor of reactive evaluation, akin to spreadsheets.

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 `data-expressions` attribute to customize the syntax or disable expressions altogether. The setting is inherited by descendant elements that lack a `data-expressions` attribute of their own. For example, for the double-brace expressions common in many templating libraries, authors can use `data-expressions="{{expression}}"`, with `expression` being a literal to separate the start and end markers. To disable expressions, authors can use `data-expressions="none"` (or any other "invalid" value).

In keeping with our goal of leveraging HTML syntax, we explored several HTML-based alternatives, such as `span data-content="expression"` to define a span that should be filled with the value of the expression. But these syntaxes were more verbose and demanded contortions to place expressions into the values of attributes. 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.

Named References

Mavo's expression syntax 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 `iff(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.

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 `data-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)`.

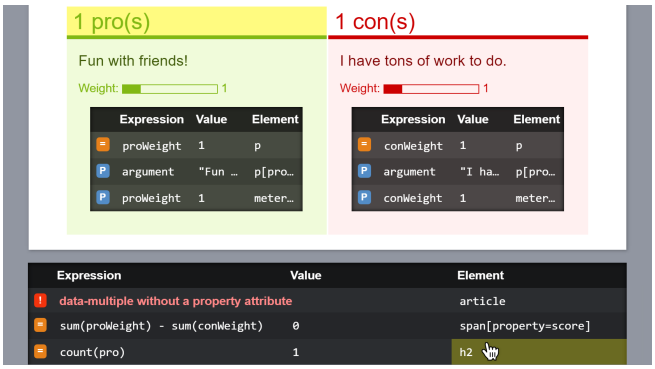

Figure 3: The debug tools in action, showing local values and warnings.

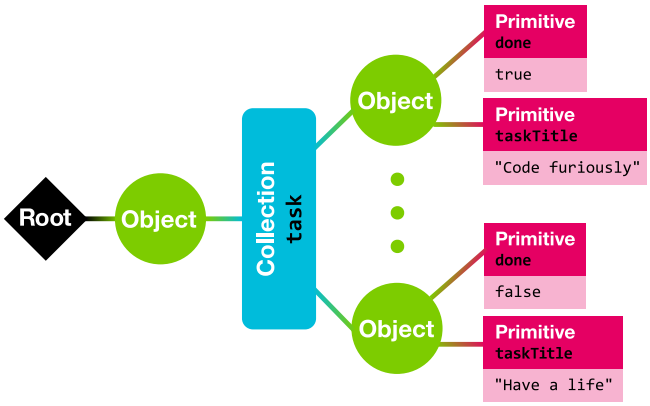

Figure 4: The Mavo tree created for the To-Do app shown in Figure 1.

Our expression syntax 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 e.g. be fed to a `count()` function.

Debugging

Mavo includes debugging tools that show the current application state, as expandable tables inside objects (Figure 3). This is enabled by placing a `mv-debug` class on any ancestor element or adding `?debug` to the URL. These tables display current values of all properties and expressions in their object, and warnings about common errors. Expressions shown can be edited in place, so that users can experiment in real-time.

IMPLEMENTATION

Mavo is implemented as a JavaScript library that integrates into a web page to simulate native support for our syntax. On load, Mavo processes any elements with a `data-store` attribute and builds an internal *Mavo tree* representation of the schema (Figure 4). It also inspects every text and attribute node on or inside every element looking for expressions, and builds corresponding objects for them. For every expression, a JavaScript parser is used to rewrite binary operations as function calls in order to enable array arithmetic.

Any remote data specified in the `data-store` attribute is then fetched and recursively rendered. Every time an object is created during editing or data rendering, it holds a reference to its corresponding node in the Mavo tree (Figure 4), which it uses as a template. This improves performance by only running costly operations (such as finding and parsing expressions) once per collection.

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. This occurs in a special execution context where current object data and Mavo functions appear to be global scope. ECMAScript 2015 Proxies are used behind the scenes to conditionally fetch descendant or ancestor properties only when needed, to allow for

identifiers to be case insensitive, to make the variables read-only, and to allow identifier-like strings to be unquoted.

There are APIs in place for third-party developers to add new default editing widgets, new expression functions, and new storage backends. In addition, Mavo includes a hooks system for developing plugins that modify how it works on a lower level. For example, both Expressions and the Mavo debugging tools are implemented as plugins and can be removed.

EVALUATION

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[5].

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 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. 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 none 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 `data-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 `iff()` syntax and logic (10 minutes).

[5] **http://mavo.io/uist2016/study**

The Structured Study

For the STRUCTURED study, 17 subjects were given static HTML and CSS mockups of one out of two applications that we created and were asked to carry out a series of tasks by editing the HTML. The tasks tested their ability to use different aspects of Mavo, as shown in Figure 5. 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.

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. 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.

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 5. 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 `data-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.

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,

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 5: 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.

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. The 6 users who completed all tasks successfully took on average 17.3 minutes (Decisions) and 22.5 minutes (Foodie) to build the entire application. Of the 10 people who failed one or more times, 5 failed on 1 task, 2 failed on 2 tasks, and 3 failed on 3 tasks. All failures were concentrated on expression tasks, usually the most difficult ones. The success rate for basic CRUD functionality was 100%. Figure 5 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 than others. For instance, all participants quickly learned where to place the `property` and `data-multiple` attributes. 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.

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

Table 1: Success rate of simple references.

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.

Common Mistakes

The most prevalent error was putting `data-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 `data-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 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 1.

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, there was no correlation between spreadsheet familiarity and occurrence of this mistake.

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.

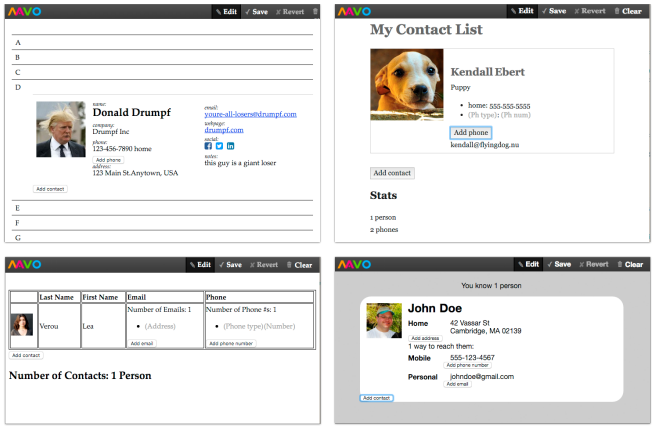

Figure 6: A sample of Own Address Book applications created by users.

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 `iff(rating > 3, count(rating))`, which just printed out the number of 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 `iff()` to be one of the hardest concepts to grasp. 40% of subjects tried `iff()` 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 `iff()`s or three adjacent `iff()` statements, each controlling the appearance of one of the designated words ("Yes", "Maybe", or "No"). 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."*

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.

Study Tasks

Before this FREESTYLE study, we provided no specification of how the application should work or look, except to say that users only needed to use HTML and CSS; that if there were lists, they only needed to provide one example in the list; and that 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]

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, on 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 6), 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 6.

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.

General Observations

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

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.

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*

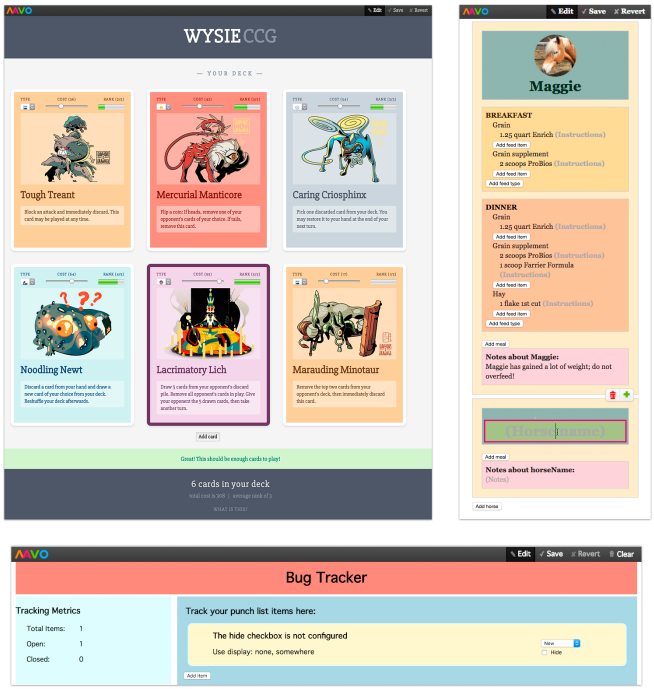

Figure 7: Mavo apps independently created by participants. Clockwise: Collectible Card Game, Horse feed management, bug tracker.

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 users 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 users felt that the Mavo attributes of `property` and `data-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, users mentioned using Mavo to build a color palette app, a movies-watched log, a basic blog, and an app for tipping. Two users mentioned using Mavo for putting out surveys and contact forms. Several mentioned using Mavo to build an online portfolio, with lists of projects.

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. 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.

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 7. The authors of the first two applications were programming novices, the latter intermediate.

DISCUSSION

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

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 presentation that conflict strongly with the underlying data schema, should such presentations ever be wanted.

Target Users

Mavo is aimed at a broad population of users. There is no hard limit to what it can do, since its expressions 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 `data-*` 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.

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.

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 [7] provides row-level access control, where each table row is "owned" by different users. 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.

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.

Encouraging Semantic Web Best Practices

The Mavo syntax for naming elements is based on a simplified version of RDFa. Its main divergence from RDFa is that scopes are inferred from the property structure instead of via a separate `typeof` attribute. Mavo then adds any missing `typeof` attributes. 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.

FUTURE WORK

Improving Expressions

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. Users struggled with conditionals (`iff()`), partly due to syntax. An HTML-based syntax for conditional logic could help rectify this. We are considering an attribute specifically for toggling content (example: `<div data-if="score < 2">No</div>`), which appears to be the most common use case for a conditional statement.

Some participants tried using `sum()` with the property name of the list items instead of the numerical property being summed. Others used `sum()` instead of `count()`. Both of these mistakes printed 0, as `sum()` drops non-numbers. A more meaningful result might be helpful. For example `sum(someObject)` could sum all numerical primitives inside the object, or could treat any non-numbers as `1` in order to have `sum` generalize `count`.

Filtering and Sorting

While 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.

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 `data-multiple` collection, Mavo converts the single item to a collection of one item.
- When a collection is made into a singleton (by removing the `data-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 `data-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.

CONCLUSION

This paper presents Mavo, a system that helps end users convert static HTML pages to fully-fledged web applications for managing 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.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their valuable feedback and our study participants for their help. This research was funded in part by a grant from Wistron Corporation.

REFERENCES

1. Jekyll. `https://jekyllrb.com`.
2. ThemeForest. `http://themeforest.net`.
3. W3C HTML+RDFa 1.1 - Second Edition. `https://www.w3.org/TR/html-rdfa`.
4. Benson, E., and Karger, D. R. 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*, ACM (2014), 1265–1274.
5. Benson, E., Zhang, A. X., and Karger, D. R. Spreadsheet driven web applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, ACM (New York, NY, USA, 2014), 97–106.
6. Berners-Lee, T., Hendler, J., Lassila, O., et al. The semantic web. *Scientific american 284*, 5 (2001), 28–37.
7. Bhardwaj, A., Bhattacherjee, S., Chavan, A., Deshpande, A., Elmore, A. J., Madden, S., and Parameswaran, A. G. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798* (2014).
8. Burnett, M., Atwood, J., Djang, R. W., Reichwein, J., Gottfried, H., and Yang, S. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of functional programming 11*, 02 (2001), 155–206.
9. Cafarella, M. J., Halevy, A., and Madhavan, J. Structured data on the web. *Communications of the ACM 54*, 2 (2011), 72–79.
10. Chang, K. S.-P., and Myers, B. A. Creating interactive web data applications with spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, ACM (New York, NY, USA, 2014), 87–96.
11. Chang, K. S.-P., and Myers, B. A. Using and exploring hierarchical data in spreadsheets. In *ACM CHI* (2016).
12. Connell, R. S. Content management systems: trends in academic libraries. *Information Technology and Libraries (Online) 32*, 2 (2013), 42.

13. Fu, Y., Ong, K. W., Papakonstantinou, Y., and Petropoulos, M. The sql-based all-declarative forward web application development framework. In *CIDR* (2011), 69–78.

14. Huynh, D. F., Karger, D. R., and Miller, R. C. Exhibit: lightweight structured data publishing. In *Proceedings of the 16th international conference on World Wide Web*, ACM (2007), 737–746.

15. Karger, D. R., Ostler, S., and Lee, R. 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*, ACM (2009), 257–260.

16. Kowalzcykowski, K., Deutsch, A., Ong, K. W., Papakonstantinou, Y., Zhao, K. K., and Petropoulos, M. Do-it-yourself database-driven web applications. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDRâĂŹ09)*, Citeseer (2009).

17. Myers, B., Hudson, S. E., and Pausch, R. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI) 7*, 1 (2000), 3–28.

18. Rosson, M. B., Ballin, J., and Rode, J. Who, what, and how: A survey of informal and professional web developers. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, IEEE (2005), 199–206.

19. Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cunniff, D., Owens, B. B., Stephenson, C., and Verno, A. Csta k–12 computer science standards: Revised 2011, 2011.

20. Shneiderman, B. Direct manipulation: a step beyond programming languages. *Sparks of innovation in human-computer interaction 17* (1993), 1993.

21. WHATWG. Microdata - HTML Living Standard. `https://html.spec.whatwg.org/multipage/microdata.html`.

22. Wilde, N., and Lewis, C. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (1990), 153–160.

23. Yang, F., Gupta, N., Botev, C., Churchill, E. F., Levchenko, G., and Shanmugasundaram, J. Wysiwyg development of data driven web applications. *Proceedings of the VLDB Endowment 1*, 1 (2008), 163–175.