Pixaxe: A Declarative, Client-Focused Web Application Framework

Rob King
Principal Researcher, TippingPoint DVLabs

Abstract

This paper provides a brief introduction to and overview of the Pixaxe Web Application Framework (“Pixaxe”). Pixaxe is a framework with several novel features, including a transparent template system that runs entirely within the web browser, an emphasis on developing rich internet applications as simple web pages, and pushing as much logic and rendering overhead to the client as possible. This paper also introduces several underlying technologies of Pixaxe, each of which can be used separately: *Jenner*, a completely client-side template engine; *Esel*, a powerful expression and query language; and *Kouprey*, a parser combinator library for ECMAScript.

1 Introduction

There has been an explosion of frameworks for the building of Rich Internet Applications (RIAs). Frameworks exist using every popular (and unpopular) programming paradigm, language, and server side technology. Frameworks range in complexity from a simple JavaScript libraries that merely ease the handling of normal DOM events to frameworks that completely abstract away HTML and JavaScript. Some frameworks run entirely on the client, while others require considerable server side support.

This paper describes the Pixaxe Web Application Framework (“Pixaxe”). Pixaxe is interesting in that it focuses on creating web pages using a powerful, functional expression language that is a superset of normal XHTML. The compiler and virtual machine for this language is implemented entirely in ECMAScript and runs entirely within a web browser. By evaluating these expressions, web pages are rendered, inputs are validated, and client-server communication is initiated.

Pixaxe’s other interesting features include a complete parser combinator library running entirely within a web browser, an extremely server agnostic design (more so than most “server agnostic” frameworks), an extremely easy to use Model-View-Controller (MVC) design, and a very bandwidth-frugal design that transmits a page only once and then transmits only changes to interesting data.

Pixaxe was deisgned to be very useful in developing web interfaces for legacy applications, or in other situations where the web interface is not the primary interface to a set of data. It was also designed to be very efficient in the use of server resources, by limiting required bandwidth and performing as much computation and rendering on the client as possible. In fact, Pixaxe requires nothing more of a server than the ability to serve static files.

In feel, Pixaxe is closest to XForms [9] [1], in that it views web pages as declarative interfaces modifying local models that can then be synchronized with servers without reloading the page.

2 A Short Example

Unlike some other application frameworks, Pixaxe attempts to keep web application development as close to web page authoring as possible. It does not abstract away HTML, CSS, or any other web technology but instead encourages the developer to write directly in HTML using a declarative, template-driven approach. This allows developers to leverage existing technologies to the largest extent possible, and turns XHTML into a powerful interface description language (especially when using the XSLT macros provided by Pixaxe, discussed in section 3.5).

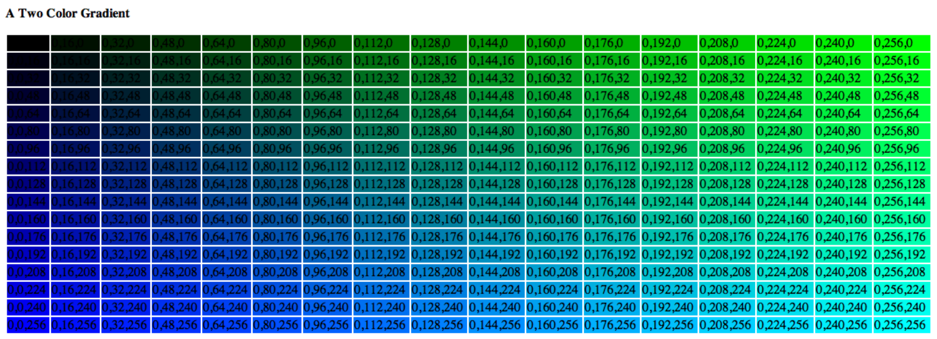
Figure 1 illustrates a simple example of a Pixaxe application. When viewed in a web browser, this example would be rendered as shown in Figure 2.

This example shows some interesting features of Pixaxe: there are no explicit calls in any scripting language, it looks like a normal XHTML document, and there are some special template directives freely mixed with the markup. What is unusual is that this page is rendered,

Figure 1 A simple Pixaxe application.

```
<html>
<head>
    <title>Simple Example</title>
    <script type="text/javascript"
            src="pixaxe.js" />
</head>

<body>
    <p><b>A Two Color Gradient</b></p>
    <table style="width: 100%;">
    ${for i from 0 to 256 by 16 return
      <tr>
        ${for j from 0 to 256 by 16 return
          <td style="background:
                     rgb(0, ${j}, ${i});">
             0,${j},${i}
          </td>
        }
      </tr>
    }
  </table>
</body>
</html>
```

Figure 2 Rendered output (in Apple Safari).



A Two Color Gradient

[illegible]	0,16,0	0,32,0	0,48,0	0,64,0	0,80,0	0,96,0	0,112,0	0,128,0	0,144,0	0,160,0	0,176,0	0,192,0	0,208,0	0,224,0	0,240,0	0,256,0
0,0,16	0,16,16	0,32,16	0,48,16	0,64,16	0,80,16	0,96,16	0,112,16	0,128,16	0,144,16	0,160,16	0,176,16	0,192,16	0,208,16	0,224,16	0,240,16	0,256,16
0,0,32	0,16,32	0,32,32	0,48,32	0,64,32	0,80,32	0,96,32	0,112,32	0,128,32	0,144,32	0,160,32	0,176,32	0,192,32	0,208,32	0,224,32	0,240,32	0,256,32
0,0,48	0,16,48	0,32,48	0,48,48	0,64,48	0,80,48	0,96,48	0,112,48	0,128,48	0,144,48	0,160,48	0,176,48	0,192,48	0,208,48	0,224,48	0,240,48	0,256,48
0,0,64	0,16,64	0,32,64	0,48,64	0,64,64	0,80,64	0,96,64	0,112,64	0,128,64	0,144,64	0,160,64	0,176,64	0,192,64	0,208,64	0,224,64	0,240,64	0,256,64
0,0,80	0,16,80	0,32,80	0,48,80	0,64,80	0,80,80	0,96,80	0,112,80	0,128,80	0,144,80	0,160,80	0,176,80	0,192,80	0,208,80	0,224,80	0,240,80	0,256,80
0,0,96	0,16,96	0,32,96	0,48,96	0,64,96	0,80,96	0,96,96	0,112,96	0,128,96	0,144,96	0,160,96	0,176,96	0,192,96	0,208,96	0,224,96	0,240,96	0,256,96
0,0,112	0,16,112	0,32,112	0,48,112	0,64,112	0,80,112	0,96,112	0,112,112	0,128,112	0,144,112	0,160,112	0,176,112	0,192,112	0,208,112	0,224,112	0,240,112	0,256,112
0,0,128	0,16,128	0,32,128	0,48,128	0,64,128	0,80,128	0,96,128	0,112,128	0,128,128	0,144,128	0,160,128	0,176,128	0,192,128	0,208,128	0,224,128	0,240,128	0,256,128
0,0,144	0,16,144	0,32,144	0,48,144	0,64,144	0,80,144	0,96,144	0,112,144	0,128,144	0,144,144	0,160,144	0,176,144	0,192,144	0,208,144	0,224,144	0,240,144	0,256,144
0,0,160	0,16,160	0,32,160	0,48,160	0,64,160	0,80,160	0,96,160	0,112,160	0,128,160	0,144,160	0,160,160	0,176,160	0,192,160	0,208,160	0,224,160	0,240,160	0,256,160
0,0,176	0,16,176	0,32,176	0,48,176	0,64,176	0,80,176	0,96,176	0,112,176	0,128,176	0,144,176	0,160,176	0,176,176	0,192,176	0,208,176	0,224,176	0,240,176	0,256,176
0,0,192	0,16,192	0,32,192	0,48,192	0,64,192	0,80,192	0,96,192	0,112,192	0,128,192	0,144,192	0,160,192	0,176,192	0,192,192	0,208,192	0,224,192	0,240,192	0,256,192
0,0,208	0,16,208	0,32,208	0,48,208	0,64,208	0,80,208	0,96,208	0,112,208	0,128,208	0,144,208	0,160,208	0,176,208	0,192,208	0,208,208	0,224,208	0,240,208	0,256,208
0,0,224	0,16,224	0,32,224	0,48,224	0,64,224	0,80,224	0,96,224	0,112,224	0,128,224	0,144,224	0,160,224	0,176,224	0,192,224	0,208,224	0,224,224	0,240,224	0,256,224
0,0,240	0,16,240	0,32,240	0,48,240	0,64,240	0,80,240	0,96,240	0,112,240	0,128,240	0,144,240	0,160,240	0,176,240	0,192,240	0,208,240	0,224,240	0,240,240	0,256,240
0,0,256	0,16,256	0,32,256	0,48,256	0,64,256	0,80,256	0,96,256	0,112,256	0,128,256	0,144,256	0,160,256	0,176,256	0,192,256	0,208,256	0,224,256	0,240,256	0,256,256

along with the special template directives, entirely by the client. This page could be stored in a file on a local filesystem and opened in a web browser, and it would be rendered correctly.

This illustrates one of the core guiding principles of Pixaxe development: all display and rendering should be done by the client. This stems from the assumption that the web interface is but one of many interfaces to the same datasource. This example also provides a sample of Pixaxe's template syntax which is based on a purely functional and side-effect-free expression language known as *Esel* [2].

This example shows how Pixaxe integrates with other technologies and encourages developers to write web pages correctly and portably while still reaping the benefits of Pixaxe.

3 Developing Rich Internet Applications With Pixaxe

Pixaxe attempts to keep development of rich internet applications similar to the development classic web pages. It integrates a classic Model-View-Controller (MVC) development paradigm (as described in [6]) and declarative paradigm, but does not enforce this. Pixaxe itself views a web page as two combined entities: a *template* and a *store*. The template corresponds roughly to the view portion of the MVC model, and the store corresponds roughly to the model portion. The controller portion of the MVC model (handling user input) is handled by a combination of Esel expressions embedded in templates (which validate the input and update the model) and the store (which synchronizes the model with all interested parties and instructs the page to re-render if necessary).

Brief overviews of the template language, store, handling user input, and client-server communications are presented below.

3.1 The Jenner Template Language

The template language of Pixaxe is known as *Jenner*. It is capable of being used independently of Pixaxe as a simple client-side template engine. (Jenner itself is a superset of the *Esel* expression and query language, which itself may be used independently of Jenner).

The Jenner template language is a full-fledged expression and data query language. Jenner expressions are used to retrieve and optionally transform data from a store and insert the results into the rendered page. Jenner expressions can also be used to validate user input and update the store when data changes.

Jenner's syntax and semantics are similar to those of Java's Unified Expression Language (see [1]); these similarities are intentional. While many small expressions are valid in both languages, the two are different enough to be considered generally incompatible.

Jenner has two different types of expressions: *value* expressions and *reference* expressions. The name "value expression" is perhaps misleading, since all Jenner expressions return a value, but it is useful to differentiate between the two types of expressions.

3.1.1 Value Expressions

An Jenner value expression can take two forms: *literal* and *bracketed*. A literal expression is used to declare a single value, though this literal expression may contain other subexpressions (which would be bracketed).

For example, these are all valid literal expressions:

```
42
Hello, World!
true
null
```

Each of the above expressions are, as are all expressions in Jenner, typed. Jenner supports all primitive types defined by ECMAScript, except for the *undefined* value which is treated as equivalent to *null*.

Jenner also supports a special *collection* type, which is roughly equivalent to an ECMAScript array. Jenner provides a powerful list comprehension syntax, discussed below, for expressing collections.

A bracketed expression is one that begins with the special sequence `${` and ends with the character }. Inside these delimiters, other Jenner expressions may be embedded, including other bracketed expressions. Within these brackets, the full expression language is available.

Jenner supports most common operators, including basic and modular arithmetic, boolean combination, string concatenation, and a C-style trinary conditional operator. Jenner also includes operators for testing set membership and fast tests for empty strings and collections.

Jenner also supports variable lookups (but not assignment). These can be written in a fashion similar to that of ECMAScript. Variables are exported to the Jenner runtime by the hosting environment; it is not possible to access objects that are not directly referencable from these exported variables.

As an example, this expression returns the sum of the value of the `aNumber` variable and 1, if `aNumber` is defined. If `aNumber` is not defined, *null* is returned:

```
${empty aNumber ? null : aNumber + 1}
```

Bracketed expressions may also contain function calls. Functions cannot be defined in Jenner itself; they must be exported by the hosting application. Functions can be namespaced to avoid name collision. Jenner contains by default a reasonable standard library of functions, but developers are encouraged to write and share new collections of functions.

Perhaps the most interesting form a value expression is Jenner's list comprehension expression, known as a *FLWR* expression. [3] FLWR expressions are used to create Jenner collections and are also used to query collections of data by comprehending a collection of results matching some predicate.

FLWR expressions always evaluate to a collection, even if that collection is empty. FLWR expressions can be viewed as iterative constructs, where a variable is assigned a numeric value from some lower bound to some upper bound. For each iteration of this loop, the value is incremented by an optional step expression or by one. FLWR expressions provide lexical scoping – local variables can be declared that are visible only within the body of a FLWR expression. For each value generated by some generator expression, if it an optional conditional clause evaluates to true, the value is added to the resulting collection in order.

For example, this expression would create a new collection consisting of all members of an array of strings which are longer than three characters, with each result capitalized:

```
for i from 0 to names.length - 1
    var s := names[i]
    where s.length > 3
        return upper(s)
```

FLWR expressions are similar to the *FLWOR expressions* of the XQuery language (see [8]); this similarity is intentional.

3.1.2 Reference Expressions

Jenner also supports expressions that return a reference to an object in the hosting application. References are represented as a tuple consisting of an object reference and a (possibly empty) property name of that object.

Reference expressions are delimited by #{ and }. Within these delimiters, a subset of the full Jenner value syntax is allowed; the result of the embedded expression must be either *null* or an object with an optional property name. Only objects exported from the hosting application are vali.

This example expression would return a reference to the object denoted by `people.addresses` with a property name of `1`:

```
#{people.addresses[1]}
```

Reference expressions cannot be mixed with value expressions and may not be embedded in other expressions; they must be entirely standalone.

Reference expressions do not directly support assignment; this is considered a feature. Instead, they may be used by the hosting application as a target for assignment, but the hosting application is free to use or not use reference expressions in any way.

3.1.3 Node Expressions

Jenner also provides a *node* type. This provides a literal syntax for elements which is identical to the XML syntax for specifying elements. A node literal may be used in an expression anywhere a literal is allowed.

Since nodes may contain other nodes and Jenner expressions, a web page can therefore be considered a single large Jenner expression.

Node literals' names must be specified directly; they cannot be expressions. This applies as well to the names of attributes in node literals. The contents of attributes and nodes, however, can be any valid non-node Jenner expression.

Jenner can leverage any expression. This has some interesting consequences. For example, a conditional expression can be used to only render a node depending on the state of the application. For example, this template could be used to display a list of messages should any be present:

```
<body>
    ${not empty msgs ?
        <ul>
        ${for i from 0 to msgs.length - 1
            return
            <li style="text:
                        ${msgs[i].unread ?
                        'red' : 'black'}">
                ${msgs[i]}
            </li>
        }
        </ul>

    : <p>No messages</p>}
</body>
```

Note the embedded Jenner expression inside the `style` attribute of the `li` element.

Jenner's template language is not particularly new or unique; it bears large similarities to templates used by Sun's Java Server Pages (see [2]) or other server-side template systems. What makes Jenner interesting is that all template evaluation and rendering is performed entirely by the client. Jenner is believed to be the first completely client-side template system in which markup and template instructions can be freely mixed (other than XSLT).

Jenner has some advantages over other client-side template systems. For example, while most modern web browsers support XSLT templates, there are no standard ways to apply XSLT transformations multiple times; generally the transformations are applied only once, at page load. Jenner may re-render the page at any time. Some other templating systems, such as JavaScriptMVC [4] perform template evaluation and rendering entirely on the client, but do not allow markup and template instructions to be freely mixed.

3.2 Storing Data

As stated above, Pixaxe loosely follows the MVC paradigm for development. A single page has a view component in the form of a Jenner template and a single model in the form of a *store*. The store contains all data that is relevant to the application at any given time.

Pixaxe keeps the store synchronized between all interested parties. When the data in the store changes, Pixaxe calls Jenner to re-render the page. If the user performs an action that results in client-server communication, Pixaxe serializes the store and passes it to the server, and then updates the store with the results of server processing. If the user performs input that updates the store, the page is re-rendered to reflect the new values.

When the store is serialized and sent to the server, the server may make changes and send an updated version of the store back to the client. The client can then re-render the page to reflect the new values. Note that this asynchronous transmission of the serialized store to and from the server is the only client-server communication in Pixaxe; the page itself is downloaded only once. This can result in significant bandwidth savings, and also allows for a very abstract server interface - client-server communication is reduced to synchronizing a simple JSON document.

Pixaxe integrates with Jenner by the simple expedient of setting the global store to be Jenner's default environment. Therefore, all but the simplest Pixaxe applications will contain something like this line somewhere in a script element:

```
com.deadpixi.jenner.defaultEnvironment =
    new com.deadpixi.pixaxe.Model( ...
```

Pixaxe tries to keep the creation of the store as declarative as possible. The `com.deadpixi.pixaxe.Model` constructor takes two arguments: the first is a single object that becomes the store. Developers are encouraged to write this object using ECMAScript object literal syntax, to keep to the declarative programming style as much as possible. An optional second argument to the constructor is a URL whose contents (which must be JSON, see [3]) will be loaded into the store after the page has finished loading.

As an example, this might be the object used as the store of a simple address book, with some potentially useful initial values:

```
{
    "addresses": [
        {"name": "Rob",
         "name": "jking@deadpixi.com"},
        {"name": "Betsy",
         "name": "betsy@example.com"}
    ]
}
```

This object would then be available to Jenner as its default environment, and therefore expressions in the template would have access to a variable called `addresses`.

Alternatively, this object (which is expressed in JSON) could be placed in a separate file, and a URL specifying that file could be passed as the second argument to the `Model` constructor. This would cause this file to be loaded and merged with the store after the page has finished its initial load.

Note that the page is initialized only once, at page load time. The page itself is never again transmitted across the network, and there is no HTML form style "submit-reload" cycle. The store is synchronized between client and server using the *de facto* standard `XMLHttpRequest` method, and the page is re-rendered by re-evaluating the Jenner expression which makes up the page.

3.3 Handling User Input

In keeping with Pixaxe's goal of leveraging existing technology, Pixaxe using XHTML as a rich interface specification language. Standard XHTML form controls are used to create input elements (possibly augmented by XSLT macros, see section 3.5).

Input controls can be placed anywhere in a document. If a control is placed inside of a `form` element, then manipulating the control will result in some type of synchronization of the store with the server (note that this does not mean a traditional XHTML form submission). Controls outside of a `form` element result only in local changes to the store (which may of course be synchronized with the server later).

Pixaxe supports all HTML control elements, including `input`, `button`, `select`, and `textarea`. Each of these controls can be linked to the page's store by placing a reference expression in the element's `name` attribute. Pixaxe will then ensure that the value in the store and the controls's value are synchronized.

For example, this declaration would create a text input element whose value would be placed in the `name` property of the page's store:

```
<input name="#{name}"
       value="${name}" />
```

In this example, whenever the user activates a submission control, the page's template is re-evaluated, rendering the input control with the current value of the *name* variable in the store. Any change in the control's value by the user would be automatically placed into the store. This synchronization between store, template, and user allows for a very powerful and declarative method of interface specification.

This gives rise to a very simple, two stage process for the handling of input when the user-activated submission control is not part of a form. First, the page's store is updated such that all controls that reference the store have their values placed in the store. The page is then re-rendered.

For example, this page will automatically re-render to display the current value of the name control whenever the user activates the submit control:

```
<p>Hello, ${empty name ? 'Stranger' :
                         name}</p>
<input name="#{name}" />
<input type="submit" value="Ok" />
```

No client-server communication takes place when the user activates the submit control in this example: the store's *name* variable is updated and the page is re-rendered by re-evaluating its template locally.

To integrate form controls with Pixaxe, the semantics of the standard attributes assigned to form controls is overloaded. The meanings assigned to each attribute are described below:

accept The optional `accept` attribute can be used to modify or validate the value of the control before it is copied to the store. The Jenner expression specified in the `accept` attribute is evaluated each time the control's value is evaluated and the value of the accept expression is instead copied to the model.

name If the value of the `name` attribute is an Jenner reference expression, then the control is linked to the model. Whenever the user activates a submit control, the value of the current control is copied to the property pointed to by the reference.

value If the value of this attribute is an Jenner expression, it is evaluated each time the page is rendered and the value of the control is set to the result.

These attributes are all specified as part of the HTML standard. All other attributes may contain Esel expressions; these will be evaluated and set each time the page is rendered (see, for example, the `style` attribute in Figure 1).

Synchronization with the store is bidirectional. If the store is updated through some other means (generally through client-server communication), the control is updated to keep its value synchronized. Thus, controls always accurately reflect the state of the store and vice-versa.

All types of controls can be used, but controls of type *hidden* are treated specially. A hidden control is used to set an initial default value for some part of the store. Thus, the developer can initialize certain values in the store when the page initially loads.

3.4 Client-Server Communications

Client-server communication is done entirely via JSON documents POSTed to URIs asynchronously. Communication is not viewed as a imperative action, but rather as a synchronization of the state of the page's store with the server (by convention this is known as "synchronizing the world").

Client-server communication is initiated when the user activates a submit control that is a child of a `form` element. There are two possible methods of communication in that case: classic form submission, and store synchronization.

If a form element's `enctype` attribute is not set to "text/javascript", then the form is submitted as per the HTML standard and whatever is returned from the form replaces the contents of the page. This is used to interface with legacy code that insists on using normal HTML form submission semantics.

If a form element's `enctype` attribute is set to "text/javascript", however, the form is not submitted using the classic method. Instead, Pixaxe first synchronizes the values of all controls with the page's store. The page's store is then serialized into JSON and POSTed asynchroniously to the URL specified by the form's target. It is expected that the server return a JSON object which is then merged with the page's store. This merge process simply copies all properties from the returned object to the store with the same name, potentially overwriting the values of old properties or adding new ones. Properties that are not named in the returned object are not affected. Note that in this case, the page is not reloaded in any way; it is simply re-rendered locally by Jenner.

For example, assume that the page's store is displayed here, as JSON:

```
{
    "name": "Arthur",
    "id": 42
}
```

If the user activated a submit control contained by a `form` element whose `enctype` attribute is set to "text/javascript", the page's store would be serialized and submitted as described above.

Below is an example response from the server, which would be returned as the body of the response to the POST request.

```
{
    "id": -1,
    "invalid": true
}
```

Pixaxe would merge this response with the page's store, resulting in the new contents of the store.

```
{
    "name": "Arthur",
    "id": -1,
    "invalid": true
}
```

After this synchronization process, the template is re-rendered locally by Jenner.

Forms whose `enctype` is "text/javascript" may also place an Esel expression in their `accept` attribute. This expression must evaluate to *true*, or the server synchronization will not happen. This expression can be used to validate user input before initiating client-server communications. The form will still be re-rendered even if this validation fails, giving the application a chance to inform the user of invalid input.

3.5 XSLT Macros

Pixaxe comes with a collection of XSLT stylesheets. These stylesheets define macros, which consist of special nodes in the original markup that are transformed by XSLT at page load time to a collection of normal HTML markup and Jenner template instructions.

Several useful XSLT macros exist, including macros that create paged tables, modal dialog boxes, lightboxes, tab boxes, and AJAX-style file upload controls. These stylesheets can be applied automatically by most modern web browsers at page load time.

One interesting aspect of these macros is that they are implemented entirely in regular HTML and Jenner instructions. They therefore require no additional server side support. Since the macro expansion is applied only at page loading time, it does not significantly affect page re-render times.

These macro packages can be arbitrarily complex. For example, Figure 3 illustrates the code for a simple tab box on a web page, and Figure 4 illustrates the rendered page. Figure 5 illustrates (partially) the expansion of the `dppx:tab-box` and `dppx:tab` macros.

Macros can be arbitrarily complex, giving developers the ability to abstract away as much XHTML as desired. Additionally, by allowing the free mixing of Jenner markup and HTML, XSLT stylesheets applied to pages have a much richer "target language" than traditional stylesheets.

3.6 Putting It All Together

This example demonstrates a complete, if simple, Pixaxe application that uses a large number of the described features. This example application builds a very simple, shared bulletin board. Users can post short messages which are then visible to all other users.

First, the page store is declared. This should be placed in a separate file from the web page, in this example called "store.js".

```
com.deadpixi.jenner.defaultEnvironment =
    new com.deadpixi.pixaxe.Model({
        msgs: [],
        newMsg: ""
    }, "messages.json");
```

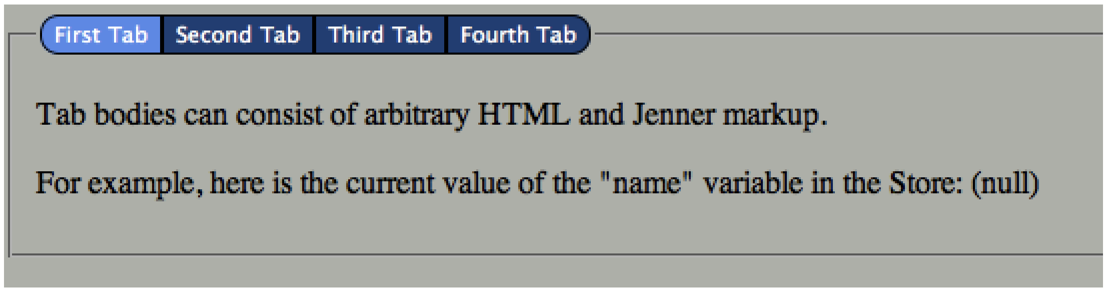
The second argument is a URL that points to a file that is assumed to contain the list of messages currently known to the server as a single JSON object, with a property called "msgs". This URL will be used to load the initial values into the page's store after the page has finished loading.

The rest of the application is defined in a normal XHTML file. For brevity, only the `body` element of this file is shown below. Esel standard function `now` is used to indicate to the user how up to date the page's display is. Below this is a list of messages retrieved from the server.

Figure 3 An example of a page using XSLT macros.

```
<body>

<dppx:tab-box>
    <dppx:tab label="First Tab" selected="true">
        <p>Tab bodies can consist of arbitrary HTML and Jenner markup.</p>
        <p>For example, here is the current value of the "name"
           variable in the Store: ${name}</p>
    </dppx:tab>

    <dppx:tab label="Second Tab">
        <p>Another tab.</p>
    </dppx:tab>

    <dppx:tab label="Third Tab">
        <p>Yet another tab.</p>
    </dppx:tab>

    <dppx:tab label="Fourth Tab">
        <p>Tabs everywhere!</p>
    </dppx:tab>
</dppx:tab-box>

</body>
```

Figure 4 The rendered example of the source in Figure 3.



Tab bodies can consist of arbitrary HTML and Jenner markup.

For example, here is the current value of the "name" variable in the Store: (null)

Figure 5 A partial expansion of the `dppx:tab-box` and `dppx:tab` macros.

```
<fieldset><input type="hidden" value="id4127134"
                 name="#{controller.dppx_tabselid4127132}"/>
        <legend><input type="submit"
                       name="#{controller.dppx_tabselid4127132}"
                       class="dppx-tab dppx-tab-left
                       dppx-tab-${controller.dppx_tabselid4127132
                       != 'id4127134' ?
                       'un' : ''}selected"
                       accept="id4127134" value="First Tab" />
                <input type="submit"
                       name="#{controller.dppx_tabselid4127132}"
                       class="dppx-tab dppx-tab-${
                       controller.dppx_tabselid4127132
                       != 'id4127149' ?
                       'un' : ''}selected"
                       accept="id4127149" value="Second Tab" />
...
    <div class="dppx-tab-body
                dppx-tab-body-${controller.dppx_tabselid4127132
                != 'id4127134' ? 'un' : ''}selected">
        <p>Tab bodies can consist of arbitrary HTML and Jenner markup.</p>
        <p>For example, here is the current value of the "name"
           variable in the Store: ${name}</p>
    </div>
```

```
<body>
    <p>${msgs.length}
        message${msgs.length != 1 ?
                    's' : ''}</p>
    <p>Updated at ${now()}.</p>

    <ul>
     ${for i from 0 to msgs.length - 1
        var m := msgs[i] return
        <li>${m}</li>
     }
    </ul>
```

After this, the user is given a text input area to post a new message. This input control is linked to the page's store by placing a reference expression in the control's `name` attribute.

```
    <hr />
    <input name="#{newMessage}" />
```

A submit control is placed inside a `form` element whose action points to the message posting script on the server.

```
    <form enctype="text/javascript"
          action="/cgi-bin/post.cgi">
        <input type="submit"
               value="Add Message" />
    </form>
```

This would create a complete and fully functional application, except for the server components. The list of messages could be a simple JSON document. The message posting script would need only to deserialize the contents of the POSTed store from JSON, extract the ”newMsg” property from this deserialized object, append its value to the messages document, and return that document to the client.

4 Comparison to Other Frameworks

Pixaxe was designed to be very light in terms of server resources. This makes it well suited for situations where it is not the primary interface to a service, or where the service is a legacy application.

This section compares Pixaxe to some other frameworks, and helps illustrate the situations for which Pixaxe is best suited.

4.1 Pixaxe and the Google Web Toolkit

The Google Web Toolkit (`http://code.google.com/webtoolkit/`) is a mixed client-server toolkit from Google. The toolkit (commonly referred to as "GWT") relies heavily on the Java language for development. GWT in fact compiles Java sources to JavaScript code which is then executed on the browser.

GWT is in many ways the opposite of Pixaxe. Developing with GWT follows much the same process as developing any large Java application - there is a compile/debug/edit cycle, and the use of Java-centric tools is encouraged. GWT encourages a mixed object-oriented and procedural development paradigm by building interfaces programatically in Java (though there has been more support lately for declarative interface specification).

While GWT can be used in a server agnostic manner, much of the code is written assuming a Java Servlet Container [5] on the server. GWT also abstracts away large amounts of HTML.

GWT has many advantages: an extremely large user base, and the support of one of the largest technology companies in the world, as well as numerous mature tools (such as Eclipse [6]) that can make the development of large applications considerably easier.

By the same token, GWT is often overkill for small projects or in situations where a Java Servlet Container is not available on the server. For smaller, rapidly-developed applications, Pixaxe's lack of a compile/debug/edit cycle can be a major advantage. In situations where server resources are limited or Java technologies cannot be used, Pixaxe's server agnosticism makes it an attractive choice. Pixaxe would likely not scale well to applications the size of Google Mail [7], but has easily handled smaller applications.

4.2 Pixaxe and SproutCore

SproutCore (`http://www.sproutcore.com`) is a popular client-centric toolkit used by many popular websites. SproutCore is close in feel to Pixaxe. SproutCore is client-focused, running its code solely within the browser. It also favors a declarative interface specification, and development follows the Model-View-Controller pattern. Its use in large, successful applications such as Apple's MobileMe [8] portal illustrates that it can be successfully used for large projects and is a mature choice.

However, SproutCore differs from Pixaxe in a few interesting ways. SproutCore still follows a shortened compile/debug/edit cycle. SproutCore requires Ruby for application development, though not to run or view applications. It is server agnostic for the most part, though much of its documentation makes assumptions that Ruby is used on the server side.

SproutCore development tends to use much more JavaScript than Pixaxe. SproutCore could be called a "JavaScript framework", in that much of the business logic of code is specified in JavaScript. Indeed, interfaces are often built (semi-declaratively) using JavaScript.

Pixaxe enables an arguably simpler development path, allowing purely declarative interfaces in XHTML and a simple expression language. New "widgets" in Pixaxe are easily created through simple XHTML, and optionally made reusable through simple XSLT macros.

Pixaxe and SproutCore are well suited to many of the same tasks. SproutCore's programming interface is more powerful than that of Pixaxe, but is concomitantly more complicated. For simple, radpily developed applications, especially single-page applications, Pixaxe may be the better choice.

4.3 Pixaxe and XForms

XForms (`http://www.w3.org/MarkUp/Forms/`) was the direct inspiration for Pixaxe. XForms is an application of XML for the specification of data processing models for XML, and user interfaces to those models. It does not require, but is often used "on top of" XHTML, using the latter as part of its presentation layer.

The original applications for which Pixaxe was used were originally to be written in XForms. Unfortunately, XForms is not natively supported in any mainstream web browser. Therefore, Pixaxe was developed to provide a similar development experience, but usable on any modern web browser.

XForms has the benefit of being extremely well specified, and builds on the extensive base of XML technologies specified by the W3 Consortium. XForms allows for the free mixing of presentational markup and logic specification in a single page, and lends itself well to forms-based development cycles.

Due to this strong inspiration, Pixaxe could easily be used in any situation where XForms could be used, but with the benefit of wider support. XForms was designed to help create applications that were still web pages for the most part, and this is a design goal shared by Pixaxe.

5 The Implementation of Pixaxe

This section of the paper discusses the implementation details of Pixaxe, including all of the technologies upon which it is built. The bulk of Pixaxe's code lies in the compiler and virtual machine for the Esel and Jenner languages. Also detailed here is the Kouprey parser combinator library, which is used to create the parser for Esel.

5.1 The Kouprey Parser Combinator Library

Pixaxe uses a parser combinator library known as *Kouprey* [9]. Kouprey eases the development of developing parsers by allowing developers the ability to express grammars using simple ECMAScript statements in something resembling Extended Backaus-Naur Form (EBNF, see [5]). The generated parsers are based on the Parsing Expresson Grammar (PEG) formalism (see [4]).

Kouprey is available to be used separately from Pixaxe. It has no dependencies other than a standard ECMAScript runtime. It is sufficiently powerful that a complete parser for the Component Pascal [10] programming language has been written entirely using Kouprey.

A full discussion of Kouprey is beyond the scope of this document, but interested readers are encouraged to consult the Kouprey home page at `http://www.deadpixi.com/kouprey`.

5.2 Esel, Jenner, and Their Virtual Machine

The Jenner template language is built on top of a small expression language known as Esel. Jenner's syntax is a pure superset of that of Esel, adding only the literal node type syntax.

Esel uses Kouprey to generate its parser. Esel expressions are compiled into abstract syntax trees, which are then passed to a code generator. This code generator creates programs for a virtual machine designed to run Esel expressions.

The Esel compiler, code generator, and virtual machine are written entirely in ECMAScript and are available for use independently from Pixaxe. Esel's virtual machine is a simple stack-based virtual machine with 32 instructions. The virtual machine itself is Turing complete, and provides support for such advanced features as lexically closed environments and a foreign function interface with ECMAScript.

5.3 The Jenner Template Engine

Jenner, Pixaxe's template engine, is remarkably simple in its implementation. Upon page load, Jenner is passed a DOM element object to treat as the root of the template; by default this is the page's `body` element.

The DOM of the page is then traversed. All text nodes and all comment nodes of a special syntax (by default, any comment node whose first two characters are "##") are appended to a single Jenner expression. Any time an element node is encountered, it is assigned a unique ID and a call to the special `jenner:nodeset` function is appended to the expression. This is done recursively until the entire page has been converted to a single large Jenner expression. To render the page, the root of this expression is simply evaluated.

The `jenner:nodeset` function is used to insert nodes dynamically into the page using standard DOM manipulation. The original page is copied to serve as a template for each render. Jenner also performs extensive caching of rendered and compiled results for speed.

Jenner also allows developers to override default handling of elements and attributes by name. Pixaxe uses this functionality to assign special meaning to various attributes, mostly for form and input handling.

Jenner is available for use independently of Pixaxe. Jenner could be very useful as a display technology for other frameworks.

5.4 Pixaxe, Input Processing and Data Management

Pixaxe itself builds data management capabilities on top of Kouprey, Esel, and Jenner. It is essentially a very thin layer on top of these technologies.

The vast majority of Pixaxe's code is used in processing user input, primarily performing form control value processing and input validation. Page store management is relatively simple, consisting mostly of copying values from controls into the model and instructing Jenner to re-render the page.

5.5 Client-Server Communications

Client-server communications in Pixaxe are simple, using the de facto standard `XMLHttpRequest` (see [7]) support in modern web browsers to POST serialized versions of the page's store and merge the returned information into the store.

Pixaxe uses native JSON processing functions if possible for speed, but will fall back to using an JSON library written in ECMAScript if native functions are not available.

One interesting feature provided by Pixaxe's client-server communications subsystem is the application of callbacks. All communication takes place asynchronously. The page store can include specially named functions that will be invoked when the store is serialized, when it is merged with the server, and on various error conditions. Pixaxe provides a standard set of callback functions that will render an "input shield" over the page, preventing any user interaction until the communications cycle is complete. Their use is, of course, optional.

5.6 Cross-Browser Support

Kouprey and Esel are written in pure ECMAScript and should work without modification in all conformant environments. Jenner and Pixaxe, however, are intimately involved in the way the browser represents pages and events and therefore must be written with cross-browser support in mind.

Jenner and Pixaxe officially support Apple Safari (versions 4 and later), Microsoft Internet Explorer (versions 7 and later), and Mozilla Firefox (version 3 and later). Other versions of these browsers and other vendors' browsers may work but they are not extensively tested.

Different browsers require different syntax in certain situations for Jenner templates. Most notably, some browsers require expressions to be inside comment nodes when they are inside `table` elements, while other browsers require them to be bare. Similar situations arise when dealing with `ol` and `ul` elements. Jenner and Pixaxe provide XSLT stylesheets that can be applied to the page as it is loaded that automatically translate pages to use the appropriate format, transparently to the developer.

6 The Future

Kouprey, Esel, Jenner, and Pixaxe are all under active development and several interesting features are planned for a future release.

Kouprey's next version is expected to be considerably faster and have better error handling and reporting. It will also be rewritten to make grammar definitions more natural when using the ECMAScript Compact Profile.

Esel's virtual machine is being rewritten to be faster and smaller. There are also some proposed language extensions, including destructuring assignment and n-way case statements.

Jenner and Pixaxe will be much more tightly integrated in a future release (though it is planned that Jenner will still be usable without Pixaxe). A faster rendering algorithm is also being worked on that involves static analysis of Esel expressions to determine which portions of the page would be affected by certain changes to the page's store. Additionally, a real-time synchronization mechanism is under development that would not require users to manually indicate that a form is ready for processing.

7 Availability

All of the technologies discussed in this paper are available under a a free software license. All of these technologies are currently available and in active use.

Download links and detailed documentation for all technologies are available at `http://wwww.deadpixi.com`.

References

[1] Chung, K.-M., Delisle, P., and Roth, M. Expression language specification. Part of the Java Community Process, see `http://www.jcp.org/en/jsr/detail?id=245`.

[2] Chung, K.-M., Delisle, P., and Roth, M. Java serverpages 2.1. Part of the Java Community Process, see `http://www.jcp.org/en/jsr/detail?id=245`.

[3] ECMA International. Ecmascript language specification. copy available at `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.p%df`.

[4] Ford, B. Parsing expression grammars: A recognition-based syntactic foundation. copy available at `http://www.brynosaurus.com/pub/lang/peg-slides.pdf`.

[5] Pattis, R. E. Ebnf: A notation to describe syntax. copy available at `http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf`.

[6] Reenskaug, T. Thing-model-view-editor. archived copy available at `http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf`.

[7] W3 Consortium. Xmlhttprequest. a working draft, copy available at `http://www.w3.org/TR/XMLHttpRequest/`.

[8] W3 Consortium. Xquery 1.0: An xml query language. copy available at `http://www.w3.org/TR/xquery/`.

[9] W3 Contortium. Xforms 1.1. copy available at `http://www.w3.org/TR/xforms/`.

Notes

[1]In fact, Pixaxe's original name was "JSONForms".

[2]The name "Esel" was inspired by "ECMAScript Expression Language".

[3]"FLWR" from "for", "let", "where", and "return", the four basic operations of the expression.

[4]See `http://www.javascriptmvc.org`.

[5]`http://java.sun.com/products/servlet/`

[6]http://www.eclipse.org

[7]http://mail.google.com

[8]`http://www.me.com`

[9]Kouprey was named after the Cambodian ox, by analogy to other parser generators such as `yacc` and `bison`.

[10]See `http://www.oberon.ch/`.