Exhibit: Lightweight Structured Data Publishing

David F. Huynh, David R. Karger, Robert C. Miller
MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, Building 32, 32 Vassar Street, Cambridge, MA 02139, USA
{dfhuynh, karger, rcm}@csail.mit.edu

ABSTRACT

The early Web was hailed for giving individuals the same publishing power as large content providers. But over time, large content providers learned to exploit the structure in their data, leveraging databases and server side technologies to provide rich browsing and visualization. Individual authors fall behind once more: neither old-fashioned static pages nor domain-specific publishing frameworks supporting limited customization can match custom database-backed web applications.

In this paper, we propose Exhibit, a lightweight framework for publishing structured data on standard web servers that requires no installation, database administration, or programming. Exhibit lets authors with relatively limited skills—those same enthusiasts who could write HTML pages for the early Web—publish richly interactive pages that exploit the structure of their data for better browsing and visualization. Such structured publishing in turn makes that data more useful to all of its consumers: individual readers get more powerful interfaces, mashup creators can more easily repurpose the data, and Semantic Web enthusiasts can feed the data to the nascent Semantic Web.

Categories and Subject Descriptors

H5.2 [**Information Interfaces and Presentation**]:
User Interfaces – Graphical user interfaces (GUI).
H5.4 [**Information Interfaces and Presentation**]:
Hypertext/Hypermedia – User issues.

General Terms

Design, Human Factors.

Keywords

Publish, presentation, faceted browsing, lens, view, template.

1. INTRODUCTION

This paper describes *Exhibit*, a very lightweight AJAX framework that lets individuals who know only basic HTML create web pages containing rich, dynamic visualizations of structured data and supporting faceted browsing and sorting on that structured data. These authors do not have to install, configure, and maintain any database or to write a single line of server-side code. They only have to be passionate about some structured data that they wish to publish and Exhibit will make publishing that structured data almost as easy as publishing unstructured documents in HTML.

1.1 Motivation

A search on Google for "breakfast cereals" (as of February 2007) turns up as the first hit not a commercial or corporate site, but rather, Topher's Breakfast Cereal Character Guide [9], a homemade site run by a single person. The site has won media honors and recognitions since 1997. But Topher's site, like sites of many early adopters of the Web, still looks and behaves like it was made in 1997, rather than in 2007. While most commercial and institutional sites are now database-backed, serving up sophisticated browsing and visualization user interfaces, homemade sites still consist of small sets of static HTML pages, lacking advanced features that web users have now come to expect. For example, while each cereal character is documented with its brand, the year it came on the market, and the countries in which it was marketed, the web site is organized only by brand so it is impossible to browse by year or by country.

Consider other authors similar to Topher:

- A professor wants to let visitors to his site sort his 97 papers by year or group them by projects, conferences, or co-authors.
- A history high-school teacher wants to showcase 57 important discoveries of bronze age tools to her students through a web site that renders both maps and time lines.
- Some politically conscious citizens have observed unsettling campaign funding patterns and would like the whole world to scrutinize their findings through dynamic plots and charts.

Simply putting up spreadsheets and raw data files will not meet these authors' goals as that requires their audience to download and then explore the data unaided. Much as authors of unstructured content want control over their documents' appearance, authors of structured data want control over their data's presentation—they want to give users the right pathways for structured browsing and rich visualizations. But to support even a basic feature like sorting, they need to create a database, design a schema, design the user interface, write server-side and client-side code, and make sure that their three-tier web *application*—overflowing with technical jargon like SQL, PHP, CSS, and straddling between the web server and many different browsers and platforms—works as expected.[1]

On the early Web, it was easy for enthusiastic but relatively unskilled individuals to stake out territory on the Web by copying other web pages and inserting their own data. These small-time *homesteaders* were crucial contributors to the early growth of the Web. Their homesteads slowly grew in size and complexity, with each homesteader incrementally acquiring specific additional skills as needed to meet their growing ambitions for rich presentation. In contrast, today's hopeful homesteaders of structured content must

[1] Even *database researchers* rarely run database backed web sites for their own collections. In an informal survey conducted by one of the authors at CIDR 2007, only seven out of roughly 140 database researchers claimed to have their publication web sites backed by databases. Even when such a site is database-backed, it might not support any user-controllable browsing features like sorting and filtering.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

pick up a whole alphabet soup of tools before they can begin. The evolution of complex web sites has dramatically raised people's expectations of what web sites ought to offer without giving small-time authors tools to meet such expectations. Consequently, structured browsing and rich presentations are offered mostly by companies and institutions who can afford web site engineering costs, or by the few technically-savvy programmers who have the whole package of web technologies under their belts.

1.2 Approach

We describe Exhibit, a lightweight structured data publishing framework designed to do for structured content what HTML has done for unstructured content: lowering the barrier to publishing while offering a high level of control over presentation.

Exhibit lowers the barrier to publishing by duplicating several key features that enabled the growth of the early Web:

- **No installation, configuration, or maintenance.** Anyone could "join the Web" simply by putting an HTML file on a web server. Similarly with Exhibit, one only need to put an HTML file plus a human-readable data file on a standard web server and their audience immediately experience that data through a rich browsing interface.
- **Copy and paste evolution.** Anyone who wishes to publish data can do so simply by copying someone else's Exhibit files and changing them to suit their needs.
- **Incremental complexity.** Authors can add additional functionality and complexity in small steps, never needing to swallow a whole new set of ideas in one dose. Meanwhile, there are few limits placed on what creative users can do with their material.
- **No network effect required.** Unlike social networking sites, Exhibit provides immediate benefits to its first adopter, regardless of others' actions. It simplifies the author's management of their data, and offers visitors using *existing web browsers* a better interface to that data than can be built by typical web authoring tools with the same effort.

Exhibit offers fine-grained control over presentation by starting with the HTML syntax, inheriting its presentation expressivity, and extending it to let authors specify dynamic generation of presentation.

Exhibit makes each author's job easier and the published data more useful to all of its consumers: readers get more powerful UIs, mashup creators can more easily repurpose the data, and Semantic Web enthusiasts can feed the data to the nascent Semantic Web.

2. RELATED WORK

Exhibit competes with many existing frameworks for publishing structured data in terms of flexibility of presentation and data modeling as well as the required efforts for adoption and usage (Figure 1). Domain-specific services and tools like Flickr and web album generators are restrictive in terms of data modeling as well as presentation. Domain-*generic* services and frameworks like DabbleDB [2], Google Base [3], and the Semantic MediaWiki extension [19] allow for flexible data models but offer little control over the presentation of that data. Any attempt to achieve a bit more control over the presentation immediately requires installation, database administration, and programming.

Fresnel [11] is a language for presenting RDF [7] data, but it only specifies how to transform a graph of structured data into a tree so that tree-based presentation technologies can be used. It does not include any solution for richer browsing, either.

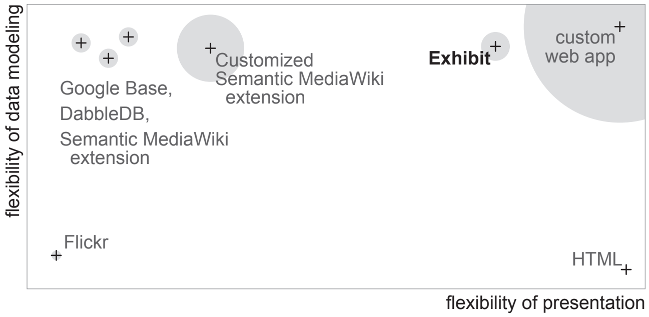


Figure 1. Flexibility of presentation and data modeling as well as the efforts required to adopt and use (circle size) for various publishing frameworks.

From the perspective of a web user rather than a web author, Exhibit is designed to enliven home-made web pages with advanced features such as sorting and filtering, and richer visualizations including time lines and maps. Here, related work includes Chickenfoot [12] and Greasemonkey [5], which let users write scripts to modify web pages being displayed in their browsers to enhance the pages for their personal use; and Sifter [15], which automatically extracts items from a sequence of related pages and offers a faceted browsing interface on those items right within the original web page. While these tools let *users* augment web pages to benefit themselves *individually*, Exhibit targets *authors* whose efforts will benefit *all* users. Compared to users, authors have more expertise about their own data and care more about organizing and presenting it well.

There have also been numerous pieces of research work on faceted browsing [13, 16, 17, 18, 20], many from the Semantic Web community. While Exhibit provides a faceted browsing interface, that is not our contribution in this paper. The interface simply serves to illustrate user interface richness made possible without server reconfiguration when the structured data within a web page is separated from its presentational elements.

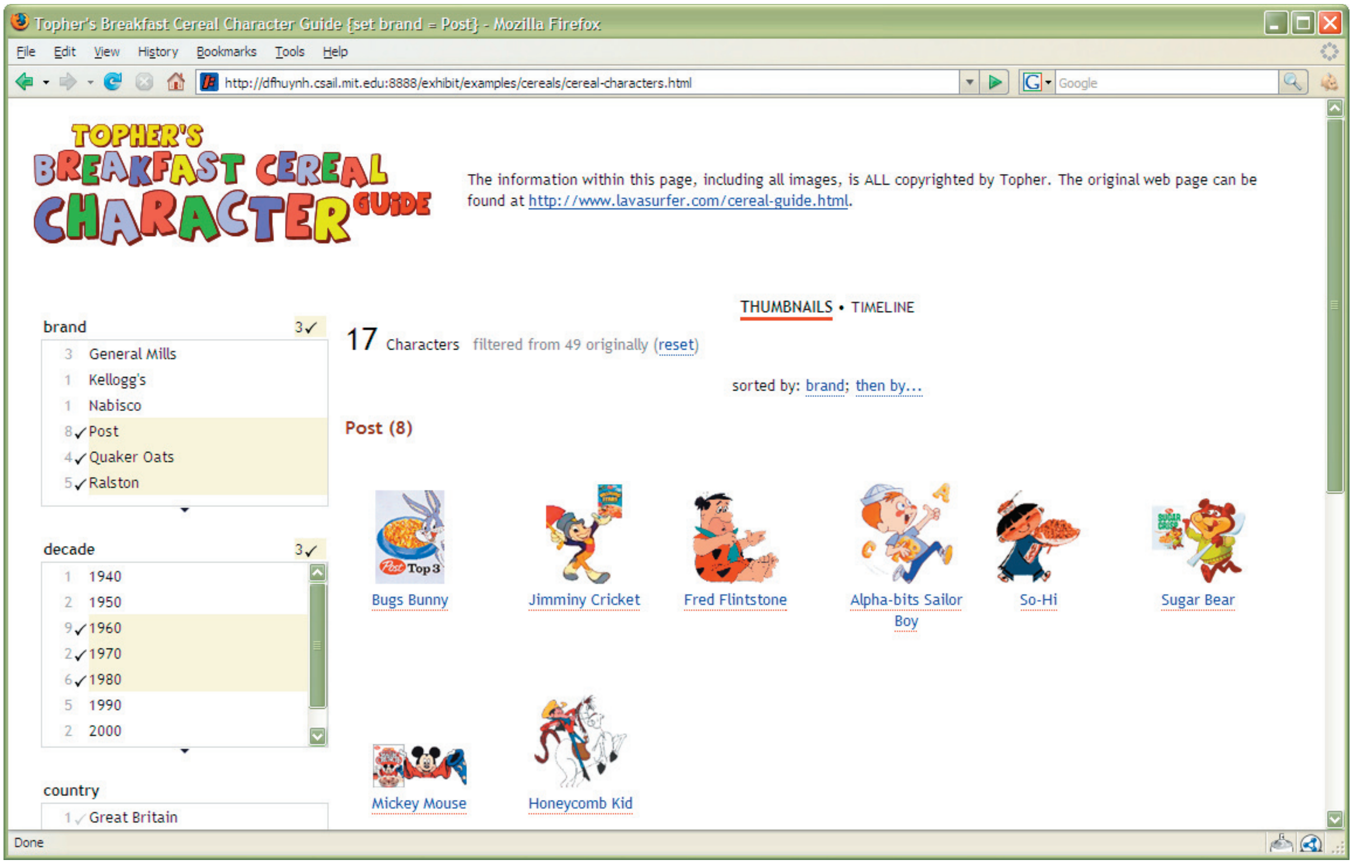
3. INTERFACE DESIGN

As Exhibit is a publishing framework, it has two interfaces: one facing every user of the published information and one facing the author. A web page that embeds Exhibit will be referred to as an *exhibit* in lowercase hereafter.

3.1 User Interface

An exhibit looks just like any other web page, except that it has more advanced features mostly seen on commercial and institutional sites. Figure 2 and Figure 3 show two exhibits covering different types of information. Each is styled differently, but there are several common elements, as described below.

Exhibit's user interface consists of two panels: the browse panel and the view panel, whose locations on the page are controlled by the author. The view panel displays a collection of items in one or more switchable *views*. The exhibit in Figure 2 is configured by the author to support two different views of the same data: THUMBNAILS and TIMELINE. THUMBNAILS is currently selected by the user and is showing. The exhibit in Figure 3 is configured to support six


Topher's Breakfast Cereal Character Guide {set brand = Post} - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://dfhuynh.csail.mit.edu:8888/exhibit/examples/cereals/cereal-characters.html
Google
TOPHER'S BREAKFAST CEREAL CHARACTER GUIDE

The information within this page, including all images, is ALL copyrighted by Topher. The original web page can be found at http://www.lavasurfer.com/cereal-guide.html.

THUMBNAILS • TIMELINE

brand 3✓

3 General Mills
1 Kellogg's
1 Nabisco
8 ✓ Post
4 ✓ Quaker Oats
5 ✓ Ralston

17 Characters filtered from 49 originally (reset)

sorted by: brand; then by...

Post (8)

Bugs Bunny
Jimminy Cricket
Fred Flintstone
Alpha-bits Sailor Boy
So-Hi
Sugar Bear
Mickey Mouse
Honeycomb Kid

decade 3✓

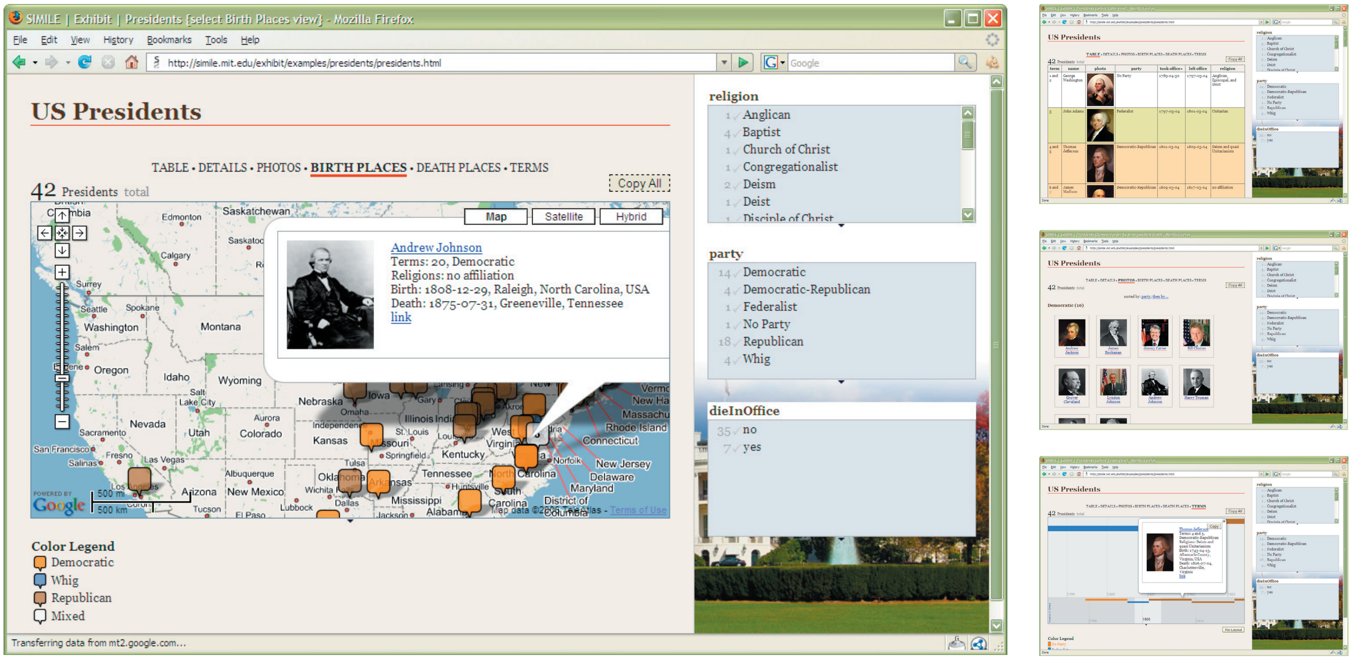
1 1940
2 1950
9 ✓ 1960
2 ✓ 1970
6 ✓ 1980
5 1990
2 2000

country

1 Great Britain

Done

Figure 2. A web page embedding Exhibit to show information about breakfast cereal characters. The information can be viewed as thumbnails or on a timeline and filtered through a faceted browsing interface. Topher, the original author of the information, has eagerly agreed to host this exhibit on his own web site.



US Presidents

TABLE • DETAILS • PHOTOS • BIRTH PLACES • DEATH PLACES • TERMS

42 Presidents total

Copy All

Andrew Johnson
Terms: 20, Democratic
Religions: no affiliation
Birth: 1808-12-29, Raleigh, North Carolina, USA
Death: 1875-07-31, Greeneville, Tennessee
link

Color Legend
Democratic
Whig
Republican
Mixed

religion

1 Anglican
4 Baptist
1 Church of Christ
1 Congregationalist
2 Deism
1 Deist
Disciple of Christ

party

14 Democratic
4 Democratic-Republican
1 Federalist
1 No Party
18 Republican
4 Whig

dieInOffice

35 no
7 yes

Transferring data from mt2.google.com...

Figure 3. A web page embedding Exhibit to show information about U.S. presidents in 6 ways, including maps, table, thumbnails, and timelines.

views, and the BIRTH PLACES view is currently showing. Each kind of view—map, timeline, table, thumbnail, and tile—supports its own configuration settings, almost all of which default to reasonable values. The Google Maps [4] map in the BIRTH PLACES view is configured to color-code its markers by the political parties of the presidents being mapped. Although these settings are specified by the author, some can be changed dynamically by the user (e.g., sorting order in Figure 2).

Items can be presented differently in different views. Where there is little space to render sufficient details in-place (e.g., on a map), markers or links provide affordance for popping up bubbles containing each item's details (e.g., map bubble in Figure 3). The rendition of each item contains a link for bookmarking it individually. Invoking this link later will load the exhibit and pop up a rendition of the bookmarked item automatically.

The browse panel (left in Figure 2 and right in Figure 3) contains facets by which users can filter the items in the view panel. This is a conventional dynamic query interface with preview counts.

The reader is encouraged to visit http://simile.mit.edu/wiki/Exhibit/Examples to try out several live exhibits, particularly those *not* made by the authors of this paper. They range in subjects from books, recipes, and restaurants to sports cars, ancient buildings, space launch sites, breweries, playscripts, and tavern keepers.

3.2 Author Interface

Making an exhibit like Figure 2 involves two tasks: authoring the data and authoring the presentation. Both are iterated until the desired result is achieved. This section briefly describes the publishing process, leaving technical details to later sections.

3.2.1 Creating the Data

Exhibit can read data in its own JSON [6] format. The data file for those breakfast cereal characters looks something like Figure 4. The items' data is coded as an array of objects containing property/value pairs. Values can be strings, numbers, or booleans. If a value is an array, then the corresponding item is considered to have multiple values for that property. For instance, according to Figure 4, the Trix Rabbit character is released in both the U.S. and in Canada. The author is mostly free to make up the names of the properties. We will discuss the data model in section 4.

Data for a single exhibit need not reside in a single file. It can be split into multiple files for convenience. For example, a couple's recipes exhibit can store their data in two files: her-recipes.json and his-recipes.json. The exhibit just needs to load both.

Exhibit data files can be edited in any text editor. We also offer a web service called Babel [1] through which authors can convert various formats to the Exhibit JSON format and even preview the results in a generic exhibit. Babel can currently convert between RDF/XML, N3, Bibtex, Tab Separated Values, Excel files, and Exhibit JSON. Exhibit can also read data directly from the RSS feed of a Google spreadsheet. The reader is encouraged to upload his or her own data to Babel, or cut and paste rows of tab-separated values from a spreadsheet into Babel, to experiment with Exhibit.

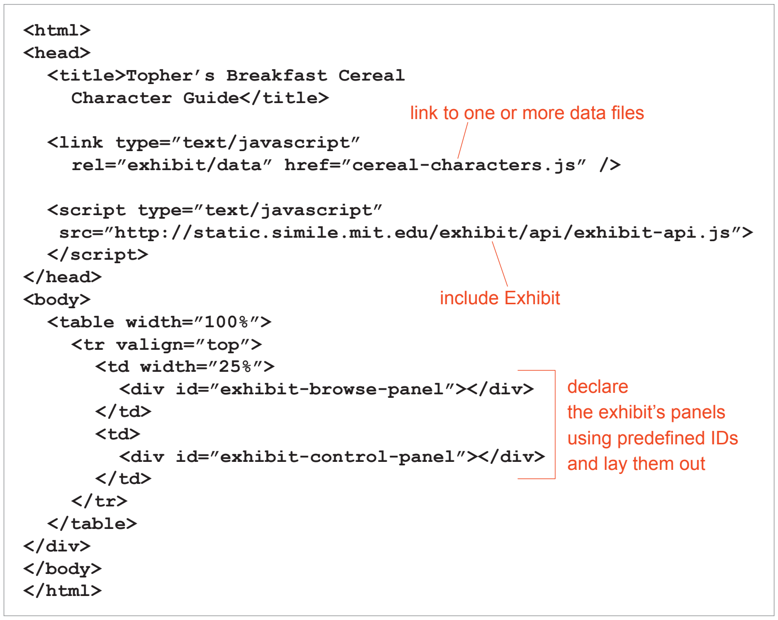
3.2.2 Creating the Presentation

The web page itself is just a regular HTML file that can be created locally, iterated locally until satisfactory, and then, if desired, saved together with the data files on a web server. Figure 5 shows the initial HTML code needed to start making the exhibit in Figure 2. This code instantiates an Exhibit instance, loads it with the data file referenced by the first **`<link>`** element, and configures the exhibit's view panel to show a tile view. The tile view, by default, sorts all items in the exhibit by labels and displays the top ten items using the default *lens*. This lens shows property/value pairs for each item. Many defaults are hardwired into Exhibit to give the author a reasonable outcome with minimal initial work.

The author does not even need to write this initial HTML code from scratch: they can simply copy code from any existing exhibit or from online tutorials. This is how HTML pages are often made—by copying existing pages, removing unwanted parts, and incrementally improving until satisfaction is achieved. The declarative syntax of HTML, the forgiving nature of web browsers and their reasonable defaults, and the quick HTML edit/test cycle make HTML authoring easy. We designed Exhibit to afford the same behavior.

```
{
  items: [
    { type:      ’Character’,
      label:     ’Trix Rabbit’,
      brand:     ’General Mills’,
      decade:    1960,
      country:   [ ’USA’, ’Canada’ ],
      thumbnail: ’images/trix-rabbit-t.png’,
      image:     ’images/trix-rabbit.png’,
      text:      ’First appearing on ...’
    },
    // ... more characters ...
  ]
}
```

Figure 4. An Exhibit JSON data file showing data for one breakfast cereal character, encoded as property/value pairs.



```
<html>
<head>
  <title>Topher’s Breakfast Cereal
    Character Guide</title>

  <link type=”text/javascript”
    rel=”exhibit/data” href=”cereal-characters.js” />

  <script type=”text/javascript”
   src=”http://static.simile.mit.edu/exhibit/api/exhibit-api.js”>
  </script>
</head>
<body>
  <table width=”100%”>
    <tr valign=”top”>
      <td width=”25%”>
        <div id=”exhibit-browse-panel”></div>
      </td>
      <td>
        <div id=”exhibit-control-panel”></div>
      </td>
    </tr>
  </table>
</div>
</body>
</html>
```

link to one or more data files

include Exhibit

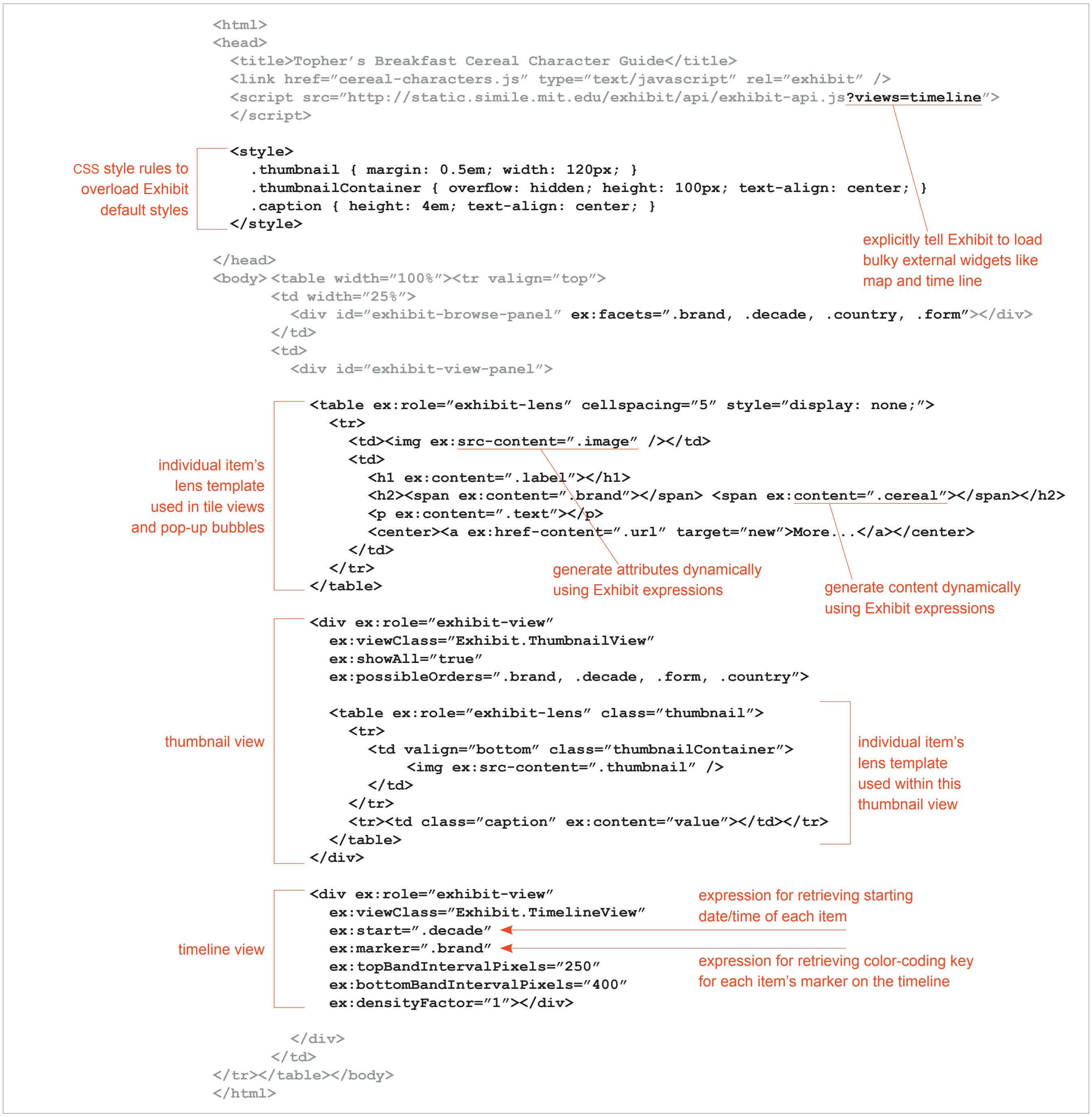
declare the exhibit's panels using predefined IDs and lay them out

Figure 5. To create the web page in Figure 2, the author starts with this boiler plate HTML code, which displays the characters in `cereal-characters.js` through the default lens that lists property/value pairs.

Figure 6 shows the final HTML code needed to render the exhibit in Figure 2 (logo graphics and copyright message omitted). The additional code, in black, configures the facets in the browse panel, the two views in the view panel, one all-purpose lens, and one lens to be used in the THUMBNAILS view.

Making the presentation look better can also involve filling in and fixing up the schema. Figure 7 shows how the plural label for the type **`Character`** is declared so that plural labels in the UI, e.g., 12 Characters, can be generated properly. The **`decade`** property values are declared to be dates instead of strings so that they can be sorted as dates. Changing the schema is just a matter of text editing rather than database administration.

So to not limit advanced authors, Exhibit provides points to escape to Javascript code. Figure 8 shows how to color-code rows of a table view using Javascript.



```
<html>
<head>
  <title>Topher’s Breakfast Cereal Character Guide</title>
  <link href=″cereal-characters.js″ type=″text/javascript″ rel=″exhibit″ />
  <script src=″http://static.simile.mit.edu/exhibit/api/exhibit-api.js?views=timeline″>
  </script>

  <style>
    .thumbnail { margin: 0.5em; width: 120px; }
    .thumbnailContainer { overflow: hidden; height: 100px; text-align: center; }
    .caption { height: 4em; text-align: center; }
  </style>

</head>
<body><table width=″100%″><tr valign=″top″>
      <td width=″25%″>
        <div id=″exhibit-browse-panel″ ex:facets=″.brand, .decade, .country, .form″></div>
      </td>
      <td>
        <div id=″exhibit-view-panel″>

          <table ex:role=″exhibit-lens″ cellspacing=″5″ style=″display: none;″>
            <tr>
              <td><img ex:src-content=″.image″ /></td>
              <td>
                <h1 ex:content=″.label″></h1>
                <h2><span ex:content=″.brand″></span> <span ex:content=″.cereal″></span></h2>
                <p ex:content=″.text″></p>
                <center><a ex:href-content=″.url″ target=″new″>More...</a></center>
              </td>
            </tr>
          </table>

          <div ex:role=″exhibit-view″
            ex:viewClass=″Exhibit.ThumbnailView″
            ex:showAll=″true″
            ex:possibleOrders=″.brand, .decade, .form, .country″>

            <table ex:role=″exhibit-lens″ class=″thumbnail″>
              <tr>
                <td valign=″bottom″ class=″thumbnailContainer″>
                    <img ex:src-content=″.thumbnail″ />
                </td>
              </tr>
              <tr><td class=″caption″ ex:content=″value″></td></tr>
            </table>
          </div>

          <div ex:role=″exhibit-view″
            ex:viewClass=″Exhibit.TimelineView″
            ex:start=″.decade″
            ex:marker=″.brand″
            ex:topBandIntervalPixels=″250″
            ex:bottomBandIntervalPixels=″400″
            ex:densityFactor=″1″></div>

        </div>
      </td>
</tr></table></body>
</html>
```

CSS style rules to overload Exhibit default styles

explicitly tell Exhibit to load bulky external widgets like map and time line

individual item’s lens template used in tile views and pop-up bubbles

generate attributes dynamically using Exhibit expressions

generate content dynamically using Exhibit expressions

thumbnail view

individual item’s lens template used within this thumbnail view

timeline view

expression for retrieving starting date/time of each item

expression for retrieving color-coding key for each item’s marker on the timeline

Figure 6. The author starts with the `code in gray` (from Figure 5), includes more and more of the `code in black`, and tweaks until the desired result (Figure 2) is achieved (logo graphics and copyright omitted). Tweaking involves following online documentation or just copying code from other existing exhibits.

4. DATA MODEL

The data model of each exhibit is a set of *items* in which each item has a *type* and several *properties*. A lot of exhibits are flat, but some contain items that reference one another. In those exhibits, the data models are graphs.

4.1 Items

Each item is guaranteed to have a textual label, which the author is required to specify as the item's **`label`** property value. This requirement ensures that Exhibit always knows how to present the item wherever its textual description is needed.

The label of an item is also used as the item's default ID—to identify that item uniquely within the exhibit. If two items need to have the same label, then the author must explicitly assign them different IDs by explicitly specifying their **`id`** property values.

An item's ID is also used to generate its URI by prepending the exhibit's URL to the ID. Essentially, each exhibit creates its own namespace. If a particular item is intended to describe some resource with an existing URI, that URI can be explicitly specified as the item's **`uri`** property value.

4.2 Types

Each item has a type, which can be specified by the author as the item's **`type`** property value, or if missing, defaults to the generic type **`Item`**.

Just like items, types are required to have textual labels and can also be explicitly given IDs and URIs. In addition to **`id`**, **`uri`**, and **`label`**, a type has one more field,[2] **`pluralLabel`**, which is used to generate more grammatical user interface text (e.g., 9 People instead of 9 Person). If **`pluralLabel`** is not explicitly declared, its value is the same as the type's label.

In fact, there is no need for the author to explicitly declare every type in the schema. A type is added to the system whenever an item of that type is added. This lets the author focus on the items—the main point of her exhibit—and only talk about types and properties when they make the user interface better.

4.3 Properties

Properties are also required to have textual labels and can optionally be explicitly given IDs and URIs. In addition to **`id`**, **`uri`**, and **`label`**, a property also has other fields: **`reverseLabel`**, **`pluralLabel`**, **`reversePluralLabel`**, **`groupingLabel`**, and **`reverseGroupingLabel`**. These additional labels are used to generate more user-friendly UI text, e.g., child of rather than reverse of parent of.

Each property can also be assigned a *value type* as its **`valueType`** field value. This specifies how all values of that property should be interpreted. For example, the **`age`** property should be assigned the value type **`number`** so that Exhibit will interpret all **`age`** property values as numbers. There is a fixed set of value types: **`text`**, **`number`**, **`date`**, **`boolean`**, **`url`**, and **`item`**. Figure 7 shows how property value types are declared. All values of value type **`item`** are interpreted as locally unique IDs of items in the exhibit.

This design choice of tying value types to properties, rather than requiring value type declaration on each property value, facilitates incremental improvements to the data. For instance, **`brand`** property values in Figure 4 are initially, by default, of value type **`text`**. This might satisfy the author until she wants to record details about the brands, at which time she can simply specify the value type for **`brand`** to be **`item`** and add the data for the brands (Figure 9).

```
{
  types: {
    ‘Character’: {
      pluralLabel: ‘Characters’
    }
  },
  properties: {
    ‘url’: {
      valueType: “url”
    },
    ‘decade’: {
      valueType: “date”
    }
  }
  items: [
    // ... items ...
  ]
}
```

Figure 7. Schema information can be added to the JSON file to improve Exhibit's user interface.

The Exhibit abstract data model is essentially the RDF [7] abstract data model except that property values cannot be assigned value types individually. In this way, the Exhibit data model is a sub-model of the RDF data model, less powerful but sufficient for many simple use cases. This simplicity allows for Exhibit's simple JSON format.

5. EXPRESSIONS

Exhibit provides an expression language for selecting data to display in lenses and views. An Exhibit expression consists of a sequence of one or more property IDs, each preceded by a *hop operator*. In RDF terminology, the **`.`** hop operator traverses from subject to object while the **`!`** hop operator traverses from object to subject. For example,

- evaluating **`.hasAuthor.teachesAt.locatedIn`** on some papers returns the locations of the schools where the authors of those papers teach;
- evaluating **`.hasSpouse!hasChild`** on some people returns their parents-in-law;
- evaluating **`!shot!arrested`** on John F. Kennedy returns the police officers who arrested his assassinator.

Figure 8 shows a few expressions being used to specify columns of a tabular view.

Currently, this expression language can only traverse the data model, and in that way, it resembles the Fresnel Selector Language [11]. In the future, we plan to extend the language to support computations so that, for example, the author can specify

`.lastName + “, ” + .firstName` and
`yearOf(now) - yearOf(.birthDate)`

as columns in a tabular view.

This expression language is currently sufficient for configuring various parts of Exhibit. In the future, a SPARQL [8] interface might be added.

[2] We use "fields" to mean properties of types and properties, and we use "properties" to mean properties of items.

```
<div ex:role=”exhibit-view”
    ex:viewClass=”Exhibit.TabularView”
    ex:label=”Table”
    ex:columns       = “.label,  .imageURL, .party, .presidency.inDate, .presidency.outDate”
    ex:columnLabels  = “name,    photo,     party,  took office,         left office”
    ex:columnFormats = “list,    image,     list,   list,                list”
    ex:sortColumn=”4”
    ex:sortAscending=”true”
    ex:rowStyler=”rowStyler”
    ></div>
```

custom column format

custom column heading label

column of indirect property

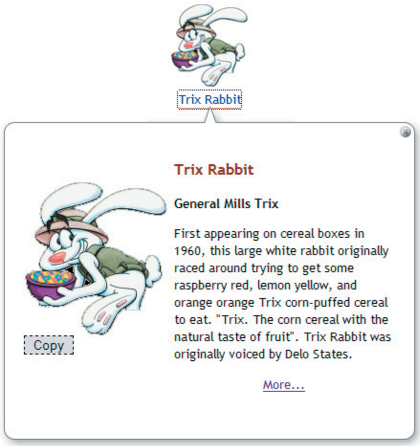
```
var rowStyler = function(item, database, tr) {
  var party = database.getObject(item, “party”);
  var color = “white”;
  switch (party) {
  case “Democratic”: color = “blue”; break;
  case “Republican”: color = “red”; break;
  }
  tr.style.background = color;
}
```

Figure 8. Advanced configuration of a tabular view. Mixing HTML and Javascript yields the best of both worlds: familiarity, simplicity, and forgiving nature of HTML’s declarative syntax to start, and expressiveness of Javascript’s imperative syntax whenever needed.

```
{
  properties: {
    ‘brand’: {
      valueType: ‘item’
    }
  },
  items: [
    { type:      ‘Character’,
      label:     ‘Trix Rabbit’,
      brand:     ‘General Mills’,
      decade:    1960,
      country:   [ ‘USA’, ‘Canada’ ],
      thumbnail: ‘images/trix-rabbit.png’,
      text:      ‘First appearing on ...’
    },
    // ... more characters ...

    { type:        ‘Brand’,
      label:       ‘General Mills’,
      headQuarter: ‘Minnesota’,
      founded:     1856
    }
    // ... more brands ...
  ]
}
```

Figure 9. Elaboration of `brand` property values by specifying the value type and adding `Brand` items.



Trix Rabbit

General Mills Trix

First appearing on cereal boxes in 1960, this large white rabbit originally raced around trying to get some raspberry red, lemon yellow, and orange orange Trix corn-puffed cereal to eat. "Trix. The corn cereal with the natural taste of fruit". Trix Rabbit was originally voiced by Delo States.

```
<table ex:role=”exhibit-lens” cellspacing=”5”
    style=”display: none;”>
  <tr>
    <td>
      <img ex:if-exists=”.image”
          ex:src-content=”.image” />
      <div ex:control=”copy-button”></div>
    </td>
    <td>
      <h1 ex:content=”.label”></h1>
      <h2>
        <span ex:content=”.brand”></span>
        <span ex:content=”.cereal”></span>
      </h2>
      <p ex:content=”.text”></p>
      <center ex:if-exists=”.url”>
        <a ex:href-content=”.url”
          target=”new”>More...</a>
      </center>
    </td>
  </tr>
</table>
```

Figure 10. Lens template for showing a breakfast cereal character in a pop-up bubble.

6. LENSES AND VIEWS

In Exhibit, lenses and views are used to display data. An Exhibit lens renders one single item while an Exhibit view renders a set of items, possibly by composing several lenses in some layout. Exhibit comes with several views: tile view, thumbnail view, tabular view, time line view, and map view. Figure 3 shows a few of these views. Third parties can implement more views and authors can use them simply by linking in their Javascript code. Each view has its own settings, which vary based on the complexity of the view. For example, the map view needs an expression that specifies the latitude/longitude pair for each item and an expression that retrieves some property which can be used to color code the map markers; the tile view, on the other hand, needs to know how to sort the items.

While views come pre-built (but are configurable), lenses can be written by coding lens *templates*. A template is just a fragment of HTML that can be specified in-line, as in Figure 10.

Within a lens template, the `content` attribute of an element specifies what content to stuff into that element when the template is instantiated for an item. For example, in Figure 10, the **<h1>** element will be filled with the **label** property value of the item.

Attributes that end with **-content** are assumed to contain Exhibit expressions. These expressions are resolved into actual values when the lens is instantiated, and the values are used to assert HTML attributes of the same name but without the **ex:** namespace and the **-content** suffix. For example, the **ex:src-content** attribute in Figure 10 is replaced with the **image** property value of the item being rendered, generating the attribute **src="images/trix-rabbit.png"** in the generated HTML **<img>** element.

The **if-exists** attribute of an element determines whether that element and its children in the template should be included in the presentation of a particular item. For example, if an item does not have an **image** property value, the template in Figure 10 will not generate a broken image.

The **control** attribute specifies which Exhibit-specific control to embed. There are only two controls supported at the moment: the **item-link** control is a permanent link to the item being rendered and the **copy-button** control is a dropdown menu button that lets the user copy the item's data off in various formats.

Note that Exhibit adopts a *template-based* approach to generating lenses. In contrast, Fresnel [11] takes a *rule-based* approach in which styling rules are declared separately and then applied to morph a generic presentation into a custom presentation.

7. EXPORTERS

The Copy button in Figure 10 and the Copy All button in Figure 3 pop up menus that let the user pick a format in which to export one or more items. Exhibit currently provides exporters for RDF/XML, Exhibit JSON, Semantic MediaWiki extension wikitext [19], and Bibtex. The author can register third parties' exporters with Exhibit so that they show up in these menus.

The purpose of these exporters is to facilitate and encourage propagation of structured data by offering convenience to both users and authors. For example, being able to copy off the Bibtex of some publications that you have found in an exhibit so that you can cite them is very convenient. You can also copy that data in Exhibit JSON format and incorporate it into your own exhibit to make an archive of related work.

Exhibit's default exporters generate an **origin** property value for each item to export. This value is the URL that, when invoked, returns to the original exhibit and pops up the view of that item automatically. This is a lightweight mechanism for attribution.

8. IMPLEMENTATION

The Exhibit framework is implemented in several Javascript, CSS, and image files. It is available at a public URL where anyone can reference it from within his or her HTML pages. Exhibit authors do not need to download any software, and exhibit viewers do not need to install any browser extension. This zero cost is the signature of client-side include Web APIs and is largely responsible for the explosion in the use of Google Maps for mashups.

Exhibit's source code is available publicly. Any of its parts can be overridden by writing more Javascript code and CSS definitions after including Exhibit's code. Third parties can implement additional views and exporters to supplement our library.

Exhibit's architecture is illustrated in Figure 11. At the bottom is a database implemented in Javascript. The database and the expression facilities form the data layer on which much of the rest of the system depends.

There are several points of extensibility. More views can be added. More exporters can be registered. The Browse Panel can also be extended with facets that don't just list their choices but specialize for the properties that they are configured with, such as showing a calendar for a date property.

The localization component encapsulates localized UI resources, including text strings, images, styles, and even layouts. This is only our early attempt—internationalizing a framework that generates user interfaces at run-time is very difficult. We note that even HTML is biased for English. For example, bold and italics, which have native syntax in HTML, are foreign concepts to most Asian scripts.

Exhibit's database and its DOM generating code are the two main bottlenecks for Exhibit's performance scalability at the moment. Currently an exhibit can handle up to a few hundred items while remaining responsive. However, we expect that our own implementation can be optimized further, that browsers' Javascript engines will get faster, and that computer hardware will also speed up in time. Furthermore, web browsers might expose local database storage APIs to web pages in the future, which can replace our Javascript-based implementation.

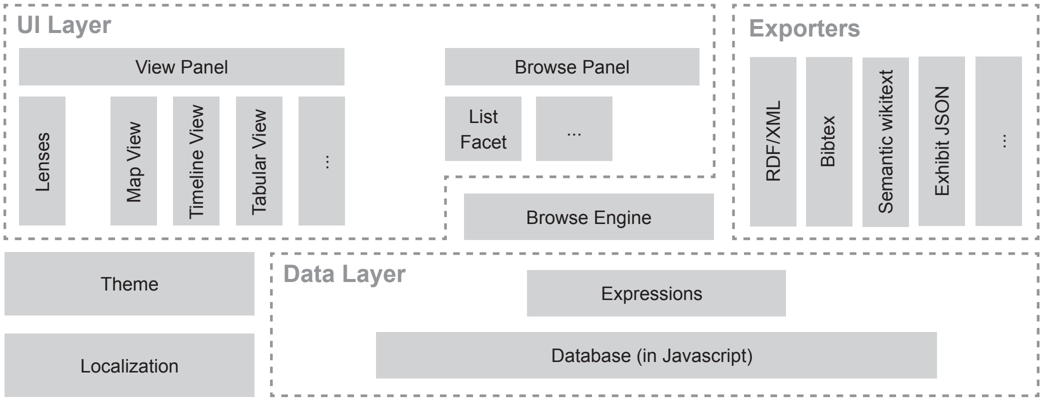


Figure 11. Exhibit's architecture.

9. DISCUSSION

The incentive for each author to use Exhibit is the ease of publishing structured data and the resulting sophisticated user interfaces. While this motivation is personal and the immediately perceptible benefit is local, there are far reaching effects as a result of the published data being entirely, publicly accessible in structured form. We now discuss how Exhibit, by design, is related to mashups, the Semantic Web, and the *Long Tail* of information domains.

9.1 An Ecology of Mashup Data

Although we have pitched Exhibit primarily to potential authors of structured data, we note that it also benefits all kinds of consumer of that data. Having the structured data readily available lets mashup creators focus on the fun of making mashups rather than on the labor of scraping data. Making mashups can also mean making new exhibits from data found on existing exhibits, and making exhibits is much easier than building web applications. The data from Exhibit-based mashups is also readily reusable for even more mashups. In contrast, the data in traditional mashups is liberated from the original sites only to be locked up again behind web servers on the mashup sites.

While mashup is often the act of repurposing existing data for new, public uses, this definition can be extended to include personal uses as well. Browser-based tools such as Piggy Bank [14] have illustrated how users can get more utility out of existing data by scraping it out of web pages, combining data from different sites, or combining public data with private data. These tools are currently not widely accepted because they either require tedious hand-written screen scrapers (as in Piggy Bank) or rely on fragile web data extraction algorithms (as in Sifter [15]). They can be made to consume exhibits' data if exhibits proliferate.

In other words, Exhibit makes presentation cheaper for authors, and reusable data cheaper for everyone else.

9.2 Exhibit as A Semantic Web Application

It is now time to admit that we see Exhibit as a *Semantic Web application*. The Semantic Web has been dismissed by many as an unrealistic vision because it is often pitched as a medium wherein automated software agents harvest structured data to accomplish complex tasks on behalf of their users. But as our tools hopefully demonstrate, even without automated agents there is tremendous value to standardizing a model of structured data involving objects with properties and values. Such standardization offers a clean way to separate content from presentation, supporting easy repurposing of content in multiple presentations and easy creation of complex presentations for an infinite variety of content.

Some of the Semantic Web faithful might dismiss our contention that Exhibit is a Semantic Web application. Where is the RDF? Where are the OWL ontologies? How will all these Exhibit authors' half-baked ontologies and duplicated instances be aligned? In response, we observe, first, that the RDF is in plain sight: it is the model being instantiated by each exhibit. We have simply chosen to offer a different syntax—other than the standard RDF/XML and more recent N3—for expressing that model. Granted, our syntax cannot express arbitrary RDF. However, we have accepted this limitation in pursuit of a simpler syntax that can be authored by less-skilled users. As for OWL and ontology alignment, we argue that worrying about these now is premature and counterproductive. For the time being, these can be left to humans, who by using Exhibit will still be far ahead of the current state thanks to a unified data syntax that does not need to be scraped. The critical first step for the Semantic Web is to generate lots of data, and Exhibit is a tool that creates incentives for authors to make this happen. Rationalization of the data can come later. The early Web was an anarchy, only later tamed by directories and search engines. Similarly, the early Semantic Web is likely to be a chaotically structured entity whose existence can motivate the development of the sophisticated alignment and reasoning tools that will bring some order to it.

Even without solving ontology alignment, the data harvested from these exhibits can already improve web searching and browsing. For example, one could search for data having a particular Exhibit type, e.g., "Character", and a particular property, e.g., "brand." The search engine might confound breakfast cereal characters with action hero characters, but not with movie characters who have no "brand." These extra query criteria help reduce the search space.

9.3 The Long Tail of Information Domains

Indeed, millions of ontologies will arise whether or not they need to be aligned. Like the Web, the Semantic Web should let anybody say anything about anything, and there are simply a lot of people with a lot to say about many things. This diversity should be cherished.

Diversity already brings big profits to online retailers. On Amazon, there are hundreds of thousands of unpopular books, each selling only a few to a few hundred copies, whose combined profit rivals that of the few hundred books that sell millions of copies. This observation is explored in "The Long Tail" [10].

If we rank the various kinds of information that people want to publish on the Web by their popularity, we should also get a long tail (Figure 12): a few dozens of readily recalled kinds of information such as consumer products, news articles, events, locations, photos, videos, music, and software projects populate the massive head of the curve while hundreds of thousands of unpopular topics, such as sugar packet collection and lock picking, spread thinly over the long tail. Clustering search engines such as clusty.com show dozens of sites of each unpopular topic. Wikipedia's several thousand categories and DMoz's half a million categories show how long the tail is.

It is much more appealing to target the head where there are more resources, more rewards, a larger target audience, and a greater sense of urgency. Consequently, too many Semantic Web projects are aimed at displacing specific existing solutions for information domains at the head, such as building Semantic Web-enabled web applications for photos and publications. These young Semantic Web technologies must compete against decades old solutions that have been heavily invested, well understood, long refined, and broadly deployed. A Semantic Web-backed web site looks just like any other web site, except that it usually suffers from poor scalability. The lack of supporting expertise, solid infrastructures, and mature programming paradigms leaves Semantic Web solutions unpolished. Where the old solutions fail in lack of generality, there has not been sufficient user interface research to make Semantic Web principles shine.

To compete on any one particular information domain at the head is to ensure defeat. It is comparable to trying to kickstart the Web by first attempting to displace large publishing houses.

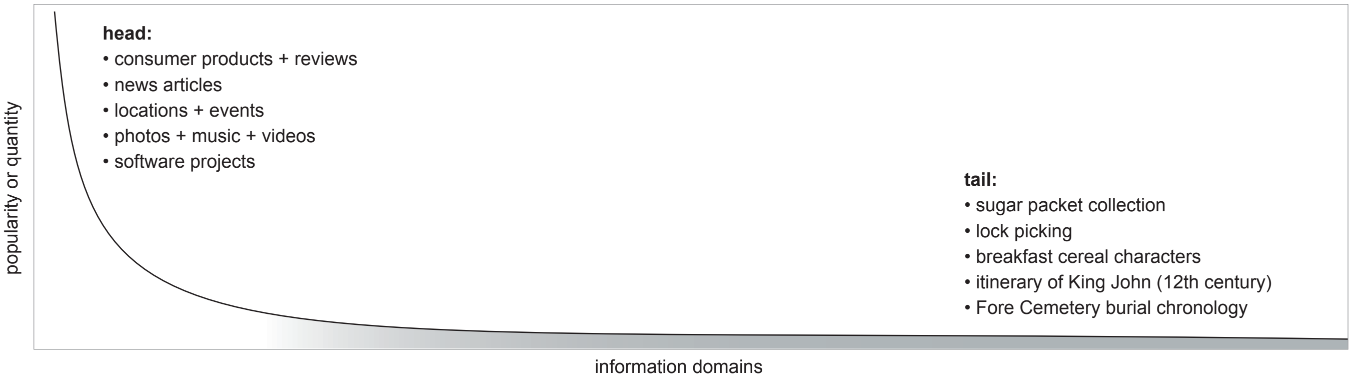


Figure 12. The Long Tail of Information Domains.

Exhibit is an attempt to embrace the information at the tail where there are few established solutions but many opportunities to diversify the Semantic Web. Two months after Exhibit was released, we have found new exhibits about sports cars, ancient buildings, space launch sites, breweries, playscripts, tavern keepers, and more. Note that it would be foolish to build custom solution for any one of these domains because the target users in that one domain are few and the amount of data is little. It is rather in the aggregate of all these domains that lies the huge amount of structured data to fuel the Semantic Web.

The insight into the long tail of information domains gives context to our previous efforts. Piggy Bank [14] and Sifter [15] target *users* at the *head* of the distribution, where the information comes from large publishers and is relatively well formatted by server-side technologies. By scraping that information to provide more values to users, we hope to "trick" users into converting information at the head into Semantic Web format. At the *tail*, where the existing information is too unstructured to scrape reliably, Exhibit enrolls *authors* into converting that information into more structured form. Thus, we have the whole distribution curve covered.

10. CONCLUSION

In this paper, we presented Exhibit, a framework that lets small-time authors publish structured data in sophisticated user interfaces at very low cost. Their efforts, personally motivated and personally rewarding, may yield a large, diverse repertoire of entirely, publicly accessible structured data, ready to fuel the nascent Semantic Web.

If Exhibit achieves wide adoption, we will study the structured data medium that it creates. In particular, we will explore ways for users to mash up data from several exhibits and perform ontology alignment on the fly to achieve value from such aggregation.

11. ACKNOWLEDGMENTS

We are grateful to the members of the Haystack group, the User Interface Design group, the Simile team, and the W3C at MIT CSAIL for their insightful comments on our work.

This work was supported by the National Science Foundation (under award number IIS-0447800), by the Biomedial Informatics Research Network (NIH NCRR mBIRN U24-RR021382), and by Nokia (AGMT. DTD. 9/2/05). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funders.

REFERENCES

[1] Babel. http://simile.mit.edu/babel/.
[2] DabbleDB. http://dabbledb.com/.
[3] Google Base. http://base.google.com/.
[4] Google Maps. http://maps.google.com/.
[5] Greasemonkey. http://greasemonkey.mozdev.org/.
[6] Introducing JSON. http://www.json.org/.
[7] Resource Description Framework (RDF). http://www.w3.org/RDF/.
[8] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
[9] Topher's Breakfast Cereal Character Guide. http://www.lavasurfer.com/cereal-guide.html.
[10] Anderson, C. The Long Tail: Why the Future of Business is Selling Less of More. New York: Hyperion, 2006.
[11] Bizer, C., E. Pietriga, D. Karger, R. Lee. Fresnel: A Browser-Independent Presentation Vocabulary for RDF. ISWC 2006.
[12] Bolin, M., M. Webber, P. Rha, T. Wilson, and R. Miller. Automation and Customization of Rendered Web Pages. UIST 2005.
[13] Hildebrand, M., J. van Ossenbruggen, L. Hardman. /facet: A Browser for Heterogeneous Semantic Web Repositories. ISWC 2006.
[14] Huynh, D., S. Mazzocchi, and D. Karger. Piggy Bank: Experience the Semantic Web Inside Your Web Browser. ISWC 2005.
[15] Huynh, D., R. Miller, and D. Karger. Enabling Web Browser to Augment Web Sites' Filtering and Sorting Functionality. UIST 2006.
[16] Oren, E., R. Delbru, S. Decker. Extending faceted navigation for RDF data. ISWC 2006.
[17] schraefel, m. c., Karam, M. and Zhao, S. mSpace: interaction design for user-determined, adaptable domain exploration in hypermedia. AH 2003: Workshop on Adaptive Hypermedia and Adaptive Web Based Systems.
[18] Sinha, V. and D. Karger. Magnet: Supporting Navigation in Semistructured Data Environments. SIGMOD 2005.
[19] Völkel, M., M. Krötzsch, D. Vrandecic, H. Haller, R. Studer. Semantic Wikipedia. WWW 2006.
[20] Yee, P., K. Swearingen, K. Li, and M. Hearst. Faceted Metadata for Image Search and Browsing. CHI 2003.