3. Publishing Data

When it came into existence, the Web was hailed for giving individuals the same publishing power as large publishers. But over time, large publishers learned to exploit the structure in their data, leveraging databases and server-side technologies to provide rich browsing and visualization features. People have come to expect from professional sites features like searching, sorting, filtering, comparison, maps, etc. Individual *small* publishers fall behind once more: neither old-fashioned static pages nor domain-specific publishing frameworks (e.g., web photo album generators) and services (e.g., Flickr) supporting limited customization can match full-fledged database-backed web applications that cost thousands of dollars. Fortunately, ideas embodied in tools that have made publishing so efficient for large publishers can also help small publishers as long as the needs and abilities of small publishers are taken into consideration. This chapter explains how.

A typical application consists of three layers: data, presentation, and application logic in between. Separating data from presentation allows mixing and matching data with presentation so that the same data can be shown in different ways. It is also easier to build tools specialized for dealing with either data (e.g., databases) or presentation (e.g., form editors) separately.

This principle of separating data from presentation has been built into technologies that target large publishers. Designed for scalability and flexibility, these technologies are far too complex for casual users to adopt. Such users have so far only been offered HTML as the generic web publishing technology, and in HTML, data and presentation are mixed together.

To let casual users benefit from the separation of data from presentation, the costs of authoring data, authoring presentation, and connecting them up must be lowered.

The cost of authoring presentation can be lowered in two ways by assuming that the presentation needs of casual users are not so complex:

• First, a variety of common features such as sorting, grouping, searching, filtering, map visualization, timeline visualization, etc. can be provided out-of-the-box so that each casual user need not re-implement them herself. (As her presentation needs get more sophisticated and unique, she will eventually need to program her own features.)

• Second, customization to the presentation can be specified in an HTML-based syntax right inside the HTML code used to layout the web page. This is so that the publisher can work on every part of the page's presentation inside a single file in a single syntax. (As the publisher's needs to customize the presentation get more sophisticated, the HTML-based syntax will no longer satisfy her.)

The cost of authoring data can be lowered in two ways by assuming that casual users publish only small data sets:

• First, if a publisher is already editing her data in some particular format and some particular editor convenient to her, she should not need to load that data into a database and then maintain the data through the unfamiliar and inconvenient user interface of the database. She should be able to keep managing her data however she likes, and the data only gets loaded into the database when it needs to be rendered. This is realizable if the data is small and loading it into the database is quick. (As the data gets larger, there is a point when the data should already be loaded into a database to ensure a responsive web site.)

• Second, data schemas can be made optional if their benefits do not justify their costs. While schemas are advantageous on large data sets for database optimizations and for managing data at a higher level of abstraction, their benefits on small data sets are much less apparent. (As the data gets larger or more complex, there is a point when schema abstractions benefit both the publisher as well as the publishing engine.)

Finally, all the costs of setting up software (database, web server, and application server) can be eliminated if the software is packaged as a Web API to be included into a web page on-the-fly. Such a Web API can also easily allow for extensions, which accommodate the increasing sophistication in a casual user's publishing needs.

These ideas have been built into the Exhibit lightweight data publishing framework, packaged as a Web API. This chapter will discuss the design of Exhibit's user and publisher interfaces in section 1, its data model in section 2, and its user interface model in section 3. Section 4 briefly describes Exhibit's implementation. Finally, section 5 reports real-world usage of Exhibit and discusses its impact.

3.1 Interface Design

As Exhibit is a publishing framework, it has two interfaces: one facing publishers and one facing users of the published information. A web page published using Exhibit will be referred to as an *exhibit* in lowercase.

3.1.1 User Interface

An exhibit looks just like any other web page, except that it has advanced features mostly seen on commercial and institutional sites. Figure 3.1 and Figure 3.2 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 mainly of *views* and *facets*, whose locations on the page are controlled by the publisher. The exhibit in Figure 3.1 is configured by the publisher to support two different views of the same data: THUMBNAILS and TIMELINE. THUMBNAILS is currently selected by the user and it is showing. The exhibit in Figure 3.2 is configured to support six views, and the BIRTH PLACES view is currently showing. Each kind of view—map, timeline, table, thumbnail, tile, scatter plot, bar chart, etc.—supports its own configuration settings. The Google Maps [11] map in the BIRTH PLACES view is configured to embed the presidents' portraits in the map marker, and the TERMS timeline view is configured to color-code the presidents' terms by their political parties. Although these settings are specified by the publisher, some can be changed dynamically by the user (e.g., sorting order in Figure 3.1). Exhibit's user interface can be extended by third-parties' views if the publisher chooses to include them.

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 3.2). 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 facets (left in Figure 3.1 and right in Figure 3.2) let users filter the currently displayed items. This is a conventional dynamic query interface with preview counts.

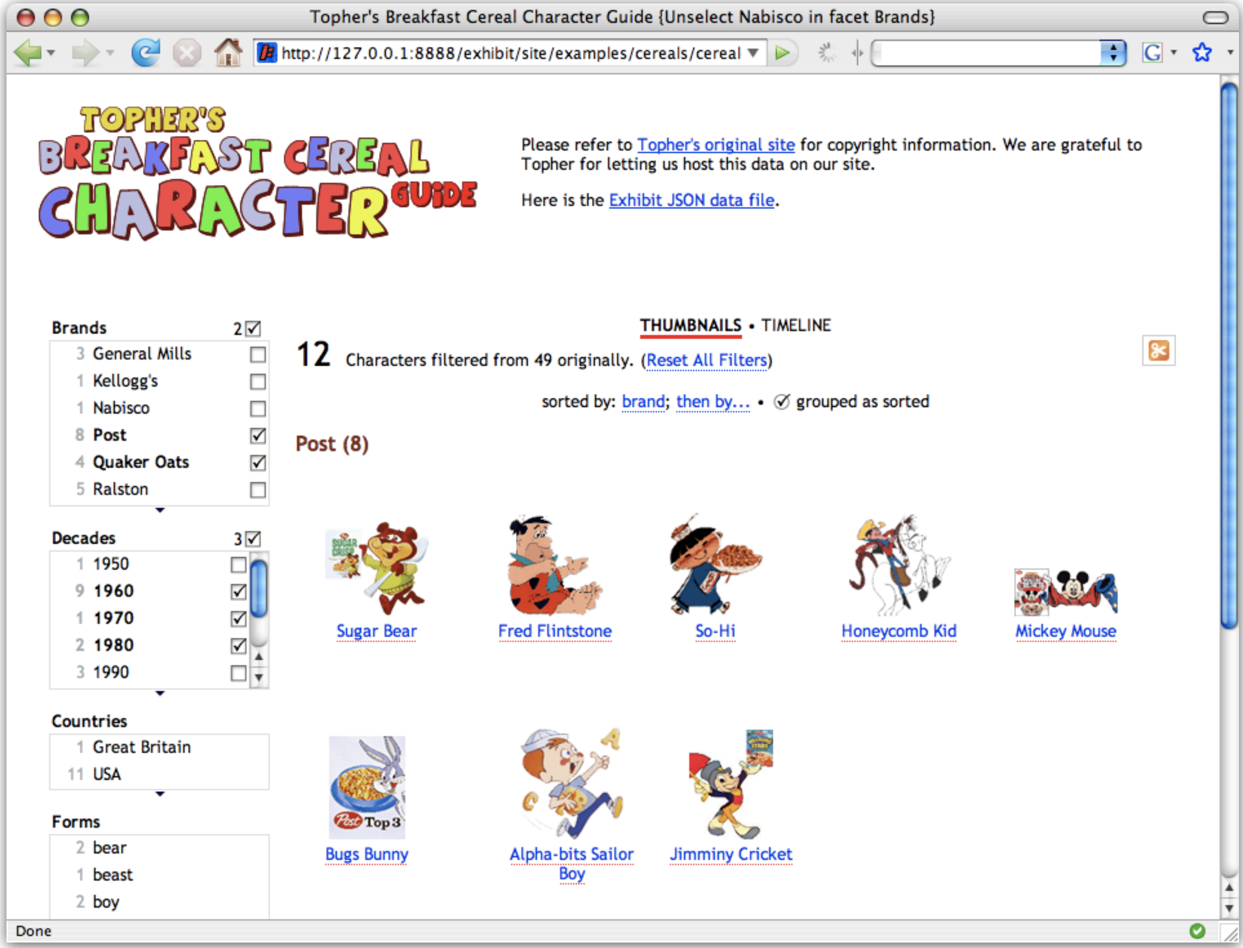


Figure 3.1. A web page embedding Exhibit to show information about breakfast cereal characters [36]. The information can be viewed as thumbnails or on a timeline and filtered through a faceted browsing interface.

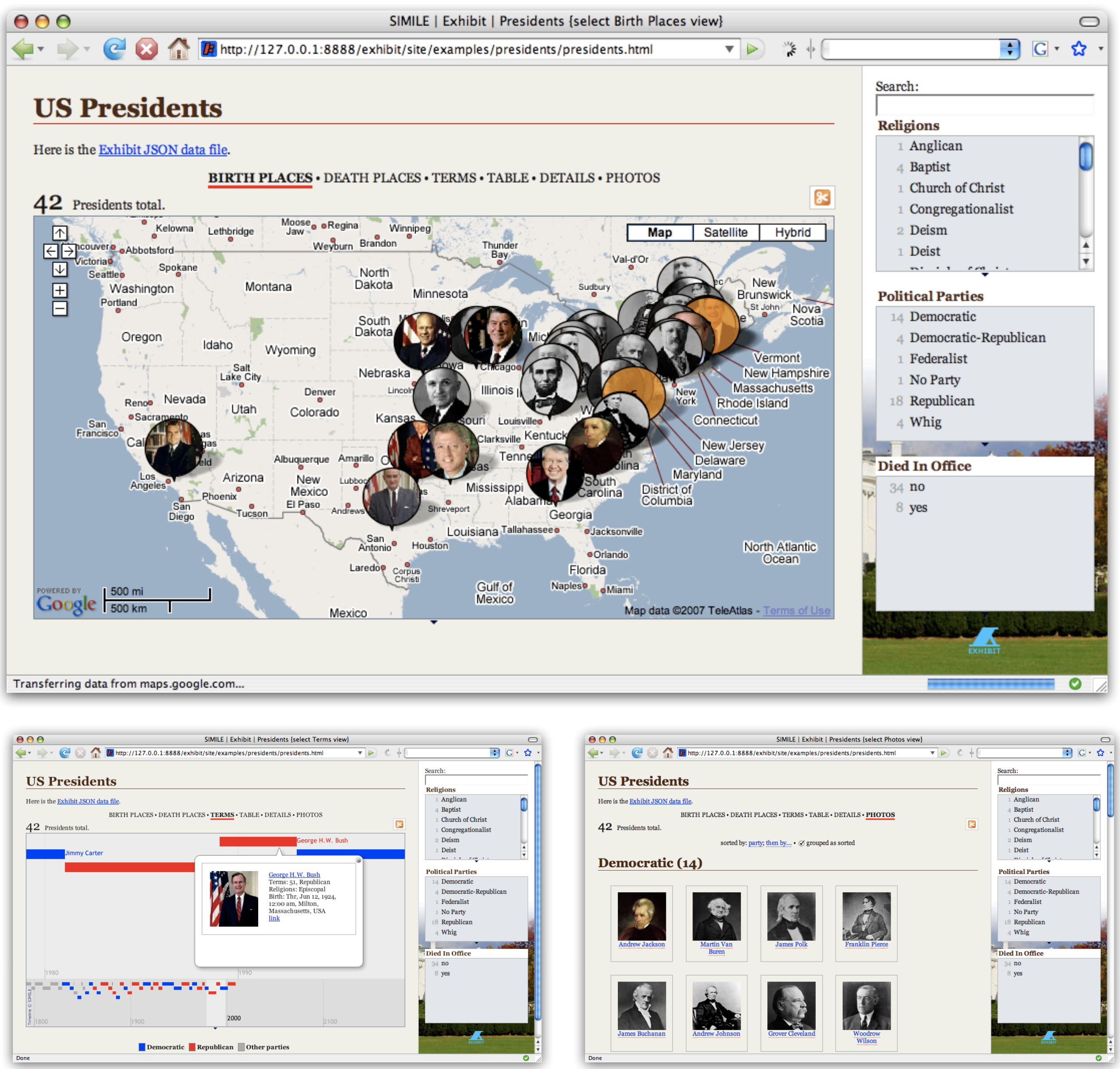


Figure 3.2. A web page embedding Exhibit to show information about U.S. presidents in 6 ways, including maps, table, thumbnails, and timelines.

3.1.2 Publisher Interface

Making an exhibit like Figure 3.1 involves two tasks: creating the data and creating the presentation. Both are iterated until the desired result is achieved. This section briefly describes the publishing process, leaving technical details to later sections.

3.1.2.1 Creating the Data

Exhibit supports its own JSON [13] format natively but can automatically import a variety of formats including Bibtex, Excel, RDF/XML, N3, and Google Spreadsheets feeds. The data file for those breakfast cereal characters looks something like that in Figure 3.3. 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 3.3, the Trix Rabbit character is released in both the US and in Canada. The publisher is mostly free to make up the names of the properties. We will discuss the specifics of the data model subsequently.

Data for a single exhibit needs not reside in a single file. It can be split into multiple files for convenience. For example, a couple's recipes exhibit can pool its data from two separate files: `her-recipes.json` and `his-recipes.json`.

3.1.2.2 Creating the Presentation

The web page itself is just a regular HTML file that can be created locally, iterated locally until satisfaction, and then, if desired, uploaded together with the data files to the web server. Figure 3.4 shows the initial HTML code needed to start making the exhibit in Figure 3.1. This code instantiates an Exhibit instance, loads it with the data file referenced by the first `<link>` element, and specifies where to embed a view panel, which shows a tile view by default. Also by default, the tile view 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. Reasonable defaults are hardwired into Exhibit to give the publisher some result with minimal initial work.

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

Figure 3.3. An Exhibit JSON data file showing data for one breakfast cereal character, which is encoded as property/value pairs.

The publisher does not even need to write this initial HTML code from scratch: it is trivial to copy this code from existing exhibits or from online tutorials. This is how HTML pages are often made—by copying existing pages, removing unwanted parts, and incrementally improving until satisfaction. The declarative syntax of HTML, the forgiving nature of web browsers and their reasonable defaults, and the quick HTML edit/test cycle make HTML publishing easy and instantly gratifying. Exhibit has been designed to afford the same behavior.

Figure 3.5 shows the final HTML code needed to render the exhibit in Figure 3.1 (logo graphics and copyright message omitted). The additional code, in black, configures the facets, the two views, and a lens template in the THUMBNAILS view. (Lens templates will be discussed in the User Interface Model section.)

Making the presentation look better can also involve filling in and fixing up the data schema. Figure 3.6 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.

To change the schema, e.g., renaming a property, the publisher can simply invoke the text editor's Replace All command. Or if the data is imported from a spreadsheet, she can just rename the corresponding column header label. Saving old versions of the data involves making copies of the data files. Changing schema and keeping versions of the data might not be as simple if databases were used.

Thus, just by editing one or two text files in any text editor, and perhaps editing data in a spreadsheet, a casual user with only basic knowledge of HTML and no programming skills can create a richly interactive web page with sorting, searching, filtering, maps, timelines, etc.—features that would otherwise take a whole team of web engineers months to implement.

```
<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-2.0/exhibit-api.js">
  </script>
</head>
<body>
  <div ex:role="viewPanel"></div>
</body>
</html>
```

Figure 3.4. To create the web page in Figure 3.1, 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.

```
1   <html>
2   <head>
3     <title>Topher's Breakfast Cereal Character Guide</title>
4     <link type="application/json" rel="exhibit/data" href="cereal-characters.json" />
5     <script src="http://static.simile.mit.edu/exhibit/api-2.0/exhibit-api.js"></script>
6     <script src="http://static.simile.mit.edu/exhibit/extensions-2.0/
7                  time/time-extension.js"></script>
8     <style>
9       .itemThumbnail {
10        width: 120px;
11      }
12    </style>
13  </head>
14  <body>
15    <table width="100%">
16    <tr valign="top">
17      <td width="20%">
18        <div ex:role="facet" ex:expression=".brand"   ex:facetLabel="Brands"></div>
19        <div ex:role="facet" ex:expression=".decade"  ex:facetLabel="Decades"></div>
20        <div ex:role="facet" ex:expression=".country" ex:facetLabel="Countries"></div>
21        <div ex:role="facet" ex:expression=".form"    ex:facetLabel="Forms"></div>
22      </td>
23      <td>
24        <div ex:role="viewPanel">
25
26          <div ex:role="view" ex:viewClass="Thumbnail"
27            ex:possibleOrders=".brand, .decade, .form, .country">
28
29            <div ex:role="lens" class="itemThumbnail">
30              <img ex:src-content=".thumbnail" />
31              <div ex:content="value"></div>
32            </div>
33          </div>
34
35          <div ex:role="view"
36              ex:viewClass="Timeline"
37              ex:start=".year"
38              ex:colorKey=".topic">
39          </div>
40        </div>
41      </td>
42    </tr>
43    </table>
44  </body>
45  </html>
```

One or more links to data
Exhibit API and extensions
custom styles
facets
lens templates specifying how to render each data item
views

Figure 3.5. The publisher starts with the `code in gray` (from Figure 3.4), includes more and more of the **`code in black`**, and tweaks until the desired result (Figure 3.1) is achieved (logo graphics and copyright omitted). Tweaking involves following online documentation or just copying code from other existing exhibits.

```
{
  types: {
    'Character': { pluralLabel: 'Characters' }
  },
  properties: {
    'url':       { valueType: "url" },
    'decade':    { valueType: "date" }
  }
  items: [
    // ... items ...
  ]
}
```

Figure 3.6. Schema information can be added to the JSON file to improve Exhibit's user interface.

3.2 Data Model

An Exhibit data model is a directed graph in which the nodes are either *items* or native values such as numbers, strings, and booleans, and the arrows are *properties* (Figure 3.7). Each property has a property value type which specifies the types—numbers, strings, booleans, items, etc.—of the nodes at the pointed ends of the arrows. In the simplest case where there is no relationship between items (no blue curved arrows in Figure 3.7), then the data model degenerates into a flat list of items with native-valued properties. A casual user would normally start with such a conceptually simple list of items and then when the need actually arises, link the items up to form a graph.

3.2.1 Items

Each item has one or more properties, one of which is the mandatory `label` property, which is a string naming that item in a human-friendly manner (friendliness is subject to the publisher's discretion). This label is used to render the item whenever a concise, textual description of that item is needed.

Every item is also mandated to have an `id` property, which identifies that item uniquely within the containing exhibit. If no `id` property value is specified explicitly when the item is being loaded into the database of the exhibit, the item's `label` property value is used as its `id`. Other items in the same exhibit can relate to this item simply by having property values equal to the item's `id`.

The third mandated property is `uri`, which is the URI used to name the item when it is exported to any RDF [28] serialization format. (The `uri` property helps make exhibits' data readily available to semantic web applications.) If no `uri` property value is specified when the item is being loaded, the URI is generated automatically by appending its id property value to the URL of the containing exhibit.

The last mandated property is `type`. An item's `type` property value is just a string, such as "President" and "Publication". If not explicitly specified, the item's

`type` defaults to "Item". Types are introduced to divide the set of items within a single exhibit into several subsets of conceptually different items, such as "Publication" vs. "Author", or "Patient" vs. "Hospital Record". If the data within an exhibit were to be stored in a relational database, there would logically be one relational table for each type.

Although there are four mandated properties, a publisher is only burdened to make up one—the `label` property—for each item. Note that in Figure 3.3, the item has neither `id` nor `uri` property value; both values will be generated. An `id` property value must be specified explicitly if another item with the same label has already been loaded into the database. A `uri` property value must be specified explicitly if the publisher intends the item to refer to some Web resource with an existing URI. For example, in an exhibit that compares several web sites' traffic on a bar chart, each item corresponds to one web site and the item's `uri` should logically be the web site's URL.

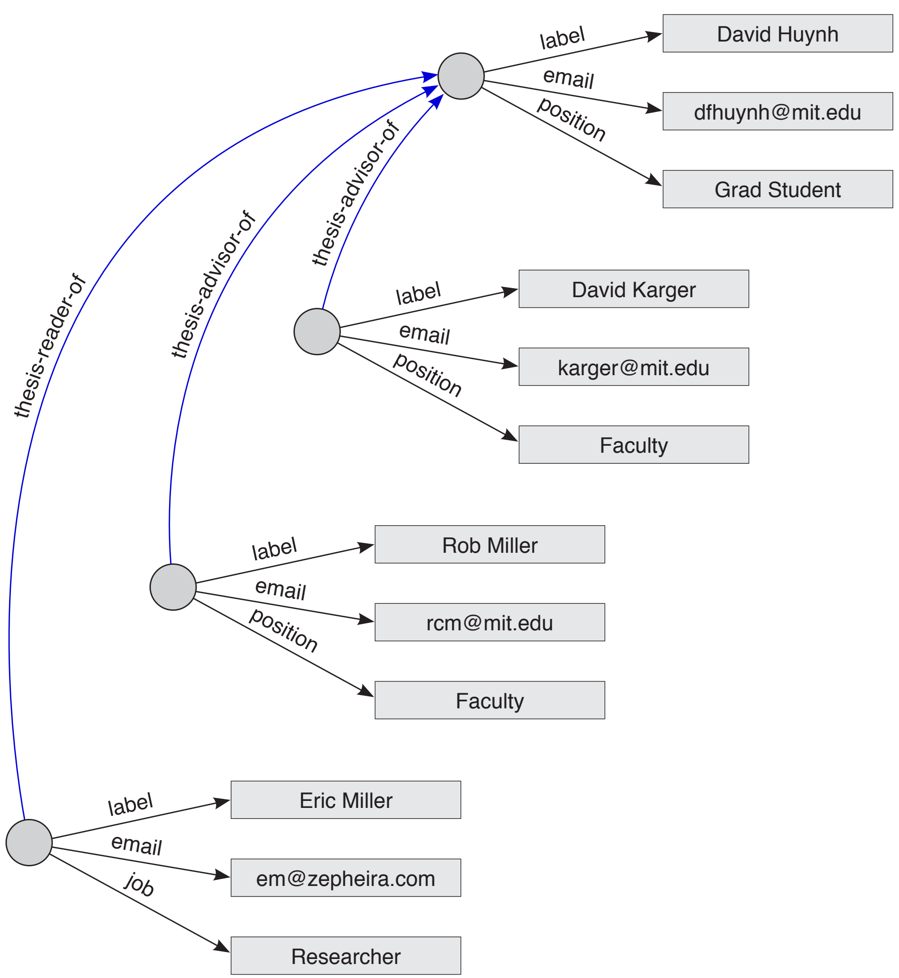


Figure 3.7. An Exhibit data model is a graph in which the nodes are items (circles) or native values (rectangles) and the arrows are properties.

3.2.2 Types

The information on types is schematic information. Each type has three mandated schematic properties: **`label`**, **`id`**, and **`uri`**. Note that when the **`type`** property of an item is specified, that **`type`** property value is the type's **`id`**, not the type's **`label`** (Figure 3.8). Whereas usually an item's **`label`** is specified and its **`id`** is generated from its **`label`**, a type's **`id`** must be specified first (by assigning some item that **`type`** property value) and then the type's **`label`** is taken to be the same as its **`id`**, unless overridden. Whereas for items, **`id`**s are primarily used to distinguish those with the same **`label`**, for types, **`id`**s are used as short-hand notations. For example, in Figure 3.8, the types' **`id`**s save the publisher a lot of typing while the types' **`label`**s are easy to comprehend for viewers.

Beside **`id`**, **`uri`**, and **`label`**, a type can have other schematic properties that help in the localization of the user interface. For instance, when a type has a **`pluralLabel`**, that schematic property value is used in English-speaking locales to generate user interface text for labeling several things of that type, e.g., 9 People instead of 9 Person.

There is no need for the publisher to explicitly declare every type in the **`types`** section. A type is added to the system whenever an item of that type is added. This lets the publisher focus on the items—the main point of her exhibit—and only add information on types and properties when they make the user interface better.

```
{
  items: [
    ...
    { ...
      type: "RCAct",
      ...
    },
    ...                 a type's id
  ],
  types: {
    "RCAct" : {
      label:       "Ribosomal Chaperone Activity",
      pluralLabel: "Ribosomal Chaperone Activities",
      uri:         "http://www.geneontology.org/go#GO:0000005"
    },
    "RMR" : {
      label:       "Regulation of Mitotic Recombination",
      pluralLabel: "Regulations of Mitotic Recombination",
      uri:         "http://www.geneontology.org/go#GO:0000019"
    }
  }
}
```

Figure 3.8. The **`types`** section in the JSON file specifies schematic properties of types, such as their **`label`**s and **`uri`**s.

3.2.3 Properties

Schematic information can also be given on properties. Properties have the same three mandated schematic properties as types (`label`, `id`, and `uri`) plus two more: `reverseLabel` and `valueType`. Together with `label`, `reverseLabel` lets a property be described in both directions (e.g., child of in one direction and parent of in the other).

A property's `valueType` hints at the nature of that property's values—whether they are numbers, or booleans, or strings, or items' `id`s, etc. There is a fixed set of value types, currently including: `text`, `number`, `date`, `boolean`, `url`, and `item`. All values of type `item` are locally unique `id`s of items in the same exhibit. Figure 3.6 shows how property value types are declared in data files.

Note that such declaration of value types are hints only. Exhibit will try to coerce property values into whatever types that are most appropriate for each particular use. For example, when a property `foo` with value type `text` is used to specify the starting dates of items on a timeline view, then all `foo` property values are automatically parsed into dates to render the timeline.

This design choice of tying value types to properties, rather than requiring type declaration on each value, facilitates incremental improvements to the data. For instance, `brand` property values in Figure 3.3 are initially, by default, of type `text`. This might satisfy the publisher 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 3.9).

3.2.4 Expressions

Exhibit provides an expression language that makes it convenient to access its graph-based data model.

Consider an exhibit whose data model is illustrated in Figure 3.10. The exhibit contains information on some publications, their authors, the schools where the authors teach, the countries where the schools are located, and perhaps more. Evaluating the expression `.written-by` on some publications yields their authors; evaluating `.written-by.teaches-at` yields the schools where the publications' authors teach; and so forth. The *hop operator* `.` traverses the graph *along* a given property. To traverse *against* a given property, the hop operator `!` is used. Thus, evaluating `!teaches-at!written-by` on some schools returns publications written by authors at those schools. The two kinds of hop operator can be mixed: evaluating `.written-by.teaches-at!teaches-at!written-by` on some publications returns all publications written by all authors teaching at the same schools as the authors of the original publications.

A sequence of alternating hop operators and property `id`s, such as `.written-by.teaches-at`, is called a *path*. Evaluating a path yields a collection of values (with or without duplicates). Why duplicate values are allowed will become clear subsequently.

```
{
  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 3.9. Elaboration of brand property values by specifying the value type for the property **brand** and adding Brand items.

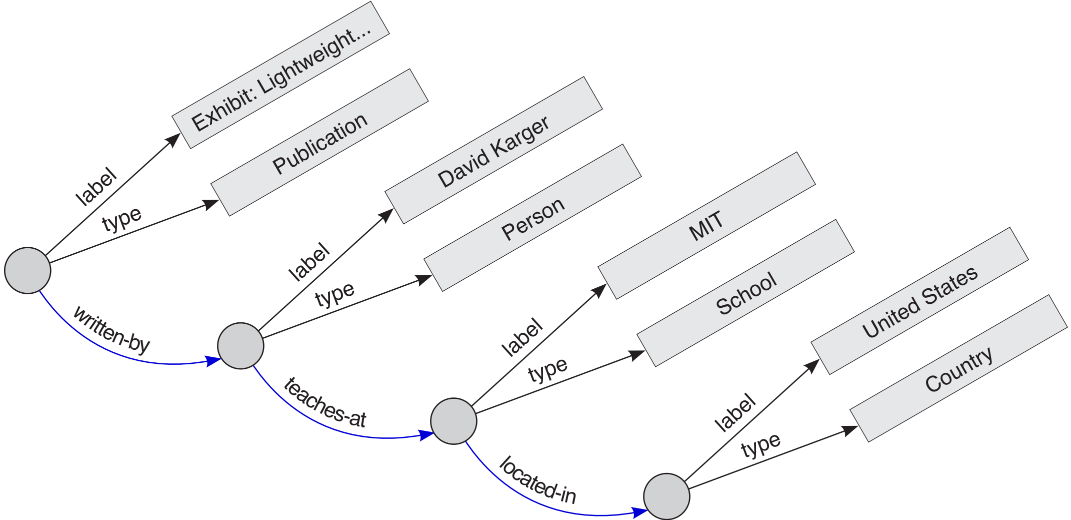


Figure 3.10. In this sample Exhibit data model, evaluating the expression .written-by.teaches-at.located-in on publications yields the countries in which the schools where the publications' authors teach are located.

In addition to paths, Exhibit expressions can also contain native values like strings and numbers, and *functions, operators* and *constructs*. For example, **`concat(.last-name, ', ', .first-name)`** computes a full name out of a last name and a first name, and **`count(.parent-of)`** returns the number of children of a person. Other functions include:

- **`union`** (for taking set unions of collections);
- **`contains`** (for testing if a collection contains a given element);
- **`exists`** (for testing if another expression returns any result, that is, if the result collection is non-empty);
- **`add`** and **`multiply`** (for taking summations and products of elements in one or more collections);
- **`and`**, **`or`**, and **`not`** (for combining boolean values);
- **`date-range`** for computing differences between two dates in a particular unit such as in days or in years;
- **`distance`** for computing geographical distance between two latitude/longitude pairs.

It is easy for third parties to implement more functions. Publishers just need to link the third parties' Javascript code into their exhibits.

Exhibit's operators are the standard arithmetic operators (**`+`**, **`-`**, **`*`**, **`/`**) and boolean operators (**`=`**, **`<>`**, **`><`**, **`<`**, **`<=`**, **`>`**, **`>=`**).

There are two constructs at the moment:

- **`if`**, which evaluates its first argument to a boolean value and based on that, chooses to evaluate either its second argument or its third argument;
- **`foreach`**, which evaluates its first argument into a collection; then for each element in that set, evaluates the expression that is its second argument on that element; and finally returns the union of all results.

The reason we need expressions to support duplicates in collections is as follows. Consider an exhibit about students and their assignments. To compute a student's average, we divide the sum of the grades of her assignments by the number of assignments: **`add(.assignment.grade)/count(.assignment)`**. Say a student gets 100 for two assignments. If the result of the sub-expression **`.assignment.grade`** is a collection without duplicates, then the student's average is computed incorrectly as 100 / 2 = 50. To fix this problem, Exhibit introduces two more hop operators **`.@`** and **`!@`** that prevent duplicates from being eliminated further down the evaluation process. Thus, **`add(.assignment.@grade)/count(.assignment)`** would compute the correct average.

The full production rules for Exhibit expression language is provided in Figure 3.11.

3.2.5 Data Import/Export

While the Exhibit JSON format is relatively readable and writeable, it might not be the format of choice for some publishers. They might already have their data in another format (e.g., BibTex), or they might be used to editing tools that do not export to Exhibit JSON (e.g., Excel). These publishers can simply annotate the **`<link>`** elements referencing their data with the right mime-type (e.g., **`type="application/bibtex"`**) and Exhibit will import it automatically. Exhibit's importer architecture allows for third parties to plug in their own importers.

Exhibit also has an extensible exporter architecture that can serialize its data into several formats, ready to be copied off. The natively provided exporters can generate RDF/XML, Exhibit JSON, Semantic MediaWiki extension wikitext [72], and Bibtex. The purpose of these exporters is to facilitate and encourage propagation of reusable data by offering convenience to both users and publishers. 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

```
<expression>          ::= <sub-expression>
                        | <expression> <expr-op> <sub-expression>
<sub-expression>      ::= <term>
                        | <sub-expression> <subexpr-op> <term>
<term>                ::= <factor>
                        | <term> <term-op> <factor>
<factor>              ::= <number>
                        | <string>
                        | <construct>
                        | <function-call>
                        | <path>
                        | "(" <expression> ")"

<construct>           ::= <if-construct>
                        | <foreach-construct>
                        | <default-construct>
<if-construct>        ::= "if" "(" <expression> "," <expression> "," <expression> ")"
<foreach-construct>   ::= "foreach" "(" <expression> "," <expression> ")"
<default-construct>   ::= "default" "(" <expression-list> ")"

<function-call>       ::= <identifier> "(" <expression-list>? ")"

<expression-list>     ::= <expression>
                        | <expression-list> "," <expression>

<path>                ::= <hop-list>
                        | <identifier> <hop-list>
<hop-list>            ::= <hop>
                        | <hop-list> <hop>
<hop>                 ::= <hop-op> <property-id>

<expr-op>             ::= "=" | "<>" | "<" | "<=" | ">" | ">="
<subexpr-op>          ::= "+" | "-"
<term-op>             ::= "*" | "/"
<hop-op>              ::= "." | "!" | ".@" | "!@"
```

Figure 3.11. Production rules for the Exhibit expression language.

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.

3.3 User Interface Model

Exhibit's user interface consists of several types of component:

- *collections*;
- widgets: *facets*, *views*, and *lenses*; and
- helpers: *coders*.

Exhibit also maintains a hierarchy of *UI contexts* that store inheritable UI settings such as *formatting rules*. These components and the formatting rules can all be specified in HTML. The following subsections define these various UI concepts and describe how they can be specified in HTML.

3.3.1 Collections

A collection contains a subset of the items currently in the exhibit's database. Currently, a collection can be defined to contain items by some particular types, e.g.,

```
<div ex:role="collection" ex:itemTypes="Man; Woman" id="adults"></div>
<div ex:role="collection" ex:itemTypes="Boy; Girl" id="kids"></div>
<div ex:role="collection" ex:itemTypes="Man; Woman; Boy; Girl"></div>
```

Each collection has an ID, which can be specified using the HTML **`id`** attribute. The ID of a collection is used to refer to it from other components, as will be discussed in later subsections. If no **`id`** attribute is given, the ID is assumed to be "**`default`**" and the effect of the definition is to redefine the **`default`** collection, which comes, by default, with every exhibit. If not redefined, the **`default`** collection contains all items currently in the exhibit's database.

What a collection originally contains is called its *root set*. This root set can be filtered down to a *filtered set* by facets attached to the collection, as will be discussed next.

3.3.2 Facets

A facet is a component whose purpose is to filter a collection's root set down to a filtered set. Facets can be declared as follows:

```
<div ex:role="facet"
     ex:expression=".ethnicity"
     ></div>
```

```
<div ex:role="facet"
     ex:expression=".profession"
     ex:collectionID="adults"
     ></div>
<div ex:role="facet"
     ex:expression=".height * 2.54"
     ex:facetLabel="Height (cm)"
     ex:collectionID="kids"
     ex:facetClass="NumericRange"
     ex:interval="10"
     ></div>
```

The **`ex:expression`** attribute is required and in the simplest case, it specifies the property by which to filter the collection to which the facet is attached. In the first example, the facet can be used to filter the attached collection by (people's) ethnicity; in the second, by professions; and in the third, by heights. The expression does not have to be a simple property; it can compute more complex values such as seen in the third example. The **`ex:collectionID`** specifies which collection a facet is attached to; if missing, the **`default`** collection is used (as in the first example).

The **`ex:facetLabel`** attribute gives a text label to the facet. If missing, and if the expression is a simple property, Exhibit uses the property's label; otherwise, Exhibit shows an error message where the label should be rendered.

The **`ex:facetClass`** attribute specifies which class of facet to instantiate. There are two classes being supported: **`List`** and **`NumericRange`** (**`List`** is the default class). **`List`** facets show each value computed by the expression as a choice for filtering while **`NumericRange`** facets convert each value into a number and group values into intervals.

3.3.3 Views

A view is a component that renders the filtered set of items in a collection to which it is attached. As the collection gets filtered by the facets attached to it, the views attached to that collection update themselves to show the current filtered set.

Views can be specified much like facets. Each kind of view supports its own settings that are appropriate to it. For example, a tabular view supports settings pertaining to its columns, e.g.,

```
<div ex:role="view" ex:viewClass="Tabular"
   ex:columns       = ".label, .imageURL, .party, .age"
   ex:columnFormats = "text,   image,     text,   number"
   ex:sortColumn    = "2"
   ex:sortAscending = "true"
   ></div>
```

3.3.4 Lenses

An Exhibit lens renders one single item. Lenses are *not* specified individually per item, but they are instantiated from a few *lens templates*, which are specified in HTML.

A template is just a fragment of HTML that can be specified in-line, as in Figure 3.12, or in a different file.

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 3.12, 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 3.12 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 **`ex: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 3.12 will not generate a broken image. The **`ex:if`** attribute of an element specifies an expression that evaluates to a boolean; if the boolean is true, the first child element of the element is processed; otherwise, the second child element is processed if it is present. The **`ex:select`** attribute of an element specifies an expression that evaluates to some value which is used to select which child element to process. The child element with the matching **`ex:case`** attribute value is processed. Otherwise, the last child element that does not have any **`ex:case`** attribute is processed.

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

3.3.5 UI Contexts and Formatting Rules

Each component has a UI context which gives it access to UI settings, such as how dates should be formatted (e.g., "July 4" or "7/4" or "7月4日"). Such formatting rules can be added to the UI context of a component through the **`ex:formats`** attribute on the component's HTML specification, e.g.,

```
<div ex:role="view"
  ex:viewClass="Timeline"
  ex:start=".birth"
  ex:formats=
    "date { mode: short; show: date } number { decimal-digits: 2 }"
  ></div>
```

In this example, whenever dates are rendered inside that timeline view, they are displayed in short form without time of the day (e.g., "04/07/07" for July 4, 2007). Whenever numbers are displayed, two decimal digits are shown.

```
<table ex:role="exhibit-lens" cellspacing="5" style="display: none;">
 <tr>
   <td>
     <img ex:if-exists=".cereal" 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 3.12. Lens template for showing a breakfast cereal character in a pop-up bubble as shown in Figure 3.13.

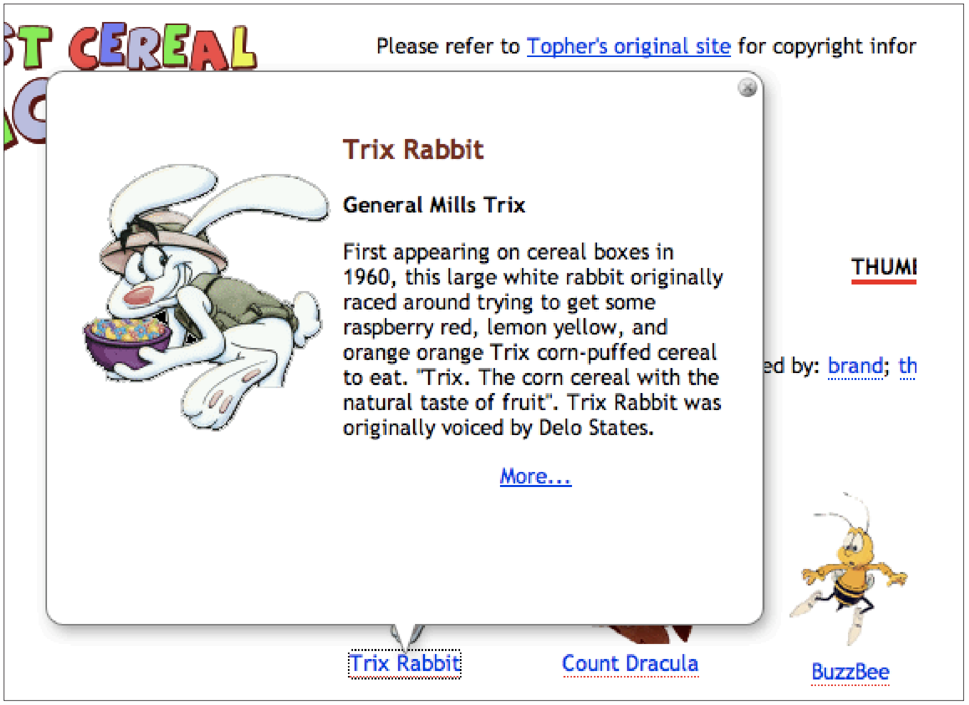


Figure 3.13. Bubble showing a breakfast cereal character.

Note that the resemblance in syntax to CSS is intended for familiarity. An **`ex:formats`** specification consists of zero or more *rules* (cf. CSS rules). Each rule consists of a selector which specifies what the rule applies to (cf. CSS selectors) and then zero or more *settings* inside a pair of braces. In the example, there are two rules and two selectors, **`date`** and **`number`**. Those two selectors select the value types to which the rules apply.

For each selector there is a different set of allowable settings. For example, for **`date`** there are **`time-zone`**, **`mode`**, and **`template`**. For each setting there is a set of acceptable setting values. For example,

- **`time-zone`** takes a number or the keyword **`default`**;
- **`mode`** takes any one of the keywords **`short`**, **`medium`**, **`long`**, and **`full`** to specify how detailed to render dates and times;
- **`template`** takes a string that acts as a template for formatting dates and times in case those four standard modes do not satisfy the publisher's needs; and
- **`negative-format`** (for currency) takes any combination of the flags **`red`**, **`black`**, **`parentheses`**, **`no-parentheses`**, **`signed`**, and **`unsigned`** to specify how negative currency values should be rendered.

As in CSS, URLs in formatting rules are wrapped in **`url()`**. Similarly, expressions are wrapped in **`expression()`**. For example,

```
ex:formats=
  "item { title: expression(concat(.full-name, ', ', .job-title)) }"
```

specifies that when a textual description of an item is needed, render the **`full-name`** property value followed by a comma and the **`job-title`** property value.

The full production rules for Exhibit's format language is provided in Figure 3.14.

Other UI settings beside formatting rules can be specified through other attributes, such as **`ex:bubbleWidth`** for the width in pixels of popup bubbles. Settings not specified in a view's HTML specification are inherited from the view's outer

```
<rule-list>       ::= <rule>*
<rule>            ::= <selector> [ "{" [ <setting-list> ] "}" ]
<selector>        ::= "number" | "date" | "boolean" | "text" | "image" |
                      "url" | "item" | "currency" | "list"

<setting-list>    ::= <setting> | <setting-list> ";" <setting>
<setting>         ::= <setting-name> ":" <setting-value>

<setting-value>   ::= <number>
                    | <integer>
                    | <non-negative-integer>
                    | <string>
                    | <keyword>
                    | <url>
                    | <expression>
                    | <flags>
<flags>           ::= <keyword>+
```

Figure 3.14. Production rules for the Exhibit format language.

lexical scope, that is, from the settings specified on components whose HTML specifications lexically contain this view's HTML specification.

3.3.6 Coders

A coder translates a piece of information to some visual feature, e.g.,

- from the **`political-party`** property value "Republican" to the color red, and from "Democrat" to the color blue;
- from the **`service`** property value "Restaurant" to an icon showing a fork and a knife, and from "Hotel" to an icon showing a bed;
- from the **`magnitude`** property value 5 (assumed to be the magnitude of an earthquake) to the number 40, indicating how large the corresponding map marker should be;
- from the **`temperature`** property value -30 (assumed to be in Celsius) to the color code **`#0000aa`**, a particular shade of blue indicating how cold it is.

For the four examples above, these are the coders' specifications:

```
<div ex:role="coder" ex:coderClass="Color" id="political-party-colors">
   <span ex:color="red">Republican</span>
   <span ex:color="blue">Democrat</span>
   <span ex:color="gray"  ex:case="default">Any other party</span>
   <span ex:color="white" ex:case="mixed">Multiple parties</span>
   <span ex:color="black" ex:case="missing">No party</span>
</div>

<div ex:role="coder" ex:coderClass="Icon" id="service-icons">
   <span ex:icon="fork-knife.png">Restaurant</span>
   <span ex:icon="bed.png">Hotel</span>

   <span ex:icon="question-mark.png"
         ex:case="default">Other service</span>
   <span ex:icon="many-question-marks.png"
         ex:case="mixed">Multiple services</span>
   <span ex:icon="x.png"
         ex:case="missing">No service</span>
</div>

<div ex:role="coder"
     ex:coderClass="NumericGradient" id="earthquake-magnitudes"
     ex:gradientPoints="1, 20; 10, 60"
     ></div>

<div ex:role="coder" ex:coderClass="ColorGradient" id="temperature-col-
ors"
   ex:gradientPoints="-40, #000088; 0, #ffffff; 50, #ff0000"
   ></div>
```

The **`ex:case`** attribute is used to specify special cases, such as when a single marker on the map corresponds to several values to translate (e.g., because several politicians belong to different political parties were born in the same city), or when there is no data to translate (e.g., because there is no temperature recorded for a particular city). The **`ex:case="default"`** attribute value is used to code all remaining cases (e.g., other political parties beside Democrat and Republican).

To connect a view or a facet to a coder, you need to specify on the view or facet's HTML specification which coder to use and what data to feed it. To have a map view plotting politicians colored by political parties, then we can specify that map view as follows:

```
<div ex:role="view" ex:viewClass="Map"
    ex:latlng=".latlng"
    ex:colorCoder="political-party-colors"
    ex:colorKey=".party"
    ></div>
```

As another example, to construct a map view of buildings and the services available in them, write:

```
<div ex:role="view" ex:viewClass="Map"
    ex:latlng=".latlng"
    ex:iconCoder="service-icons"
    ex:iconKey=".service"
    ></div>
```

Several coders could potentially be used in the same view. For example, on a map plotting natural disasters, an icon coder can indicate whether a disaster is an earthquake, a volcano eruption, a tsunami, etc.; a marker size coder can show the magnitudes of those disasters; and a color coder can show the casualties (e.g., more red means more casualties).

3.4 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 publishers do not need to download any software, and users who view exhibits do not need to install any browser extension. This zero cost is the signature of client-side Web APIs and is largely responsible for the explosion in the embedding use of Google Maps [11].

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 components to supplement Exhibit.

Exhibit's architecture is illustrated in Figure 3.15. At the bottom is the data layer consisting of the database, the expression language parser and evaluator, and importers and exporters. At the top is the user interface layer, which consists of three sub-layers:

- UI contexts and localization resources—storage of presentation settings for the rest of the user interface layer.
- collections and coders—components that do not render to the screen but determine what data widgets should render and how to render it.
- widgets which perform the actual rendering and support interactions.

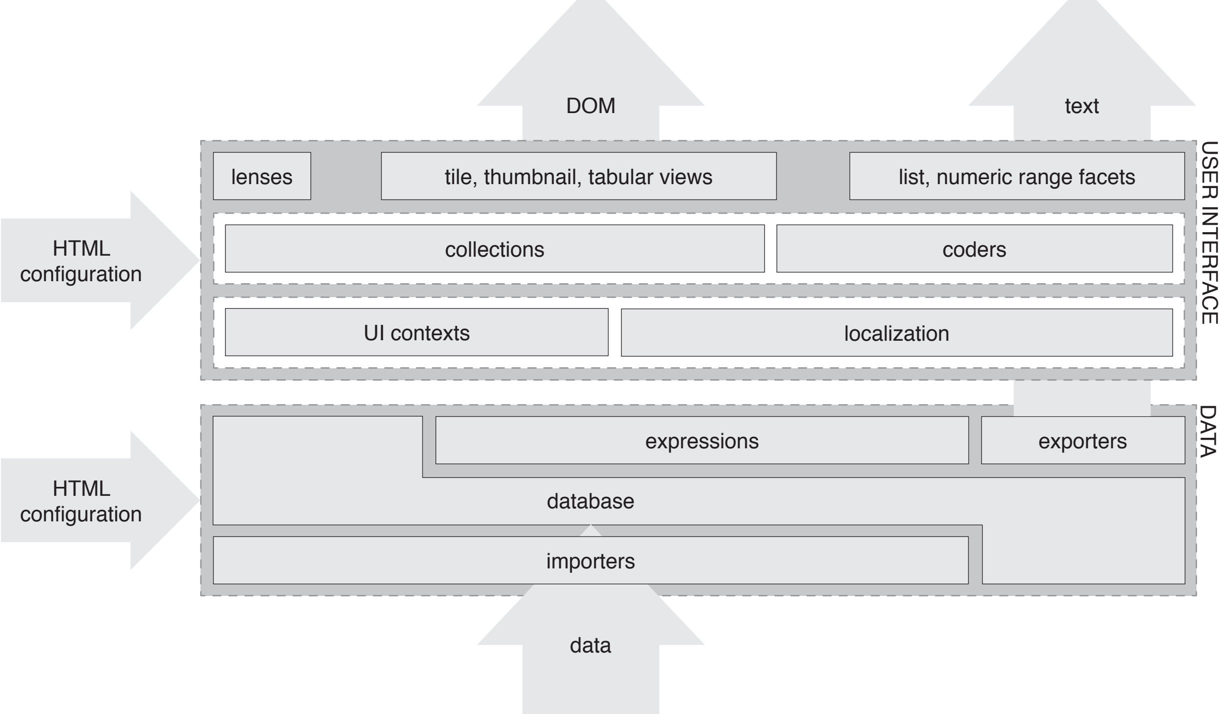


Figure 3.15. Exhibit's architecture

There are several points of extensibility. More views, facets, and coders can be added. More importers, exporters, and functions (to be used in expressions) can be registered. For example, a new facet class can be added to show a calendar or a timeline supporting date range selection instead of just listing individual dates.

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.

3.5 Evaluation

Exhibit was evaluated in two ways: by its performance in contemporary popular browsers and by its actual usage by casual users.

3.5.1 Performance

Four platform/browser configurations were chosen for testing Exhibit version 2.0 as of July 26, 2007:

- MacBook Pro laptop, Mac OSX 10.4.10
  2.16 GHz Intel® Core 2 Duo, 2 GB RAM
  1. Firefox 2.0.0.5
  2. Safari 3.0.2 (522.12)
- Dell Dimension 8200 desktop, Windows XP SP2
  2.53 GHz Intel® Pentium® 4 CPU, 1.00 GB RAM
  3. Firefox 2.0.0.5
  4. Internet Explorer 7.0.5730.11

For Firefox, new profiles were created for testing. For the other browsers, their caches were cleared.

Exhibit's performance was measured in two ways: by its load time and by its interactivity time. The only independent variable was the number of items in the test exhibit, which varied from 500 to 2,500. The results are shown in Figure 3.16 and Figure 3.17, which will be explained below.

3.5.1.1 Load Time

Load time was divided into two stages:

- Data loading: populating the database with data that has already been received at the browser, thus excluding network traffic cost;
- UI rendering: constructing the initial presentation of the exhibit, which includes all the costs of computing the facets, querying the database for data to construct the lenses, and actually generating DOM elements.

There were two variations of UI rendering:

- All items were sorted by label and rendered in a tile view.
- All items were sorted by label but only the first 10 were rendered.

Four platform/browser configurations combined with two variations of UI rendering yielded the eight charts shown in Figure 3.16. These charts show that UI rendering cost considerably more than data loading, and that by rendering only the first 10 items, reasonable load time performance (under 5 seconds) could be achieved in all tested platform/browser configurations for 1,000 item exhibits.

3.5.1.2 Interactivity Time

To test interactivity time, I measured the time it took from clicking on a facet value with a count of 3 (after the test exhibit has finished loading and rendering) to when

the exhibit has finished rendering the filtered set of items and updating the facets. There were also two variations of initial UI rendering as in the load time test, producing a total of eight traces in the two plots in Figure 3.17. Rendering all items at the beginning did affect the filtering time substantially, perhaps because there was a high cost for removing generated DOM elements. Responsive performance (within half a second) could be achieved in all platform/browser configurations if only 10 items were rendered initially.

These test results yielded encouraging evidence that reasonable performance is achievable by Exhibit's client-side approach. More browser optimizations and faster hardware will soon make Exhibit scale better. Already we see that the latest beta version of Safari double the speed of the latest released version of Firefox on the same platform.

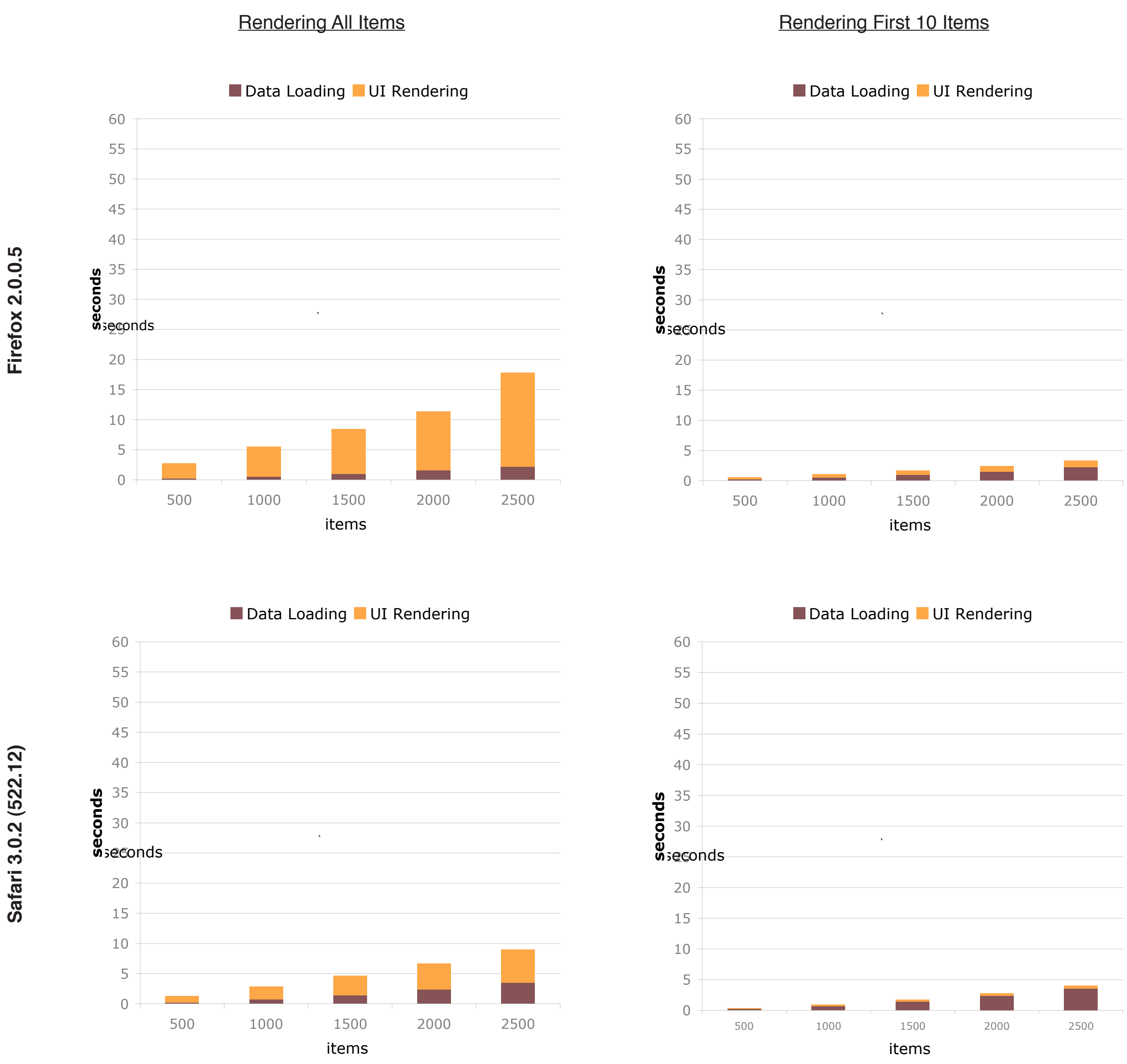
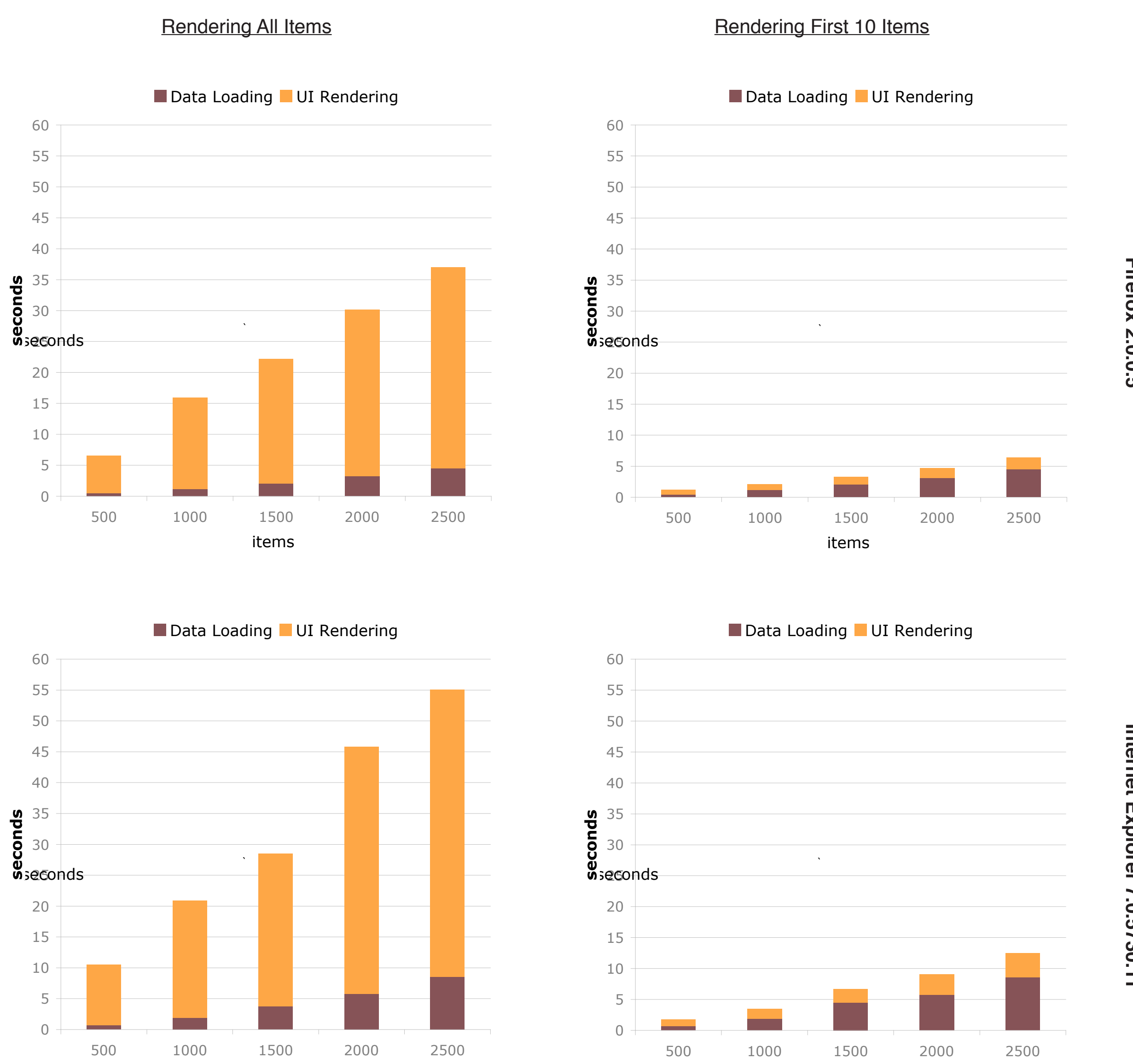


Figure 3.16. Exhibit's load-time performance results on four configurations (two operating systems, two browsers per operating system) with the number of items as the independent variable. At 1000 items, by rendering only the first 10 items, the data loading and UI rendering times combined take shorter than 5 seconds on all configurations.


Windows XP SP2, Pentium(R) 4 CPU 2.53GHz, 1.00 GB RAM
Rendering All Items
Rendering First 10 Items
Data Loading
UI Rendering
seconds
items
500
1000
1500
2000
2500
Firefox 2.0.0.5
Internet Explorer 7.0.5730.11

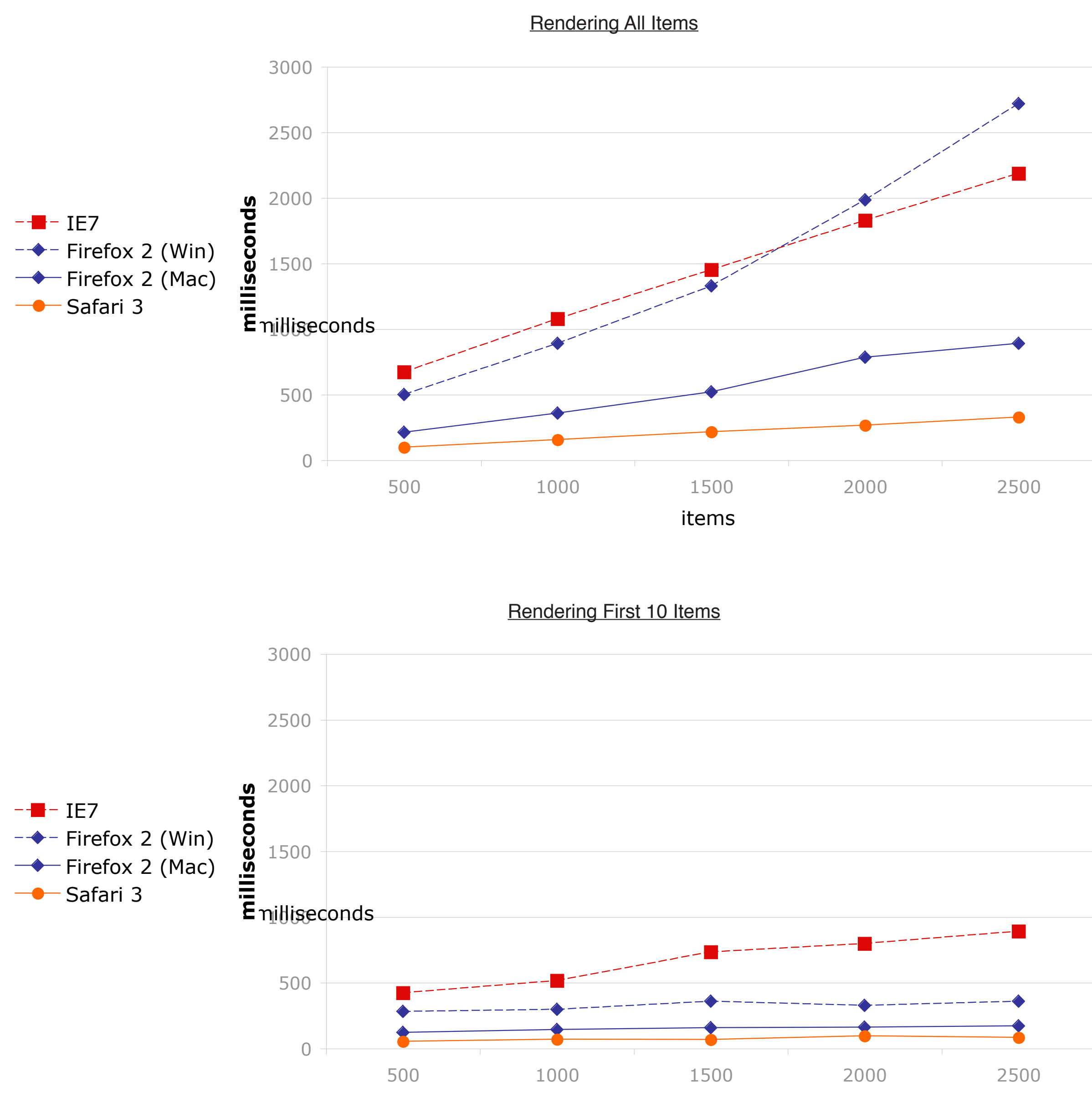


Figure 3.17. The times taken to filter down to 3 items from the same initial configurations in Figure 3.16. By rendering only the first 10 items initially, the filtering times on all configurations can be reduced to below one second even for exhibits containing 2500 items in total.

3.5.2 Usage

Eight months after Exhibit 1.0 was released, server logs indicate that more than 800 web pages link to the Exhibit API. Note that deploying Exhibit as a Web API serves two purposes: letting Exhibit authors do away with downloading and setting up software as well as allowing us to track actual usage of Exhibit through referrals in our server logs.

Table 3.1 shows usage of Exhibit at several top level domains. Many of them are attempts to follow a tutorial for creating exhibits. Some are broken and abandoned attempts; some are duplicate versions of the same exhibits; and a few are in foreign languages which I do not understand.

It is very difficult to categorize the rest as they are very unique and different from one another, as much as existing web sites are unique. An early attempt to group them by topics is shown in Table 3.2. Note the many esoteric topics that form the long tail distribution. Even within a single topic such as "people" there is diversity, ranging from university presidents, to army officers, to tavern keepers. These exhibits can contain as few as three items, to as many as 1,700. A few have been well designed visually, many have layouts and color themes copied off from other exhibits, and the rest are bare. Out of those 700 or so pages, 175 make use of the timeline view and 123 of the map view.

Figures 3.18 – 3.27 show ten exhibits created by real Exhibit authors. Figure 3.18 shows an exhibit that has been created by someone who claimed in an e-mail message to me not to be a programmer. Even though there is plenty of information online about music composers, he still felt compelled to create a "database" of music composers.

Figure 3.19 displays information about historical figures. It is meant to be classroom materials for grade six. Even though the same materials are available elsewhere, this teacher wanted to present them to his students in a different way, perhaps to match the rest of his own curriculum. Similarly, a Charmed TV series fan put up Figure 3.20 even an official episode guide exists.

On the other hand, the information shown in Figure 3.21—information about someone's ancestors—will unlikely be published anywhere else, as it is very personal information. Similarly, the data about sweet products of a small Spanish shop shown in Figure 3.22 is unique.

Figure 3.23 shows an exhibit of banned books. By making it easier for end users to publish arbitrary data, Exhibit makes such information, shunned by official sources, more likely to see the light of day. From this perspective, Exhibit serves as a tool for empowerment.

Exhibit is also found to be a useful organization tool thanks to its faceted browsing and flexible visualization features: Figure 3.24 shows information about little league games and practices. Several tabular views are configured to show the same data differently, perhaps to different groups in the exhibit's target audience. The four digit numbering scheme in the events' labels indicates expected long-term use.

More interestingly, Exhibit has been found useful even by programmers. The programmer responsible for implementing a browsing interface for the learning resources in the Language Learning and Resource Center at MIT chose Exhibit over building his own web application and have made several exhibits, including the one shown in Figure 3.25.

Similarly, a database researcher has found it easier to make an exhibit of his publications, seen in Figure 3.26, than to build his own database-backed publication site.

Finally, semantic web researchers have turned to Exhibit as a means to display their data (Figure 3.27), especially because they have no tool to display their data in such a usable and useful manner.

3.6 Summary

This chapter explored several ideas that made publishing data to the Web in browsable and reusable forms easy for casual users, those without programming skills. Embodying these ideas, the lightweight publishing framework Exhibit has let a few hundred people create richly interactive web sites with maps and timelines.

TLD	exhibit count
.com	246
.edu	215
.org	91
.net	35
.nl	20
.fr	19
.ca	18
.uk	17
.it	11
.info	7
.co	6
.ro	6
.de	5
.us	5
.mil	4
.at	3
.dk	3
.fi	2
.gov	2
.ws	2
.ar	1
.au	1
.cx	1
.cz	1
.gr	1
.hu	1
.in	1
.int	1
.is	1
.nu	1
.pl	1
.ru	1
.se	1
.za	1

Table 3.1. A survey of real-world exhibits by topics shows adoption in many top level domains.

topic	exhibit count	item count
publications	20	4013
people	18	2647
scientific data	6	2517
photos	5	871
classroom materials	5	477
engineering timetables	4	899
products	3	433
movies	3	237
sports	3	195
events	3	162
presentations	3	128
projects	2	957
web sites	2	317
books	2	199
locations	2	127
recipes	2	97
personal histories	2	12
play castings	1	441
games	1	422
space launch sites	1	280
world conflicts	1	278
hotels	1	203
restaurants	1	97
blog posts	1	97
cars	1	89
links	1	70
buildings	1	68
breweries & distilleries	1	55
elections	1	50
activities	1	50
museums	1	45
sounds	1	32
historical artifacts	1	24
characters	1	24
streets	1	24
early christian periods	1	21
HMong state population	1	15
vocabulary	1	14
history	1	13
materials	1	8
documents	1	7
plays	1	5
wars	1	3

Table 3.2. A survey of real-world exhibits by topics shows a long tail distribution.

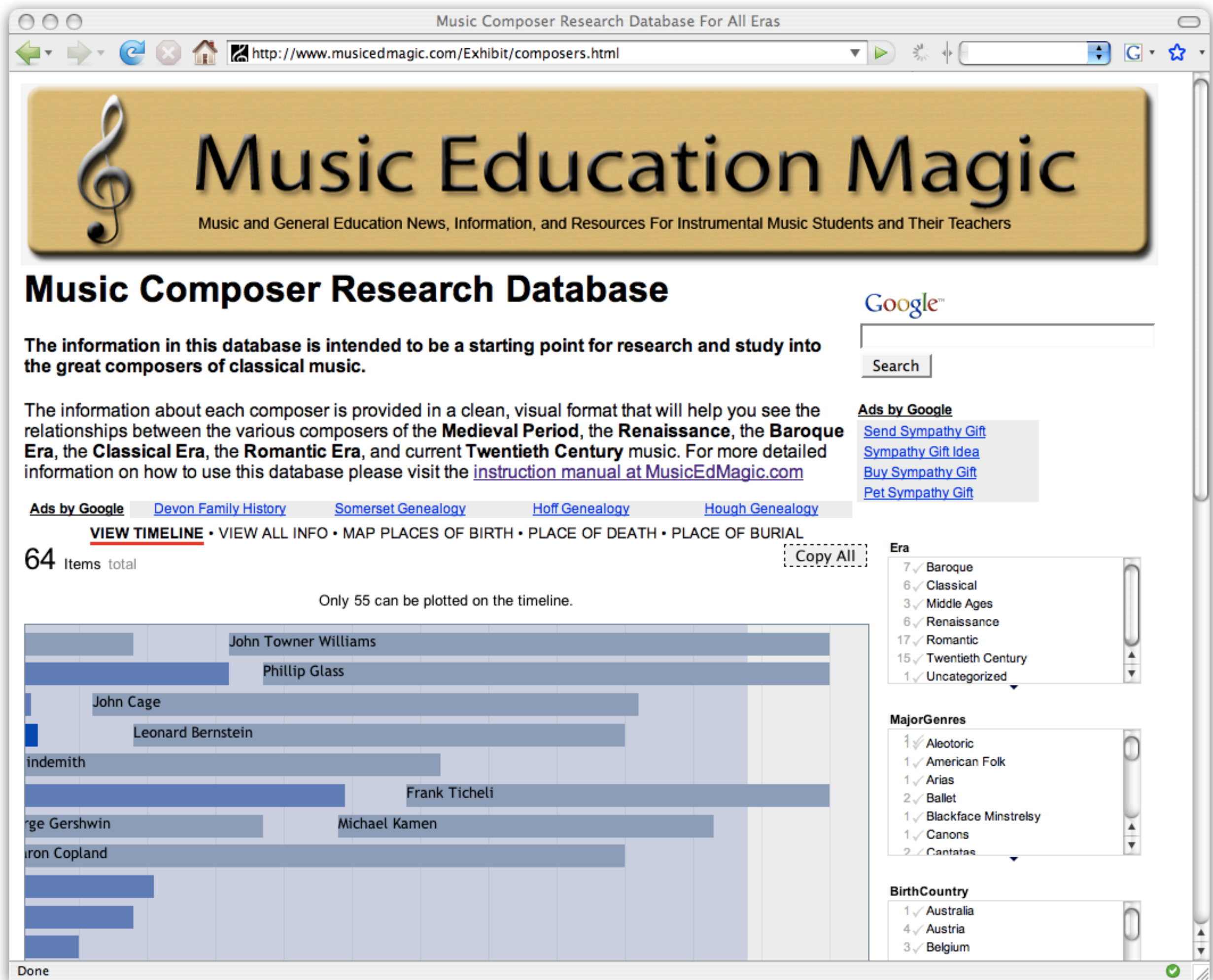


Figure 3.18. An exhibit of 64 music composers from the eleventh century to the present day. The composers can be viewed on a timeline by their lifetimes, and on maps by their birth places, death places, and burial places. They can be browsed by the music eras, by their major genres, and by their countries of birth. Even with all the information about composers already on the Web, someone still feels compelled to create his own research "database" of composers.

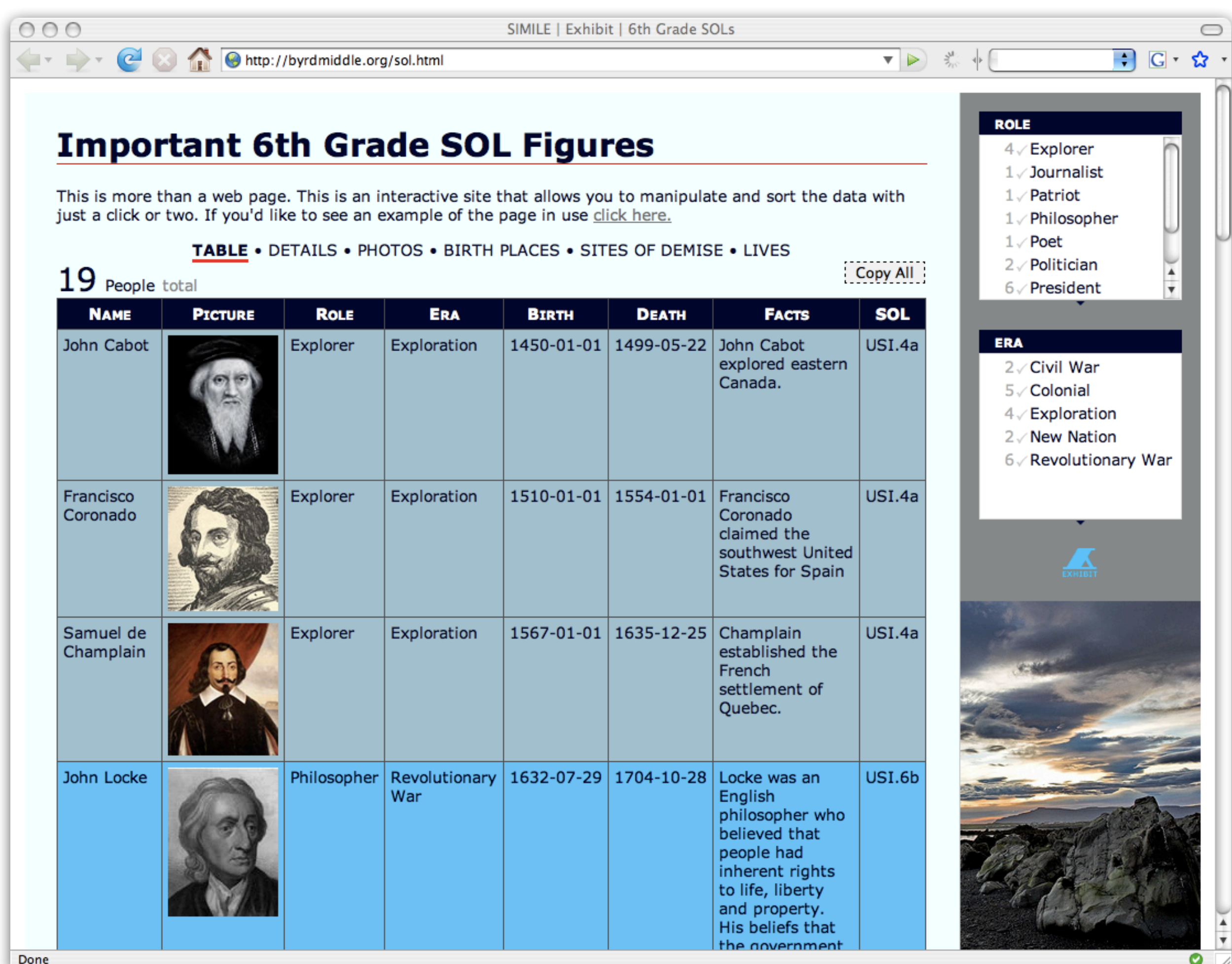


Figure 3.19. An exhibit of 19 important figures in history that serves as educational materials in a grade 6 classroom. Even though the same materials are available elsewhere, each teacher might want to present them to their students in a different way, adding personal insights that match the rest of their curriculum.

Figure 3.20. An exhibit of 178 episodes from the Charmed TV series.

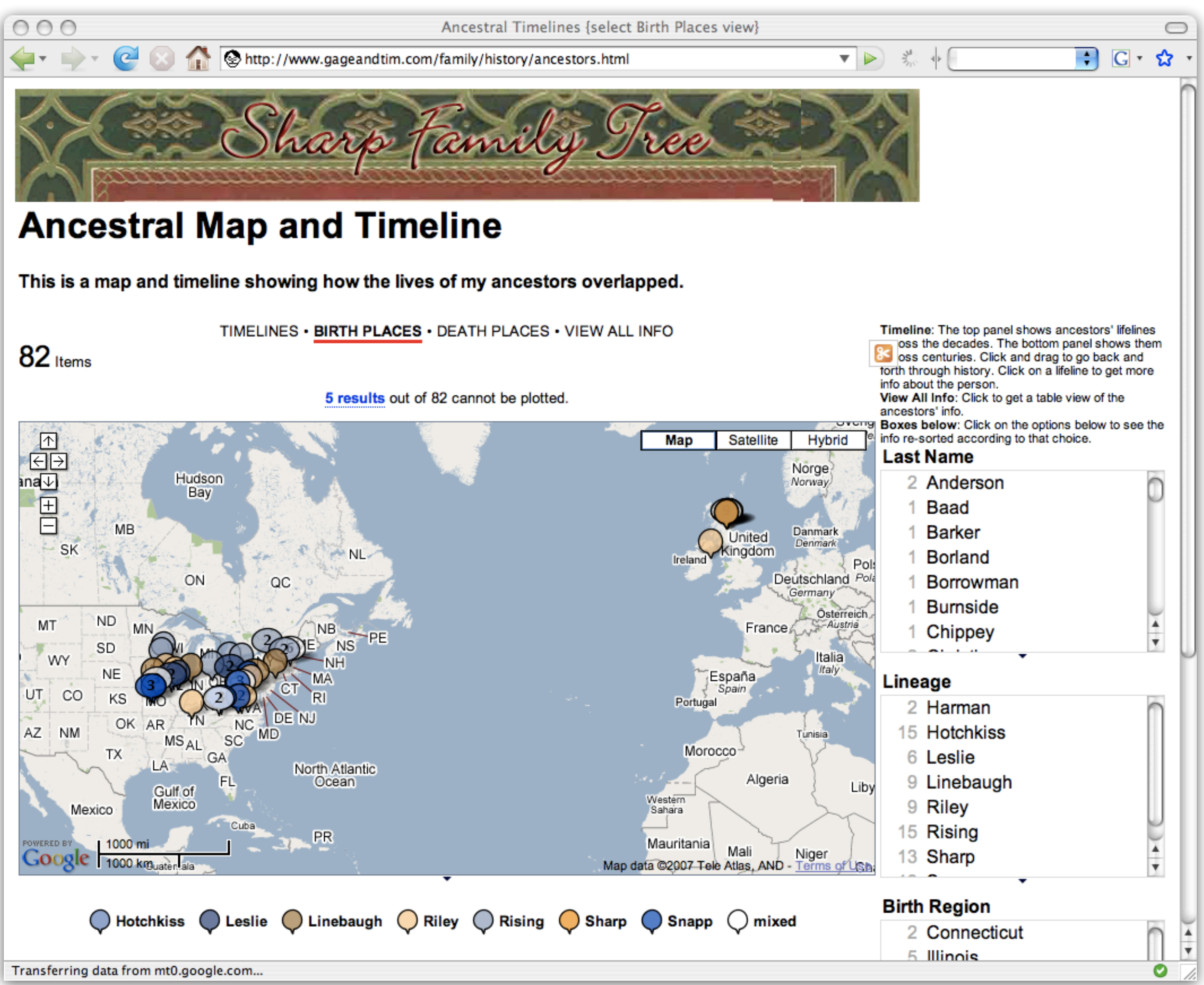


Figure 3.21. An exhibit of 82 ancestors containing a timeline and maps of birth places and death places. The people can be browsed by their last names, lineage, birth regions, and death regions. The information found in this exhibit will unlikely be on any other web site. That is, even with Wikipedia and Freebase growing in size everyday, some information will just never find its way into such public repositories.

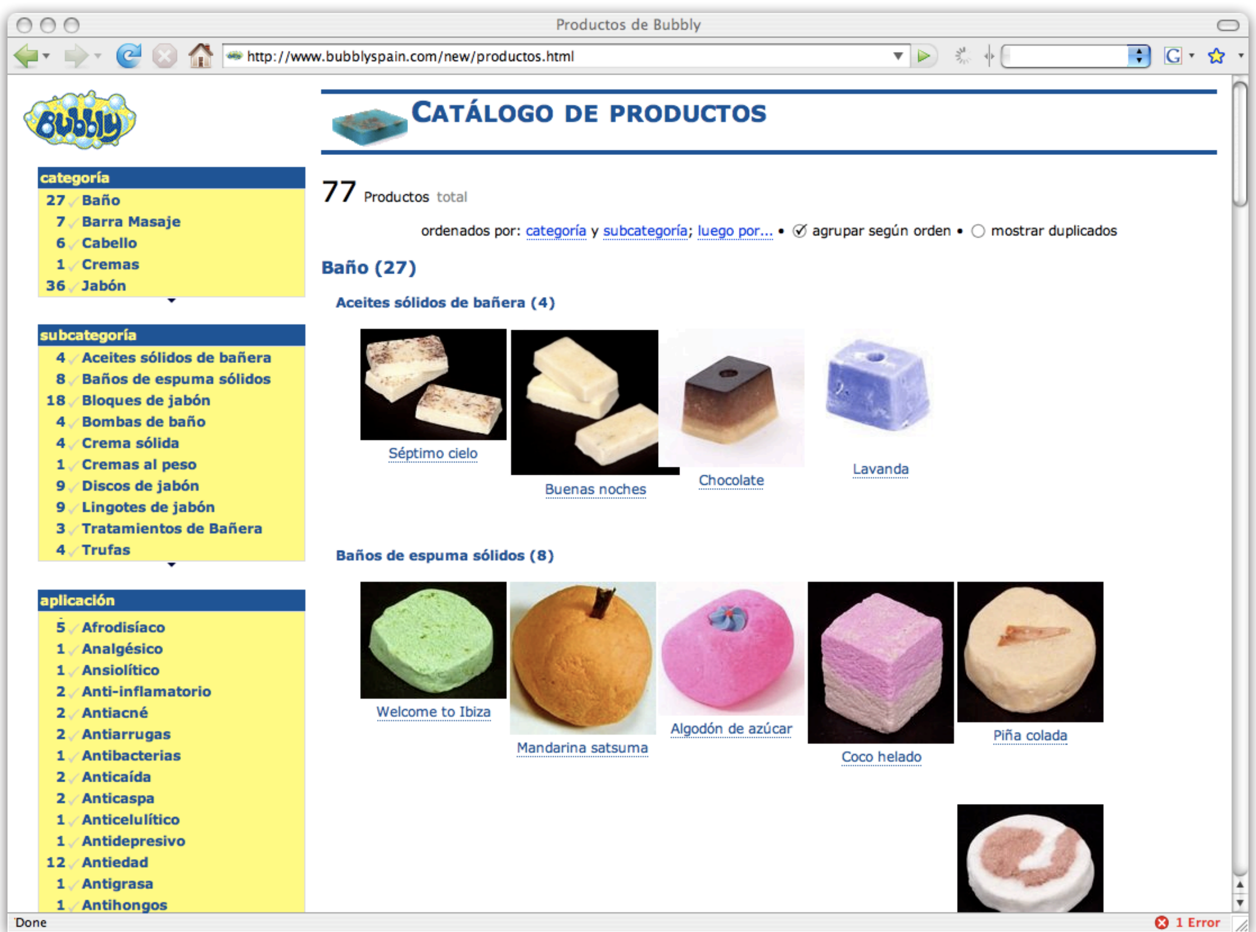


Figure 3.22. An exhibit of 77 sweet products in a Spanish shop. For such a small business, investing in a full three-tier web application is costly.

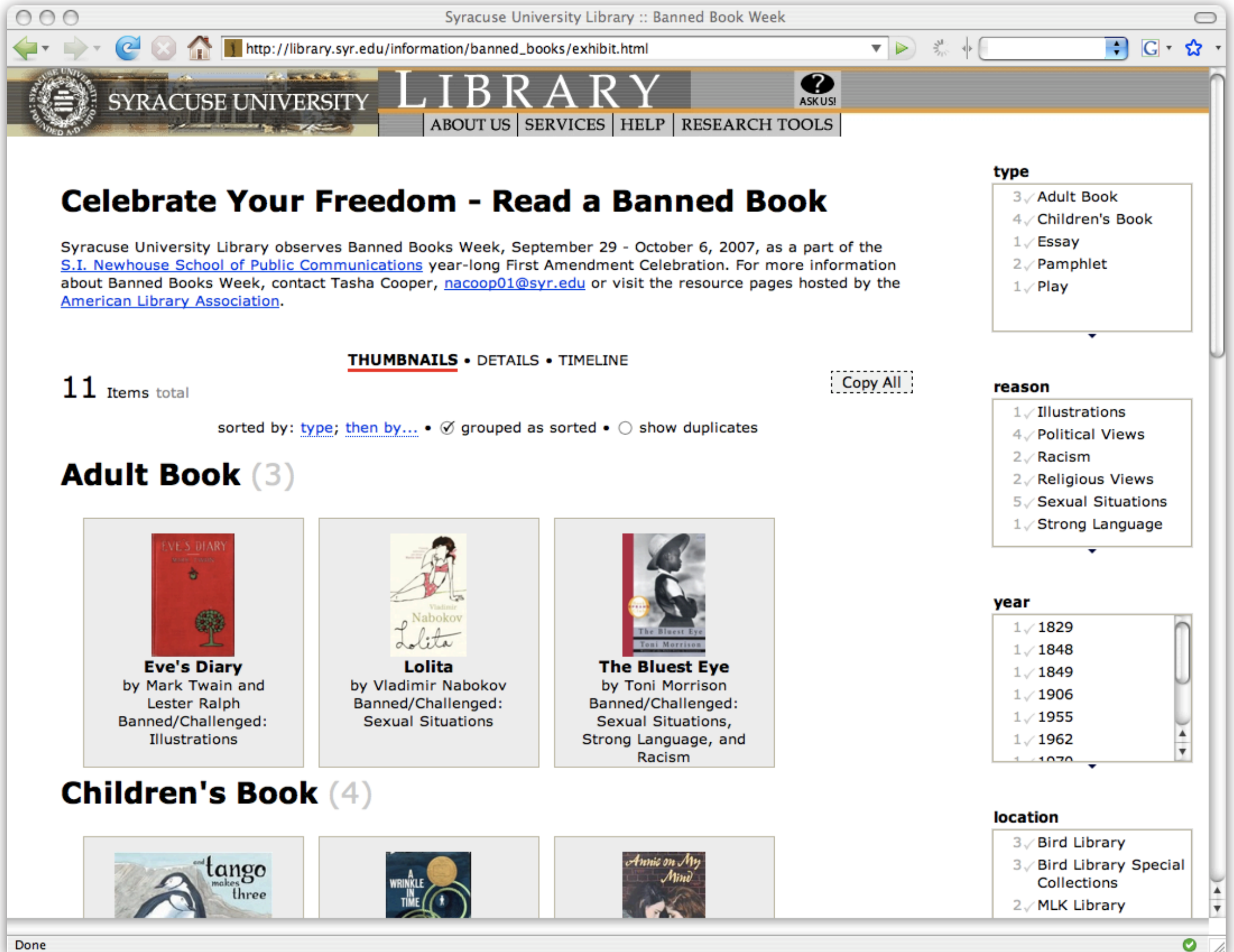
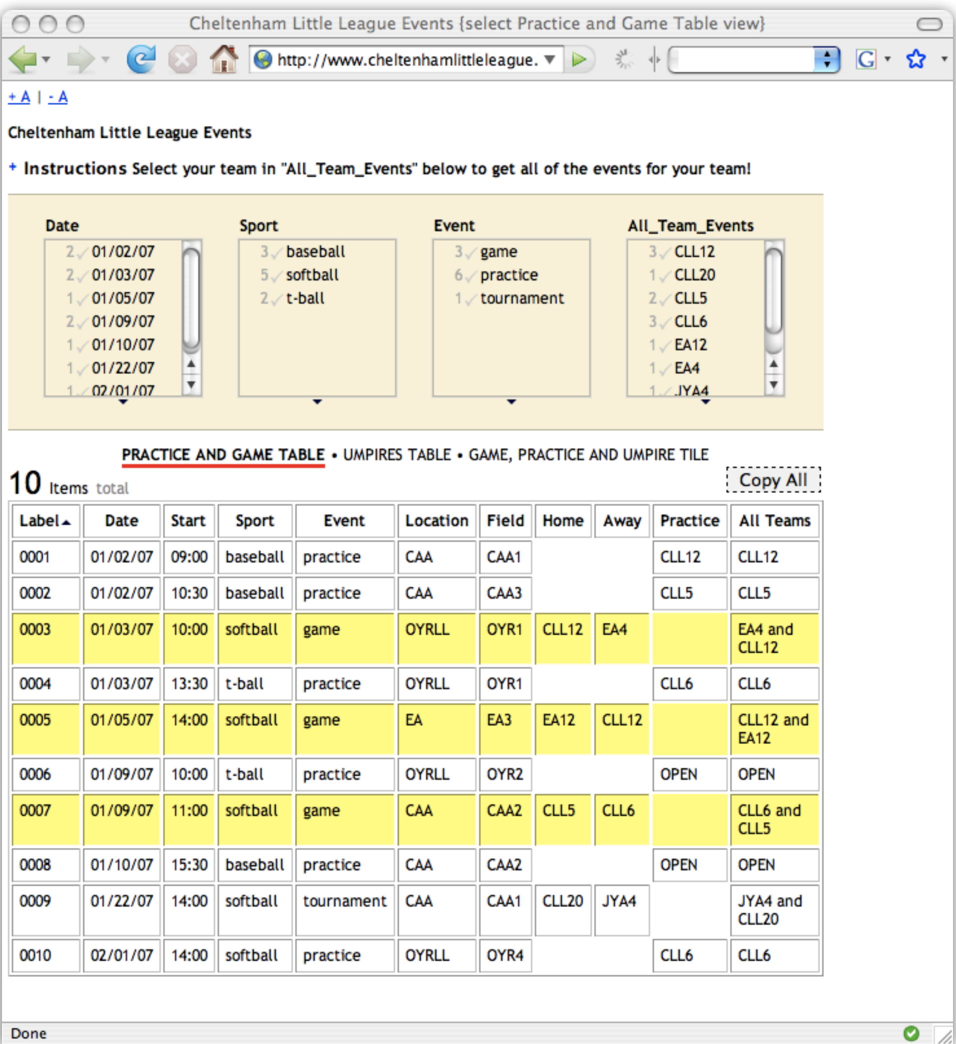


Figure 3.23. An exhibit of 11 banned books. Certain materials may never find their way into official discourse. Empowering end users to publish is even more important a cause.



Label	Date	Start	Sport	Event	Location	Field	Home	Away	Practice	All Teams
0001	01/02/07	09:00	baseball	practice	CAA	CAA1			CLL12	CLL12
0002	01/02/07	10:30	baseball	practice	CAA	CAA3			CLL5	CLL5
0003	01/03/07	10:00	softball	game	OYRLL	OYR1	CLL12	EA4		EA4 and CLL12
0004	01/03/07	13:30	t-ball	practice	OYRLL	OYR1			CLL6	CLL6
0005	01/05/07	14:00	softball	game	EA	EA3	EA12	CLL12		CLL12 and EA12
0006	01/09/07	10:00	t-ball	practice	OYRLL	OYR2			OPEN	OPEN
0007	01/09/07	11:00	softball	game	CAA	CAA2	CLL5	CLL6		CLL6 and CLL5
0008	01/10/07	15:30	baseball	practice	CAA	CAA2			OPEN	OPEN
0009	01/22/07	14:00	softball	tournament	CAA	CAA1	CLL20	JYA4		JYA4 and CLL20
0010	02/01/07	14:00	softball	practice	OYRLL	OYR4			CLL6	CLL6

Figure 3.24. An exhibit for organizing little league sport games. It is filled with acronyms and is tailored for a very small audience, to whom it proves useful.

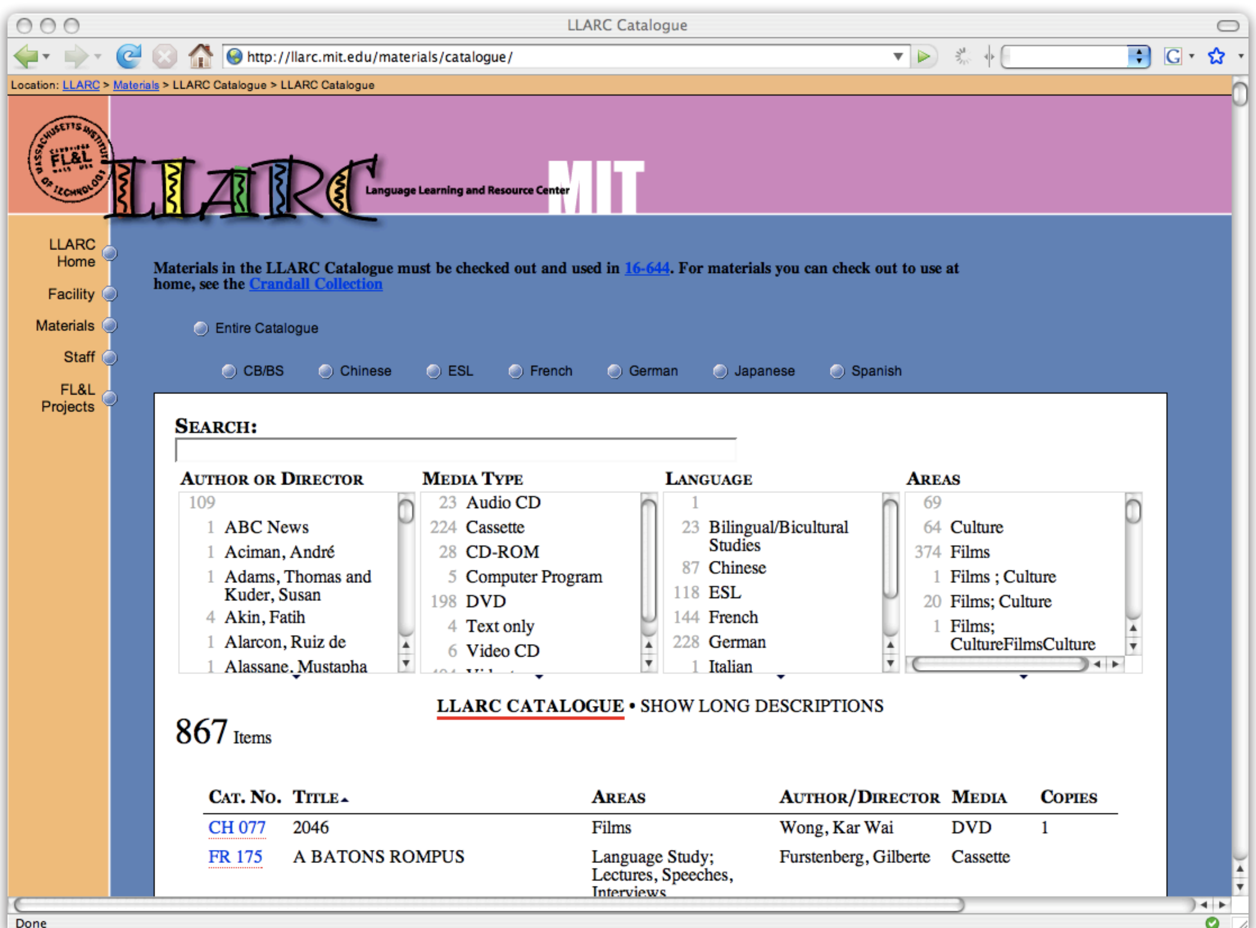


Figure 3.25. An exhibit of 867 teaching resources in the Language Learning and Resource Center at MIT. Its author is skilled in server-side programming but he still finds Exhibit a quick way to build a browsable interface for his data.

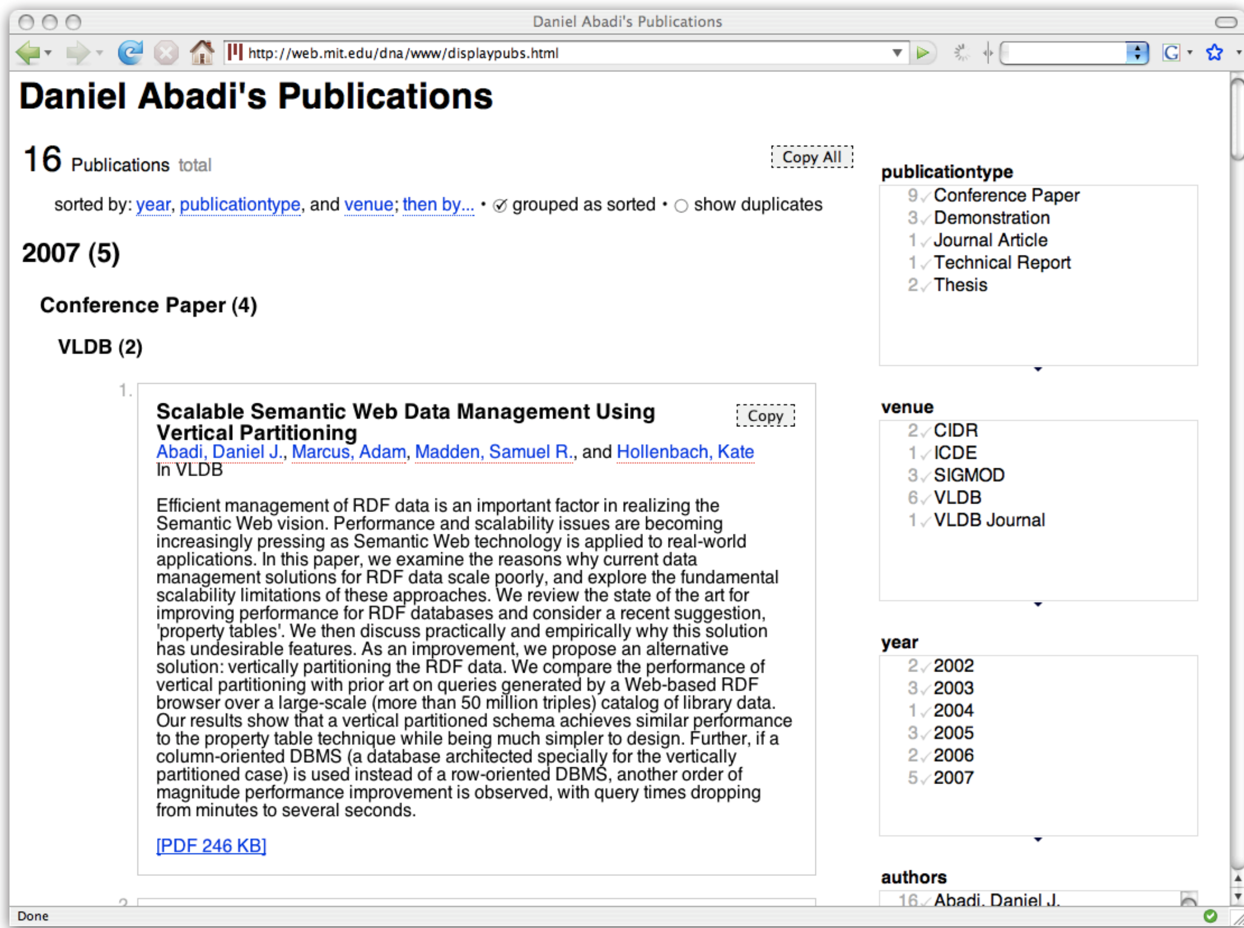


Figure 3.26. An exhibit of a database researcher's publications, who finds it easier to use Exhibit than to build a database-backed web site for his publications.

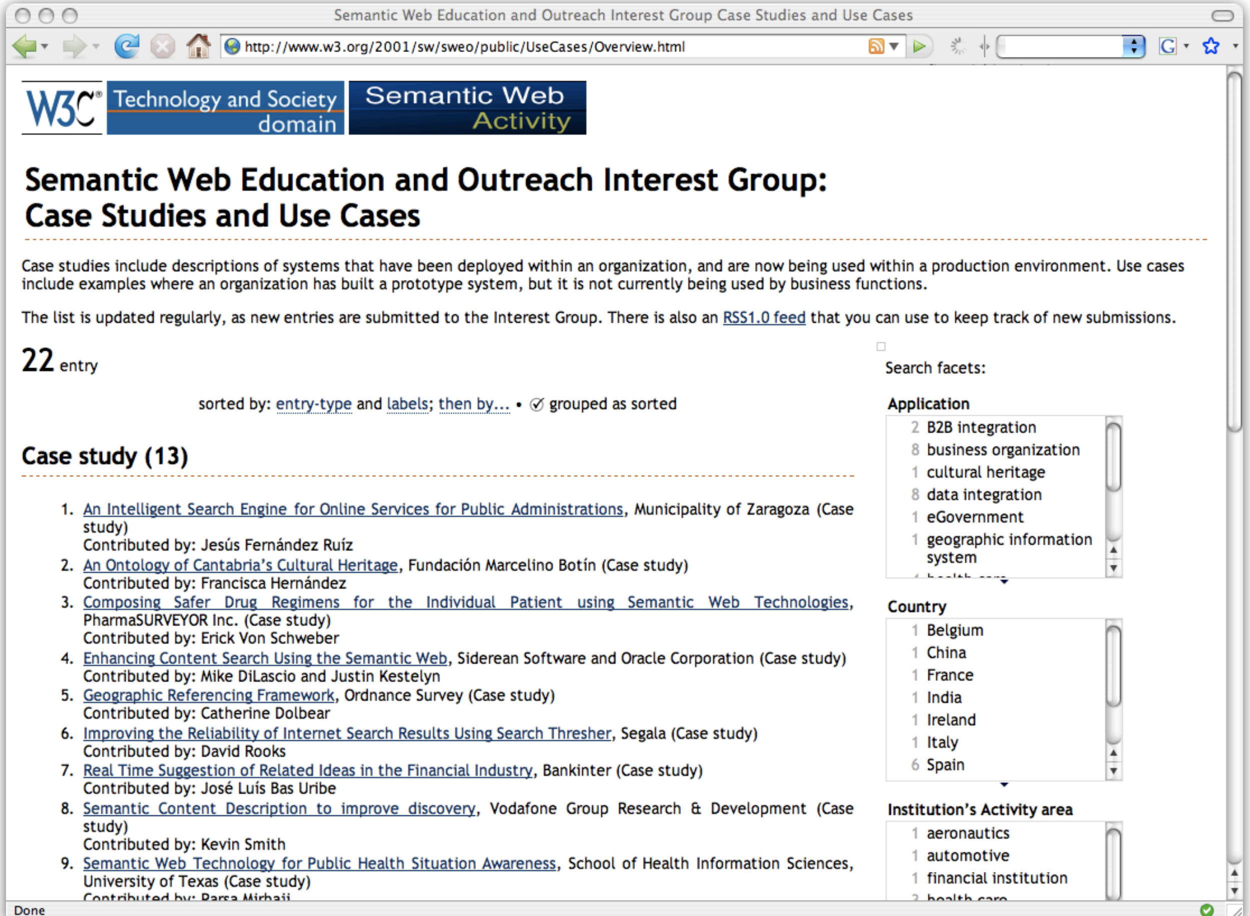


Figure 3.27. An exhibit of Semantic Web case studies and use cases. Exhibit is making Semantic Web data useful by making it viewable by end users.