Fiz: A Component Framework for Web Applications

John K. Ousterhout
Department of Computer Science
Stanford University

Abstract

Fiz is a framework for developing interactive Web applications. Its overall goal is to raise the level of programming for Web applications, first by providing a set of high-level reusable components that simplify the task of creating interactive Web applications, and second by providing a framework that encourages other people to create additional components. Components in Fiz cover both the front-end of Web applications (managing a browser-based user interface) and the back end (managing the application's data). Fiz makes it possible to create components that encapsulate complex behaviors such as Ajax-based updates, hiding many of the Web's complexities from application developers. Because of its focus on components, Fiz does not use mechanisms such as templates and model-view-controller in the same way as other frameworks.

1 Introduction

Although the World-Wide Web was initially conceived as a vehicle for delivering and viewing documents, its focus has gradually shifted from documents to applications. Facilities such as Javascript, the Document Object Model (DOM), and Ajax have made it possible to offer sophisticated interactive applications over the Web. In the future it is likely that more and more of the world's interesting applications will be delivered via the Web, displacing the traditional model of installable binaries.

Unfortunately, creating an interactive Web application today is not an easy task. Some of the problems are inherent in the Web, such as the separation between server and browser, and the numerous languages and technologies that must be combined in a Web application. Other problems come from the development frameworks and libraries available today, which are too low-level. The result is that application developers spend too much time building complex facilities from scratch; there are few high-level reusable components available to Web developers, and it is difficult for developers to carry over code from one application to the next.

This paper describes Fiz, a new server-side framework for Web applications. Fiz' goal is to simplify Web development by raising the level of programming: instead of building from scratch, developers create applications quickly using a library of flexible components ranging from high-level objects such as navigation bars and tree views down to smaller components such as specialized value editors. In addition to its built-in library of components, Fiz provides a framework that encourages the development of new components and the composition of existing components into even larger and more useful structures. We will release Fiz in open-source form and hope to build a user community that creates an ever-increasing set of interesting components, which will make it dramatically easier to create applications that advance the state-of-the-art in Web interactivity.

The rest of this paper is organized as follows. Section 2 summarizes the intrinsic properties of the Web that complicate application development, and Section 3 reviews the most common facilities provided by existing frameworks. Section 4 presents the Fiz architecture, along with the kinds of components it encourages and some of the infrastructure it provides for creating components. Section 5 summarizes the implementation status of Fiz and evaluates Fiz' successes and risks. Section 6 compares Fiz to other frameworks, and Section 7 concludes.

2 Why Web Development is Hard

There are several factors that contribute to the difficulty of building interactive Web applications:

- The functionality of a Web application is distributed, with part running on a server near the application's data and part running in browsers near the application's users.
- Web applications must integrate a complex collection of diverse technologies, including HTML [12], Javascript [5], CSS [8], DOM[8], URLs, XML[1], SQL[13], HTTP, SSL, and one or more server-side languages such as PHP [20], Java [4], Python [11], or Ruby [6]. Different languages are typically used for server and browser programming.
- Web applications must support a variety of browsers with feature sets that are still not 100% compatible.
- Web applications support multiple users with (what appears to be) a single server. This creates interest-

ing opportunities for collaboration, but forces developers to deal with problems related to concurrency and scalability. Popular Web applications must handle 100-1000x the workload of traditional applications.

- Web applications must be highly customizable. In the pre-Web GUI world a uniform look and feel was encouraged, but Web applications strive for unique appearances and behaviors. This makes it more difficult to create reusable components.
- The public accessibility of Web applications introduces a variety of security and privacy issues, and the complexity of the Web development environment makes it easy for unaware developers to create security loopholes such as SQL injection attacks.

Creating a prototype implementation of a simple Web application may seem easy, but it is a much more difficult proposition to develop a sophisticated application that is secure, robust, and scalable. Ideally, a good component set should help to manage the factors described above, but these factors also make it difficult to create such a component set.

3 A Brief History of Web Frameworks

The features of current Web frameworks arose through a series of systems implemented over the last ten years; this section reviews four major developments that explain the current state-of-the-art.

The first Web applications were enabled by the CGI (Common Gateway Interface) protocol. Prior to CGI, Web servers returned only static files. CGI defined a mechanism that maps certain URLs to executable programs instead of files; for those URLs the Web server invokes the executable as a subprocess. The subprocess receives the URL as input, generates an HTML Web page, and writes the contents of the page to its standard output, which is then returned to the requesting browser. CGI programs are written using a variety of languages such as Perl or C++.

The CGI mechanism is *stateless*: a new subprocess is invoked for each incoming request, and the subprocess exits upon completion of the request. If a CGI application needs to maintain state across a series of requests, it must store that information in a file or database and reload it for each new request. The stateless property has become a hallmark of Web applications; although it complicates application development it improves robustness (no state is lost if a server machine crashes between requests) and simplifies scaling (incoming requests can be distributed across a cluster of server machines without worrying about which one holds the state for that request).

CGI programs were quickly replaced by first-generation Web frameworks such as PHP [20] and Java servlets [2]. In these frameworks the runtime system for a particular language is bound tightly to the Web server (to avoid the overhead of starting a new process for every request) and augmented with Web-specific library packages that provide features such as the following:

- Parsing incoming URLs and dispatching to an appropriate piece of code for the URL
- Interfacing to a variety of backend databases and information sources
- Using browser cookies to implement *sessions*, which store the state for an ongoing interaction with a particular browser
- Manipulating information in formats such as HTML and XML.

First-generation frameworks also introduced *templates* to simplify HTML generation. A template is an HTML document that has been annotated with code in the framework's language. To generate a Web page the template is *expanded* by replacing the annotations with HTML computed by the code. The code in the annotations can consist of simple variable substitutions, more complex statements that compute content, or even looping structures that replicate portions of the template (such as the rows of a table). Templates have become the centerpiece of nearly all Web frameworks.

Recent years have seen the arrival of a second generation of Web frameworks. These frameworks provide ORM (Object Relational Mapping) facilities that simplify the use of relational databases in Web applications. One of the best examples of ORM is provided by the Rails framework [21]. Rails associates each database table with a class in the underlying Ruby language, with one object of the class for each row in a table and one field of an object for each column of a row. Rails manages the movement of information between the database and the objects, so Web developers can think in terms of the Ruby classes rather than SQL. Rails also manages relationships between database tables so that they appear as convenient references between collections of Ruby objects rather than foreign keys. Similar ORM facilities have appeared in other second-generation frameworks such as Django [3], and ORM libraries have been added to some first-generation frameworks.

Second-generation frameworks also introduced a stylized division of labor for Web page generation based on the model-view-controller (MVC) paradigm [16,17]. *Model* classes manage back-end data using the framework's ORM mechanism. *Views* are templates that generate the final Web page. *Controllers* provide the glue that ties the pieces together. For example, in Rails each incoming URL is dispatched to a controller, which invokes models to fetch data for a Web page, then invokes a view to generate the page's HTML.

Concurrently with the server-side developments described above, an assortment of Javascript library packages has evolved for use in writing code for the browser. The initial motivation for such libraries was to hide incompatibilities between browsers, but over time they have incorporated numerous additional facilities such as:

- Extensions to the DOM (Document Object Model) to make it easier to modify HTML documents in the browser
- Extensions to the Javascript object model
- Communication with the server (e.g., using the Ajax mechanism)
- Animations and other dynamic effects
- Higher-level user interface controls, such as a calendar or rich-text editor

Some examples of popular Javascript packages are Prototype [14], Scipt.aculo.us [19], Jquery [10], and YUI [22]. Google's GWT framework also supports the development of higher-level controls for the browser, using an approach where developers write in Java, which is then translated automatically to Javascript [9].

Although current frameworks and library packages provide a vast improvement over the CGI approach, the server-side facilities are still relatively low-level: their abstractions look more like the Web's raw materials (HTML, Ajax, SQL, etc.) than the applications they are used to create. Some browser-side frameworks provide higher-level components, but these components exist only on the browser; it is up to application developers to integrate the browser components with server-side facilities.

4 Fiz Architecture

Fiz is a Web development framework written in Java that runs on Web server machines. Its overall goal is to raise the level of programming for Web applications so that developers can spend more time thinking about the structure and behavior of their applications and less time dealing with arcane details of the Web. Fiz' approach is to encourage the creation of high-level components: it contains a collection of library procedures

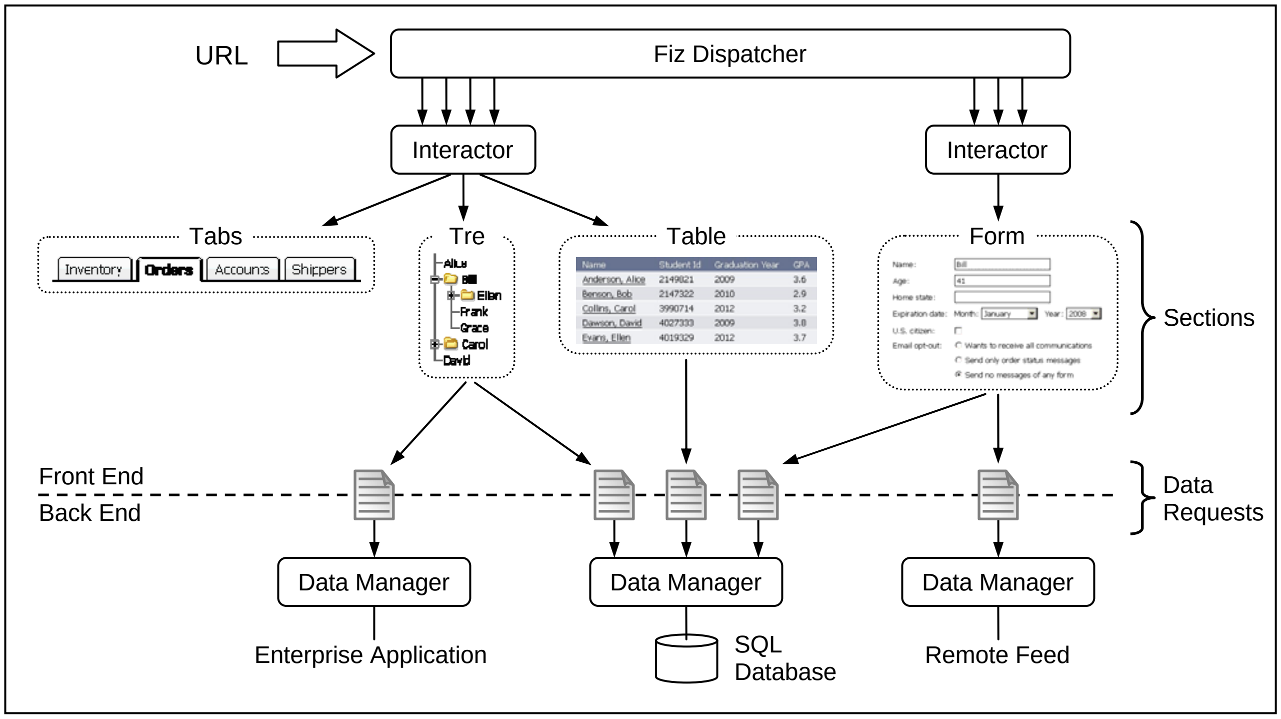


Figure 1. A Fiz application consists of a front end, which services browser requests and manages the application's user interface, and a back end, which manages the application's data. The primary components for the front end are *sections*, which implement specialized chunks of user interface, such as a form or a collection of tabs. Each incoming request for a URL is dispatched by Fiz to an *interactor* object; the interactor creates a Web page by assembling sections. The back end of an application consists of one or more *data managers*, each of which provides access to a particular kind of information. Data requests provide asynchronous communication between sections and data managers.

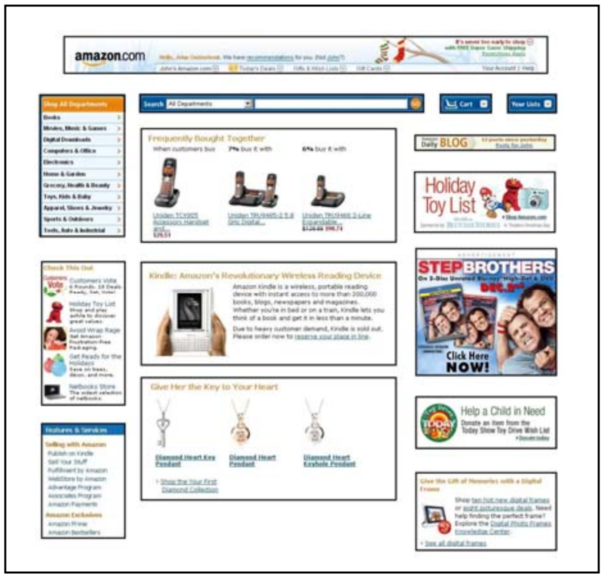


Figure 2. An example Web page, exploded to display one possible decomposition into sections.

that provide a framework for developing and integrating components, and it also contains an initial set of components.

Figure 1 shows the overall structure of a Web application based on Fiz. Fiz applications divide into two major parts: a front-end, which handles browser communication and creates the application's user interface, and a back-end, which manages the application's data. Each of these parts offers opportunities for interesting components. Sections 4.1 and 4.2 discuss front-end components and Section 4.3 discusses back-end components. Sections 4.4 and 4.5 describe additional Fiz APIs to simplify the creation and integration of components. Section 4.6 shows how components can encapsulate complex behaviors such as dynamic Ajax requests, and Section 4.7 describes the ways in which Fiz allows components to be customized.

4.1 Sections

Unlike most Web frameworks, Fiz is not organized around templates (though it does have a "tiny template" mechanism that is discussed in Section 4.5). Whereas templates encourage designers to start by thinking about HTML, Fiz encourages designers to focus on the high-level structure of a Web page using *section* objects. A section represents a self-contained element of a page such as a table, a form, a navigation bar, or an embedded advertisement. Some sections may be special-purpose, such as an application-specific navigation bar; others, such as forms, tables, and tab sets, can be parameterized to support a variety of uses. For the most part HTML is generated by section objects and therefore invisible to developers. Figure 2 shows an example of how a Web page might be broken down into sections.

Each kind of section is implemented by a Java class. A developer instantiates a section by providing a collection of *configuration properties* (name-value pairs) containing overall parameters for the section. In addition, many sections contain one or more subcomponents (either other sections or smaller objects), which the developer also provides.

Figure 3 illustrates how a simple section can be instantiated in Fiz, as well as the way in which different Fiz components work together to generate HTML for the section. Figure 3 creates an instance of TableSection, one of the built-in section classes defined by Fiz. The code in Figure 3(a) specifies a single configuration property (`request`), whose value (`getStudents`) indicates a source of data for the section. The code also creates a Column object for each column in the table. Column is a Fiz class with two methods: one to generate HTML for the column's header and another to format data for that column in each row of the table's body. The TableSection collects the data for the table using the mechanism described later in Section 4.2, creates the overall HTML table structure, and provides conveniences such as error handling and HTML `class` attributes to enable odd-even row displays and other CSS effects. Then for each row of the table TableSection invokes the Column objects in turn to provide the HTML inside the `<td>` element for that column. For the non-header rows, TableSection passes all of the data for that row to the Columns; each Column extracts the particular data values that it needs. In simple cases only a single value is needed for a particular column, but the first column in Figure 3 uses three data values.

The components used in Figure 3 are reusable and customizable in several ways:

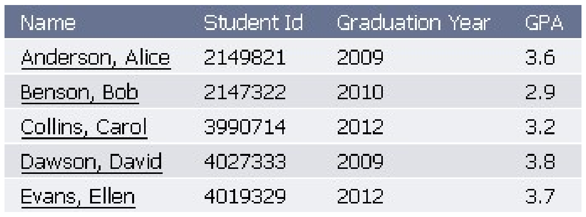
- The TableSection class can be used for a variety of tables with different data sources, different numbers of columns, and different formatting of individual columns.
- This mechanism supports a variety of different column displays by defining classes such as Link, which format values for the body of the table. For example, the Checkbox class extracts a specified value from the input data, treats it as a Boolean value, and displays one of two images: an empty box (for false) or a checked box (for true).
- Classes such as Link or Checkbox can be used in other sections besides just TableSections: they are examples of *formatters*, which use a record contain-

```
        new TableSection(
            new Dataset("request", "getStudents"),
            new Column("Name", new Link("@last, @first", "student?id=@id")),
            new Column("Student Id", "@id"),
            new Column("Graduation Year", "@grad"),
            new Column("GPA", "@gpa"))
```

(a)

id	first	last	grad	gpa
2149821	Alice	Anderson	2009	3.6
2147322	Bob	Benson	2010	2.9
3990714	Carol	Collins	2012	3.2
4027333	David	Dawson	2009	3.8
4019329	Ellen	Evans	2012	3.7

(b)



Name	Student Id	Graduation Year	GPA
Anderson, Alice	2149821	2009	3.6
Benson, Bob	2147322	2010	2.9
Collins, Carol	3990714	2012	3.2
Dawson, David	4027333	2009	3.8
Evans, Ellen	4019329	2012	3.7

(c)

Figure 3. An example of a Fiz section: (a) Java code to construct the section; (b) the data for the section as returned by the `getStudents` request, consisting of name-value pairs for each row; (c) the appearance of the section in the resulting Web page. Configuration properties for the section are provided using a Dataset object containing name-value pairs (see Section 4.4). Each Column generates HTML for one column of the table. For example, the last Column will display "GPA" in the header row, and for each body row it will extract the value named `gpa` from the data for the row and display that value (the string "`@gpa`" is a simple form of template where "`@`" causes variable substitution; see Section 4.5). For the first column of the table a Link object generates the HTML for the body rows; its arguments specify two templates, one for the column's text and another for a URL that will be visited when the text is clicked.

ing name-value pairs to generate HTML in a class-specific fashion.

Figure 3 illustrates Fiz' goal of providing components that look more like the finished application than its raw materials. The TableSection is defined by describing how the data for the table is mapped to columns and how the individual columns are displayed. There is no HTML in Figure 3 except for the URL template for the Link. HTML is generated by the various components, and Fiz automatically handles issues such as quoting HTML special characters that might appear in data values (see Section 4.5).

Fiz currently provides six built-in section classes; see Table 1 below.

4.2 Interactors and Dispatching

In a Fiz application Web pages are created by assembling collections of sections; this task is performed by *interactors*. Interactors are the top-level classes that manage interactions with the browser. Each interactor handles a collection of related URLs, such as those for viewing and editing inventory, or those for posting and viewing a blog. For example, a blog-management interactor might support URLs for viewing blog entries, searching a blog, posting a new entry, or commenting on an existing blog entry. The URLs handled by an interactor may include both those for generating normal HTML pages and those for responding to Ajax requests. In many ways Fiz interactors are similar to con-

CompoundSection	Manages a group of child sections; includes a general-purpose layout mechanism to describe arrangements of sections, and optionally displays a custom frame around the entire group.
FormSection	Displays a collection of FormElements, each of which allows a particular item in a record to be edited. Also handles form posting and error handling (error messages are displayed inline in the form).
TabSection	Displays a collection of tabs, with control over what happens when a tab is clicked (display a new URL, invoke an Ajax request, or invoke a Javascript statement).
TableSection	Displays data in a two-dimensional array.
TemplateSection	Displays literal HTML strings and simple templates.
TreeSection	Displays a hierarchical structure that can be expanded and collapsed dynamically by clicking on "+" and "-" boxes; uses Ajax requests to expand nodes.

Table 1. The section classes currently implemented by Fiz.

troller classes in Rails.

For each URL handled by an interactor there is a method in the interactor called an *entry method*. The entry method generates a Web page by creating a collection of sections and passing that collection to a method in Fiz called `showSections`. `ShowSections` renders the page using a multi-pass algorithm described in Section 4.3 below.

Fiz is responsible for dispatching URLs to interactor entry methods; these URLs have the form

`/fiz/`*class*`/`*method*`/`…

where *class* specifies an interactor class and *method* specifies a method within that class. The Fiz dispatcher uses Java reflection to construct an instance of the class (if it hasn't been referenced in a URL already) and invoke *method*. Any public method of an interactor class is automatically accessible through a URL if it has a signature appropriate for an entry method; there is no need to create a configuration file for dispatching.

4.3 Data Managers and Data Requests

The notion of separating the management of data in an interactive application from the management of its user interface has been widely accepted since the original model-view-controller proposal for Smalltalk [16,17]. This separation makes sense because the same data may be presented and manipulated in several different ways, and because the two parts lend themselves to different developers. Most Web frameworks incorporate this separation, and so does Fiz.

However, data management in Fiz differs from other Web frameworks in two significant ways. First, most frameworks are biased towards relational databases as the back-end source of data; they provide little or no support for other data sources and may not even handle multiple database instances gracefully. Fiz uses a back-end framework that encourages a variety of data sources. Second, most frameworks use synchronous mechanisms that serialize access to data. This approach is fine for simple applications, but complex applications such as Amazon.com or Facebook can use data from dozens or hundreds of sources in single page; they require parallel execution of data requests to get adequate performance. Fiz uses an asynchronous approach to data management, which allows requests to execute in parallel.

Back-end components in Fiz are called *data managers*. A data manager is a Java class that provides access to a particular kind of data, such as a relational database, a remote data feed, an existing application, or an application-specific file format such as Excel spreadsheets. Some data managers manage their data directly while others provide an access layer for data managed by a remote service or an external application on the same machine. Data managers can be layered, with one data manager creating additional functionality on top of lower-level functions provided by one or more other data managers. There can be multiple instances of the same kind of data manager, such as multiple data managers for SQL databases, each providing access to a different database instance.

A data manager is invoked using a *data request* object that provides a rendezvous point between caller and callee. A data request is initialized with a set of name-value pairs that specify a particular data manager, an operation for the manager to perform, and any additional parameters needed by the operation. Once the operation has completed the data manager adds result information to the data request; or, if the request failed then the data manager adds error information to the data request. The caller uses the data request to access the results or error information. The dataset facility described in Section 4.4 is used for representing the parameters, results, and error information in a data request.

Although data requests can be created and invoked synchronously in Fiz, they are typically used in an asynchronous fashion as part of a three-pass mechanism for rendering Web pages. Recall from Section 4.2 that a Web page is typically generated by an interactor, which creates a collection of sections describing the page and then passes the collection to the `showSections` method in Fiz. `ShowSections` renders the page using the following approach:

1. `ShowSections` makes a pass through all of the sections, invoking the `registerRequests` method in each section. `RegisterRequests` creates data requests for any data needed by that section. Request processing is not initiated at this point; the data requests are collected in a pool associated with the Web page. The section retains references to its requests for use later to retrieve the results.
2. Once every section has had a chance to register its requests, `showSections` initiates processing on all of the requests by passing them to their respective data managers. At this point each data manager has a choice: either it can process the request immediately and synchronously, adding result or error information to the request and marking the request complete, or it can return immediately and complete the request in the background (e.g., by passing the request to a separate thread for processing). Handling requests asynchronously adds complexity to

3. `ShowSections` makes another pass through all of the sections, this time invoking the `html` method for each section. `Html` generates the HTML for the section, along with any Javascript, CSS, or other information needed for the section. When the `html` method attempts to retrieve result or error information from a data request, Fiz checks to see whether the request has completed; if not, the rendering thread stalls until the data manager marks the request complete.

With this approach developers can work in a world that appears almost entirely sequential, yet gain the benefits of concurrency between data managers. Interactors and sections are always invoked sequentially, so their developers need not worry about concurrency issues. At the same time, all of the data requests can be handled in parallel (assuming the data managers are written to be asynchronous), and this is typically the most time-consuming part of generating Web pages. Data manager developers are the only ones who have to deal with concurrency issues, and the concurrency model for data managers is relatively simple, with most concurrency issues (such as synchronization over request completion) handled automatically by Fiz.

In comparison to the synchronous approach used in other Web frameworks, Fiz' mechanism for data management improves performance in three ways:

- Requests can be executed in parallel.
- If there are multiple requests for the same data manager, the data manager receives them together in a batch. This may allow the data manager to process them more efficiently: for example, it can package them together in a single message exchange with a remote server, rather than using separate message exchanges for each request.
- If the same data request is made by multiple sections (not an unusual occurrence) the requests are merged into a single request that is shared by the sections. This avoids redundant processing of the same request, which would happen in a traditional synchronous approach.

Although Fiz' approach can usually handle data requests in parallel, there are some situations where data requests must be serialized. For example, the section structure of a page may depend on user preferences stored in a database. In this situation the interactor will first have to issue a synchronous data request to retrieve the structural information. Once it has this information, it can create the sections for the page; the remaining data requests will then proceed in parallel.

4.4 Datasets

In order for a framework to allow a variety of components to work together, it must provide a communication mechanism between the components that is generic and extensible. The mechanism must be generic in the sense of being simple and uniform, so that any component supporting the mechanism can be combined easily with any other component supporting the mechanism. The mechanism must be extensible in the sense that it must allow components with a deeper understanding of each other to communicate in a more specialized fashion, without interfering with the more generic communication supported universally.

As an example of such a generic and extensible mechanism, consider pipes in the Unix operating system [18]. A pipe is a unidirectional stream of bytes from one process to another. Unix uses pipes as a mechanism for interconnecting processes and makes it easy (using various shell programs) to create collections of processes that communicate via pipes. Pipes are generic in that they are easy to support and place almost no limits on the kind of information that can be transmitted. Pipes are extensible in that readers and writers can impose additional structure on the information in the pipe. For example, most applications assume that information is textual, and many applications treat the text as line-oriented, but other applications make no interpretation whatsoever of information transmitted via the pipe. Unix's pipe mechanism is one of the system's most powerful features.

Fiz uses a structure called a *dataset* for communication between components. In its simplest form a dataset is a collection of name-value pairs such as the following:

```
    state:       California
    capital:     Sacramento
    flower:      poppy
    2000census:  33,871,648
```

Both names and values are arbitrary strings; a dataset may contain any number of entries or none at all. The value of an entry in a dataset can also be a nested dataset or an array of nested datasets (see Figure 4); this allows datasets to represent complex hierarchical structures. Internally, datasets are stored as hierarchical hash tables; externally they may be stored in a variety of formats such as XML, YAML, or Javascript literals (datasets only use a subset of the features of these external formats).

Datasets are used for numerous purposes in Fiz, including the following:

- The parameters for each data request are specified with a dataset.

```
name:    Alice
age:     24
height:  64
courses:
  - name:      Math 104
    semester:  Fall 2007
  - name:      English 208
    semester:  Spring 2008
```

(a)

```
Dataset d = new Dataset(
    "name", "Alice",
    "age", "24",
    "height", "64",
    "courses", new Dataset(
        "name", "Math 104",
        "semester", "Fall 2007"),
    "courses", new Dataset(
        "name", "English 208",
        "semester", "Spring 2008"));
```

(b)

Figure 4. A dataset represents a hierarchical collection of name-value pairs, where each name is a string and each value is either a string, a nested dataset, or an array of nested datasets: (a) an example dataset in its YAML external representation; `name`, `age`, and `height` have string values, while the value for `courses` is an array of two nested datasets, each with `name` and `semester` entries; (b) the same dataset defined in Java.

- The response for each data request is a dataset. For example, the response for a TableSection must contain an entry named `record` whose value is an array of nested datasets, each containing the data for one row of the table.
- If a data request fails, it returns error information in a dataset. There will always be a `message` element containing a human-readable message, but some errors also return additional values that may be useful in handling the error, such as a `culprit` value identifying a particular form field that was invalid.
- Datasets provide configuration properties for sections and other components. This is more convenient than an ordered list of parameters for the constructor, because the components typically support a large number of optional properties of which only a few are used in any given instance. Using a dataset allows the uninteresting properties to be omitted, and the name-value syntax helps to clarify the properties that are supplied. (An implementation language offering a keyword-value form for method arguments might make this use of datasets unnecessary.)
- When an incoming URL contains query values, such as `/a/b/c?id=41243&color=red&size=M`, Fiz makes the query values available in a dataset. Incoming data from posted forms is handled in the same way.
- Datasets are used to pass structured data between the server and browser using the reminder mechanism described in Section 4.6.
- Configuration information for Fiz is stored on disk as datasets.

One disadvantage of Fiz datasets is that the Java language forces a somewhat verbose notation for defining datasets. Scripting languages such as Javascript tend to offer dataset-like features with a more concise syntax.

4.5 Templates

Although Fiz does not encourage the use of templates as an overall structuring mechanism for Web pages, it does provide a "tiny template" facility that is useful for generating HTML inside sections and other components. Generating a Web page requires a considerable amount of string manipulation, typically combining computed values with static HTML, Javascript, URLs, etc. The Fiz template mechanism provides two benefits for such string manipulation: first, its concise syntax reduces coding and makes it easier to visualize the final strings that will be generated; second, it automatically quotes substituted values in the appropriate fashion for the output being generated.

Templates in Fiz behave roughly the same as templates in other Web frameworks. A template is *expanded* by combining it with a dataset to produce a new string. Most characters in the template are simply copied to the output string, but a few patterns cause special processing; for example, if `@id` occurs in the template, then the value of the `id` entry in the dataset is added to the output string instead of `@id`. Fiz templates support only a small set of common substitutions, which were chosen by examining an existing Web application to identify the most common idioms; see Table 2.

Fiz templates are intended for generating small snippets of HTML, Javascript, etc. This usage resembles the format strings used for the C `printf` function more than it resembles templates in other Web frameworks, which are larger objects stored in separate files. Figure 5 shows an example of typical template usage and illustrates the conciseness/visualization advantage of templates over traditional I/O mechanisms.

When Fiz substitutes a value in a template it automatically quotes characters in the substituted value that

Pattern	Action
@*name*	Insert the value specified by *name* into the output string.
@(*name*)	Insert the value specified by *name* into the output string. Useful when *name* is immediately followed by alphabetic characters.
{{*template*}}	If any of the substitutions in *template* references nonexistent data then do nothing; otherwise expand *template*.
@*name*?{*template*}	If *name* exists then insert its value into the output string; otherwise expand *template*.
@*name*?{*template1*|*template2*}	If *name* exists then expand *template1*, otherwise expand *template2*.

Table 2. When a template is expanded each character from the template string is copied to the output string, except for a few patterns that cause substitutions. If one of the patterns on the left occurs in the template string, its characters are not copied to the output; instead, the corresponding action on the right is taken.

have special meaning in the output. For example, when generating HTML a `<` character in a substituted value will be converted to the HTML entity `&lt;`, which the browser will then display as `<`. Without the quoting the browser would interpret the `<` as the beginning of a tag. This feature spares developers from adding extra code to quote special characters, which is tedious and a common source of errors (forgetting to quote special characters can lead to problems such as SQL injection attacks).

Although Fiz quotes special characters in substituted values, it does not quote characters in the template string. This distinction works well in Fiz, since templates almost always contain meta-characters such as < that need to retain their special significance in the output, while substituted values almost always contain raw data that should be treated as literal. Automatic quoting is more difficult in other templating systems because the templates are used in a more general fashion where substituted values often contain special characters that must not be quoted.

The Fiz templating mechanism is used for more than just generating HTML. The same mechanism is also used for generating Javascript, URLs, CSS, and SQL statements, each of which has its own special characters and quoting mechanisms. The template methods provide an optional argument to identify the output format, and different quoting rules are applied for each format. For example, consider the task of generating an HTML tag containing a Javascript event handler: one template is expanded to generate the Javascript code (using Javascript quoting), then the result of this template is

```
Template.expand("<table id=\"@id\" {{class=\"@class\"}}>", properties, html);
```

(a)

```
html.append("<table id=\"");
html.append(properties.get("id"));
String value = properties.check("class");
if (value != null) {
    html.append(" class=\"");
    html.append(value);
    html.append("\"");
}
html.append(">");
```

(b)

Figure 5. Templates provide a concise mechanism for substituting dynamic values into formats such as HTML: (a) Java code to generate an HTML tag using a Fiz template; (b) code that generates the same result using standard Java string facilities. In each case a `<table>` tag is added to the `html` string, substituting the `id` value from a dataset named `properties`; if there is a `class` value in the dataset then a `class` attribute is included in the tag. In addition to reducing code size, the implementation using a template makes it easier to visualize the string that is generated. Furthermore, the code in (b) does not quote special characters in substituted values, whereas this is handled automatically by the template.

used as a substituted value in a second template to generate the HTML tag (using HTML quoting for any special characters in the Javascript).

4.6 Encapsulating Ajax: Reminders

For a component framework to succeed it must support *encapsulation*: a developer must be able to use a component in a Web page without understanding details of its implementation. This allows components to hide complexity from developers. One example of encapsulation in Fiz is Javascript. Many components require Javascript to be downloaded to the browser to give them dynamic behavior; this can be done in Fiz without any awareness of the Javascript outside the component.

A more challenging encapsulation issue arises from Ajax requests. Ajax is a protocol that allows Javascript in the browser to issue an HTTP request to the server and use the results to modify the current page [7]. For example, consider the TreeSection component in Fiz, which displays a hierarchical structure and allows the user to expand and collapse portions of the structure by clicking on them. In order to handle large trees efficiently, the tree component initially displays only the top level of the structure; when the user clicks on an unexpanded node, the browser issues an Ajax request that returns the children of that node for display. Ideally it should be possible for the tree component to encapsulate its Ajax requests: a Web developer should be able to use the component in a page without knowing that the component makes Ajax requests and without having to write any code to deal with the Ajax requests.

The first problem with Ajax requests is dispatching. One approach is for the interactor that generated the page to receive any Ajax requests for the page and pass them on to the appropriate component. Rails encourages an approach similar to this in its controllers, but in so doing it forces the page developer to be aware of all of the Ajax requests related to any of its components or their subcomponents; as pages become more dynamic, with more Ajax requests, this creates increasing complexity for the page developer.

In Fiz a component can arrange for Ajax requests to be dispatched directly to it, bypassing the interactor that generated the page containing the component. This allows a component to encapsulate the handling of its Ajax requests so that they are not visible to the page developer.

However, dispatching directly to a component introduces a problem with state management. In order for a component to handle an Ajax request it needs access to the parameters used earlier to create the component, such as the data request for retrieving the contents of a node. This information was passed to the component by the interactor at the time the page was rendered, but that information is no longer available: it was discarded after the original page was rendered. If a component is to encapsulate its Ajax requests, it must have a mechanism for saving its state between the original page display and later Ajax requests.

One approach is for the component to record its state on the server, indexed by an identifier for the particular instance of the component. The instance identifier is embedded in the Web page and returned with the Ajax request, allowing the component to locate the saved state. The problem with this approach is that the server cannot tell when the saved state is no longer needed: the user could return to an old page at any time in the future (using the “Back” button) and cause an Ajax request. Thus, the server would have to retain the state for all instances of all components for the life of the browser session.

Instead of recording state on the server, Fiz uses a mechanism called *reminders*, where component state is passed to the browser and then returned with future Ajax requests. A reminder is a dataset that can be moved back and forth between the server and the browser. Fiz provides facilities that make it easy for a component to define reminders and arrange for them to be returned to the server in later Ajax requests; Fiz handles the details of serializing reminders, transmitting them to the browser as part of a Web page, storing them in the browser, and returning them to the server in the Ajax requests. When an Ajax request is dispatched later to the component, each of the request’s reminders is available to the component as a dataset. For example, when TreeSection issues an Ajax request to expand a node, the request includes two reminders: one contains overall information for the tree and the other contains information specific to the node being expanded.

Reminders solve the state management problem for components, but they introduce potential security issues: a rogue browser can read the contents of reminders and can potentially forge reminders. For example, the reminder for a tree control contains information describing the data request to issue when expanding a node: without any protection, a rogue browser could forge this information, thereby issuing any data request supported by the server. To prevent this sort of attack Fiz signs each reminder with an SHA-256 cryptographic checksum based on the session identifier. This prevents browsers from forging a reminder or using a reminder in a session other than the one for which it was defined. Browsers can still read the contents of reminders; we plan to add encryption to the reminder mechanism for cases where reminders contain sensitive information.

Reminders are similar in many ways to cookies, which are passed back and forth between the browser and the server to manage Web sessions. However, cookies contain information that is global to a session, whereas reminders carry information that is local to particular sections in particular pages. Reminders are also similar to “viewstate” information in the ASP.NET framework.

4.7 Customization

Web applications strive for unique appearances and behaviors. If an application is going to use pre-existing components, it must be able to customize those components to implement the application's desired look and feel. One risk for a component framework is that the components will not enable a sufficient degree of customization to meet the needs of applications, or that the effort to customize existing components might be comparable to the effort required to build the application from scratch.

One of the goals for Fiz is to provide a variety of easy-to-use mechanisms for customization. Here are a few examples of mechanisms available today:

Configuration properties: as described in Section 4.1, components can be customized with a variety of parameters.

CSS: Fiz components add `class` attributes to the HTML elements they generate, which application developers can use in CSS stylesheets to modify the appearance of the components. For example, TableSection adds `odd` and `even` classes to the `<tr>` elements for table rows, which can then be referenced in CSS to display striped backgrounds for the table. Unfortunately this form of customization expands the interface of the components by exposing some of the details of the HTML generated by the components.

Templates: templates provide a convenient mechanism for a variety of customizations. For example, the FormSection displays error information next to form elements that are invalid; the application developer can provide a template for the error message, which is expanded in the context of a dataset containing details about the error; this allows the developer to control what information is displayed as well as the precise HTML formatting of that information. Another example is the TreeSection component: the developer can provide a template for displaying a particular node, which allows the developer to select the information to display for the node, one or more icons to display next to the node, and so on.

Image families: some components use a family of images to create their appearance; for example, the TabSection class uses several images to give the notebook tabs their appearance and to differentiate the selected tab. A developer can create an alternate family of images to change the appearance, and supply the base name for this family as a configuration property. We will encourage Fiz users to contribute their custom image families, which we will then distribute with Fiz to increase the set of appearances available “out-of-the-box”.

Although Fiz components support customization, they do not require it: components have default appearances and behaviors that should be adequate for many purposes.

5 Status and Evaluation

Implementation of Fiz began in January 2008; by September 2008 it had enough functionality for us to begin using it in the development of a test application (a community Web site for Fiz). The application has exposed several shortcomings and missing features, many of which have since been addressed. We are continuing to develop both Fiz and the community Web site in parallel, using the Web site as driver for Fiz evolution.

We will release Fiz in open-source form in the summer of 2009, and we will continue to develop Fiz based on input from users. The community Web site, which will become available at the same time as the open-source release, will allow Fiz users to contribute, share, and rate additional components. Our longer-term goal is to stimulate the creation of a large and robust set of components for interactive Web applications. We hope that Fiz will encourage contributions on several levels: in addition to creating new sections and data managers, users can contribute components for use inside sections, such as form elements or columns for tables, and families of images for customizing existing components.

To measure the benefits of using components, we implemented three Web page fragments twice, once using Rails templates and once using Fiz components. Figure 6 shows the results: the component-based approach reduced code size for each fragment by 5-10x. The examples in Figure 6 are relatively simple ones; as more complex components are developed, it seems likely that the gap between components and templates will grow.

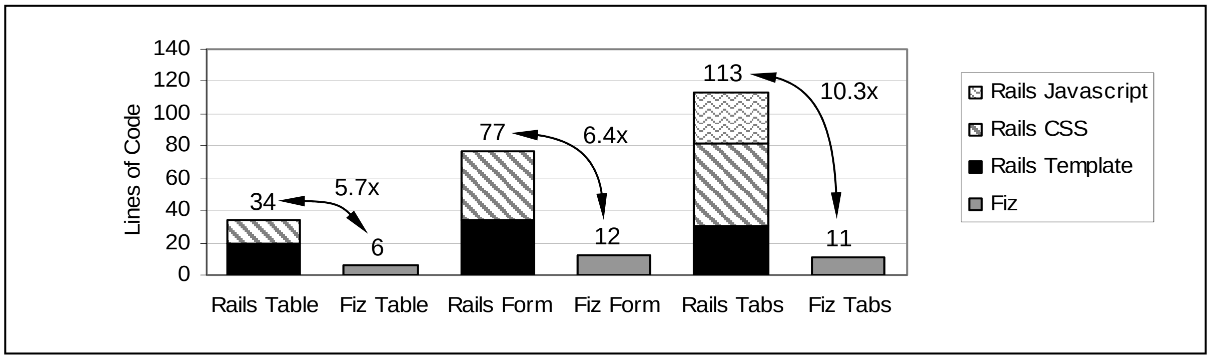


Figure 6. A comparison of Fiz components and Rails templates for implementing three sections: the table from Figure 3, a form with five text entries, and a tab set with five tabs. The line counts for Rails include the template plus any associated CSS and Javascript. The numbers for Fiz count lines of Java code to construct the section using existing Fiz components.

Although these small-scale comparisons are encouraging, the most important questions have to do with the scalability of the Fiz approach. For example, is there a large space of interesting reusable components that can be created? Can new components be created by composing existing components? How well will Fiz work for large applications: can most of the applications' functionality be created with existing components, or will each application require the creation of a large number of new components? Are the components sufficiently customizable to meet the needs of real applications? We hope that the open-source release of Fiz will lead to wide enough usage of Fiz to answer these questions.

Another risk to the success of Fiz is complexity: for Fiz to succeed it must reduce the complexity of Web application development. The Fiz components help by encapsulating complex issues so that developers don't have to consider them. For example, reminders automatically take care of MAC-based integrity checks, and sections such as TreeSection hide the use of Ajax requests for dynamic updates. However, developers must now learn about the Fiz component set, which adds complexity. It is not clear how to measure complexity quantitatively, so we can only understand Fiz' net impact on complexity by observing usage of the framework in a variety of applications. We hope this will happen as a result of the open-source releases. In the meantime, we recognize that our mission is to develop components that provide maximum power with minimum complexity.

6 Comparisons

Fiz differs from other Web frameworks in several significant respects. Most of these differences came about because Fiz is focused on creating higher-level components that hide HTML, whereas most existing frameworks are focused on generating HTML.

Almost all existing Web frameworks besides Fiz use templates as the top-level structuring mechanism for Web pages. This works for simple document-oriented Web pages but does not work as well for Web applications: templates encourage the developer to think about low-level HTML rather than the high-level structure of the application. Furthermore, Web pages for applications tend to be highly conditionalized: the templates end up with too much code to visualize the HTML structure and too much HTML to visualize the code structure.

Most templating systems permit templates to reference other templates and encourage developers to use templates as the basis for reusable components. However, templates do not provide a good mechanism for creating reusable components. Top-level components specifying the overall page structure should contain little or no HTML, so there is no particular advantage in using templates to represent them. Lower-level component templates need to be highly parameterized if they are to be reusable; this increases the amount of code that they contain, which exacerbates the problems of the previous paragraph. Overall, templates work against reusable components.

The facilities of an object-oriented programming language provide a better basis for creating reusable components than templates. Hence Fiz bases its component mechanism on sections defined as Java classes. This encourages developers to think more about the top-level structure of their pages and less about low-level HTML. Fiz does use templates, but only for what they do best, which is generating HTML and other low-level formats inside components. This leads to a different style of usage for templates, where there are many small templates embedded in highly conditionalized code; the containing code implements much of the conditional behavior, so the templates can employ simpler substitution mechanisms.

Most existing Web frameworks use an overall application structure based on the model-view-controller paradigm (MVC). However, the division of functionality between view and controller is a bit unclear in MVC and different frameworks make the division differently. Fiz uses a two-part division into back end and front end (see Figure 1), where the back end is equivalent to the model from MVC and the front end includes both view and controller.

Combining view and controller functionality creates a cleaner separation between components. In the Rails framework, for example, a controller typically collects all of the data needed for a page into global variables, then invokes one or more views (templates) to render the page using the collected data. This means that the controller has global knowledge of all the data in the page. In Fiz, however, each section handles both the data collection and rendering functions for its part of the page. This provides better encapsulation, eliminating the need for global knowledge of the page.

Fiz also differs from other Web frameworks in its data management. Section 4.3 has already discussed how Fiz encourages a variety of data managers, rather than just relational databases, and uses an asynchronous invocation approach to improve performance. In addition, Fiz does not use ORM (object-relational mapping). Instead of creating a separate Java class for each database table and instantiating an object of that class for each row that is processed, Fiz uses a single structure, the dataset, to represent any row (or collection of rows) from any table. This makes sense because Web applications do very little processing of their data; their primary function is to move data from an input source into an output Web page and vice versa. Datasets work well for this because they integrate cleanly with data managers for input and with templates for generating Web pages. Most data never needs to be converted to an internal representation: it arrives as a string, is stored in the dataset as a string, and is incorporated in another string for output.

Fiz datasets are similar to several other mechanisms for representing structured data, such as XML, JSON, and YAML, and several of these formats can be used to store datasets externally. The primary benefit of datasets is the way they are integrated into the flow of Web page processing via data requests, templates, reminders, etc. Datasets provide a flexible integration mechanism between components in the server and between the server and the browser.

7 Conclusion

Fiz takes a different approach to Web application development than previous frameworks. Whereas previous frameworks take a bottom-up approach, layering on facilities to optimize HTML generation, Fiz takes a top-down approach focused on creating high-level reusable components. The Fiz framework makes it possible to create components that encapsulate complex presentations and behaviors, yet can be instantiated in a simple, almost declarative, fashion.

The next steps for Fiz will take it in two directions: first, wider usage in production applications to test its overall usability; and second, creation of additional components to see how large a library can be created. These experiences will provide a more complete evaluation of how well Fiz raises the level of programming for Web applications.

8 Acknowledgments

Michael Armbrust, David Erickson, Guru Parulkar, Mendel Rosenblum, Abraham Sebastian, William Sobel, Eric Stratmann, and Guido van Rossum have improved the presentation of this paper by applying a critical eye to various drafts.

9 References

[1] Bray, Tim, et al., editors, *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, W3C Recommendation (http://www.w3.org/TR/2006/REC-xml-20060816), 2006.

[2] Coward, Danny, et al., editors, *Java Servlet Specification, Version 2.4*, Sun Microsystems, 2003.

[3] Django project home page (http://www.djangoproject.com/).

[4] Flanagan, David, *Java In A Nutshell*, O'Reilly Media, 2005.

[5] Flanagan, David, *JavaScript: The Definitive Guide*, O'Reilly Media, 2006.

[6] Flanagan, David, and Matsumoto, Yukihiro, *The Ruby Programming Language*, O'Reilly Media, 2008.

[7] Garrett, Jesse James, *Ajax: a New Approach to Web Applications*, http://www.adaptivepath.com/ideas/essays/archives/000385.php.

[8] Goodman, Danny, *Dynamic HTML: The Definitive Referenc*, O’Reilly Media, 2006.

[9] Google Web Toolkit home page (http://code.google.com/webtoolkit/).

[10] Jquery project home page (http://jquery.com/).

[11] Lutz, Mark, *Programming Python*, O'Reilly Media, 2006.

[12] Musciano, Chuck, and Kennedy, Bill, *HTML and XHTML: The Definitive Guide*, O'Reilly Media, 2002.

[13] Ousterhout, John, "Scripting: Higher-Level Programming for the 21st Century," *IEEE Computer*, Vol. 31, No.3, March 1998, pp. 23-30.

[14] Prototype project home page (http://www.prototypejs.org/).

[15] Ramakrishnan, Raghu, and Gehrke, Johannes, *Database Management Systems*, McGraw-Hill, 2003.

[16] Reenskaug, Trygve. *Models-Views-Controllers*. Xerox PARC technical notes, December, 1979 (http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf).

[17] Reenskaug, Trygve. *Thing-Model-View-Editor*. Xerox PARC technical note, May 1979 (http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf).

[18] Ritchie. D.M., and Thompson, K. "The UNIX Time-Sharing System." *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.

[19] Script.aculo.us project home page (http://script.aculo.us/).

[20] Tatroe, K., Lerdorf, R., and MacIntyre, P. *Programming PHP, Second Edition*. O'Reilly Media, April 2006. Also see http://www.php.net/.

[21] Thomas,D., and Heinemeier Hanson, D. *Agile Web Development with Rails*. Second Edition, The Pragmatic Bookshelf, 2007.

[22] Yahoo! User Interface Library home page (http://developer.yahoo.com/yui/).