treehouse.

A Web Development Magazine

April 2006

ISSN :1558-013X

Inside the Treehouse

Treehouse Cover
by Brian Campbell

Treehouse Logo
by Julius Santiago

From the Editors — 3
by Kevin Hale

To the Editors — 4
Letters & Comments

Cartoon — 33
Robot Tea Party

Network News — 34
Around 9rules

Marketplace — 47
Classifieds

Code

How to Build a Blog Out of JavaScript — 7
by John Resig

Interview
Thomas Fuchs — 14

Book Review — 18
Code Complete

Best of the Web — 19
Code Links

Design

Styling for Submission — 21
by Trevor Morris

Interview
Bryan Veloso — 25

Book Review — 30
CSS Mastery

Best of the Web — 31
Design Links

Business

Business Taxes 101 — 36
by Jason Guthrie

Interview
Inkling — 40

Book Review — 45
Getting Real

Best of the Web — 46
Business Links

The Great Half Trip

By Kevin Hale

When we first started *Treehouse*, we promised ourselves that no matter what happened we'd commit ourselves to 6 months of issues and 1 free month to start everything off. This issue represents the closing bracket of that promise to ourselves and we couldn't imagine a better half-trip around the sun.

In September of last year, Ryan and I quit our jobs while Chris took one for the team to stay behind in a cube. The plan was that he would split his paycheck with the two of us and that was our way of making the venture seem less frightening than it should have been. We planned it so that we would have only one month to manifest *Treehouse* from vapors. There were very long nights of deciding and planning for what would be our routine for the next several months. In that first month, Ryan had to build a store and payment system by himself during the day and I had to design the modular template system by myself during the day. When Chris returned home from the cube at night, we rallied writers and contributors together.

At the end of that first month, we had a magazine, a store and a paycheck split in three. That was a hard month. That was an awesome month. As time went on, success collected in heaps. Because of our promise, *Treehouse* was always with us at each moment. In Boston, it was materialized hours after finding out we would be recipients of seed funding for another project. In California, when we flew across the country to pursue this project. In the days before one of the biggest presentations of our lives. *Treehouse* was there, waiting--marking every milestone.

And so it is at this latest milestone, that we've decided that it's time for us to take a hiatus from our dear *Treehouse*. We're not exactly sure how long, but for the next several months, we'll be dedicating our energies completely toward our shiny new adventure. Till we return, we've decided to release content periodically on our site, old and new. Until, then, thanks everyone for a great half trip.

treehouse.

April 2006

Volume II. Issue IV.

Editor-in-Chief
Kevin Hale

Managing Editor
Ryan Campbell

Business Manager
Chris Campbell

Copy Editor
Elizabeth Dehnart

Our Mission
To make the web easier to improve.

Our Mantra
Give them our best and toss in a little fun while we're at it.

© 2005 Particletree Inc.
All rights reserved.

All scripts, however, are managed under a Creative Commons License and individual articles are the copyright of their respective owners.

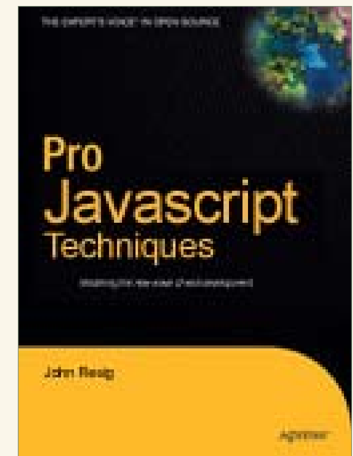
digest.

"Don't worry about what anybody else is going to do.
The best way to predict the future is to invent it."

Alan Kay

How to Build a Blog Out of JavaScript

By John Resig



The following is a chapter from John's upcoming book *Professional Javascript Techniques*, due to be released Fall of 2006. More information can be found on the book's site at http://jspro.org.

A live demo of this application can be found on the *Pro JavaScript* book web site at:

http://jspro.org/demo/blog/

With AJAX and MVC frameworks (such as Ruby on Rails or Django) flying to the forefront of web application development, it's as good a time as ever to take a step back and look at some upcoming alternatives. The example that I decided to explore was a simple, private, in-browser blog. It's very easy to use too; whenever you visit the page you're given your own private blog that you can view and post to. Additionally, there are live previews of your posts and a little SQL console that can be fun to play with.

What makes this project so unique is that the blogging engine logic is written entirely in Javascript, sending database queries directly to the server. While the demo isn't the most practical (having a blog that only you can see is sort of counter-intuitive), it does demonstrate a number of fundamental concepts necessary for developing a modern web application. The flow of the application moves something like this:

• A user visits the web page. If it's their first time, a new database, table, and test data will be created.

• All the user's posts will be requested from the server and returned as a JSON structure. The returned structure will then be transformed and inserted into the web site.

• The user then adds a new post to the blog, sending the data to the server. Upon completion, the new post is retrieved from the server (like before) and inserted into the site.

The application, itself, is broken down into two portions, client and server (as is the case with all AJAX applications) both of which I will be covering extensively. The client is responsible for interacting with the user and manipulating the interface. The server is responsible for managing the data source, a SQL database, and converting the data into a format that's

The Files

The whole application comes together in just a couple files, the majority of which goes toward operating the client.

index.html — The main application page, pulls all the client code together.

css/style.css — All the CSS for the application.

js/blog.js — The main Javascript code, responsible for binding events and running SQL queries.

js/sql.js — The code responsible for communicating with the server, retrieving the JSON from the SQL queries.

js/jsont.js — A local copy of the JSONT library.

js/textile.js — A copy of the Javascript Textile library (for converting text to HTML).

js/jquery.js — A copy of the current release of jQuery.

api/index.cgi — The main server code. This file could contain the Perl, Ruby, or Python versions of the server. If you wish to use the PHP version, rename this file to index.php.

data/ — Where all the SQLite user databases are stored.

usable by the client, JSON.

There are a couple concepts that I hope you'll be able to learn when walking through this particular example. First, that JSON is a powerful, and viable, alternative to using XML in web applications. Second, by keeping your server-side code as simple as possible you can give the front-end more control over what to do with the user data. Finally, all modern server-side scripting languages behave virtually identically and are rather easy to swap in and out (which is why I've written an identical version of the server-side code in the four most popular scripting languages: Perl, PHP, Python, and Ruby).

To begin our tour of the application, we need to start by looking at the various JSON-related technologies that I used and why I decided to use them over their XML counterpart.

The Client-Side Code

As discussed before, the client portion of the application is responsible for displaying the current blog posts, allowing a user to write a new post, and giving the user a simple console to interface with the database. This whole flow makes extensive use of JSON, wrapped up nicely with the easy-to-use jQuery library (which allows us to quickly bind events, handle AJAX requests, etc.). More information about the jQuery library can be found on its web site: jquery.com.

To walk through the client code, I'm going to assume that you already know the various AJAX technologies (XML, XMLHttpRequest, and maybe some XSLT) so that I can show you how the complementary JSON technologies behave. If you're unfamiliar with them, then this is a great time to learn the concepts, as JSON is a very simple technology to pick up and grasp.

JSON

The topic of JSON has been discussed a lot, mostly due to its overwhelming simplicity. Let's take a second to reexamine the basics of JSON, in the context of our application. At its core, JSON is a way of representing values (Strings and numbers), arrays of values, and hashes (key/value pairs) using Javascript-compatible object notation. For example, a portion of a JSON structure returned by the server to the blog, looks something like this:

Listing 15-1: A sample JSON structure, returned from the server. The structure is an array of hashes (each hash representing a row matched in the database). Each key/value pair in the hash represents a column name and column value within the row.

```
[
    {   title: "New Post",
        content: "Welcome to my wonerful blog!",
        date: "20060324122514" },

    {   title: "Another Post",
        content: "Lorem ipsum dolem…",
        date: "20060321101345" },
    …
]
```

If this JSON structure was in a variable named 'json' we could easily access the various posts using typical Javascript conventions.

```
// Get the first post
json[0]

// Get the title of the second post
json[1].title
```

The simplicity of navigating and accessing JSON structures is a huge win for Javascript developers. When you compare it to a similar piece of XML/DOM code, it would look something like this:

```
// Get the first post
document.getElementsByTagName("row")[0]

// Get the title of the second post
document.getElementsByTagName("row")[1]
    .getElementsByTagName("title")[0].firstChild.nodeValue
```

Within our blogging application, we make extensive use of JSON. Primarily, we use it as a way to format the database query results coming back to us from the server. To do this we use a JSON-formatting module (in our server-side language of our choice) and push our SQL query results through them, returning the formatted result to the client. The whole process is generally much simpler than attempting to push database results into a similar XML format, which is significantly helpful to developers.

The simplicity of JSON, alone, is generally enough to win most over to giving it a try. However, that's not all that JSON is good at. As we'll learn in the next section, it's also an incredibly fast means of loading data into a site.

Retrieving JSON

On-Demand Javascript centers on the technique of dynamically embedding new <script> tags into your site, during the course of normal operation, to load in JSON objects. The primary advantage of using this technique over using an XMLHttpRequest object is that you can make JSON requests to services that aren't on the same domain as yours. For example, using On-Demand Javascript you can dynamically load your favorite links, from the popular site del.icio.us since they provide access to your links formatted as a JSON object.

A second technique, that's used in conjunction with On-Demand Javascript, is that of JSONP (or JSON with Padding). In essence, you tell the server the name of a function that will receive the JSON object (typically as a 'callback' argument), and the server automatically wraps the returned object with the specified function name.

```
// A regular JSON object
{ cat: "meow", dog: "bark", mouse: "squeak" }

// A JSONP object
myFunction({ cat: "meow", dog: "bark", mouse: "squeak" })
```

A typical implementation of On-Demand Javascript and JSONP looks something like this (a derivative of this code is used in our application to send SQL queries to the server):

```
function loadJSON(url, callback) {
 // Save a pointer to the actual function that needs to be called back, on load
 jsonLoaded = callback;

 // Create a new script element, to embed into the header
 var script = document.createElement('script');
```

```
 // Set the URL of  the script to some
remote Javascript file
 script.src = url +
"?callback=jsonLoaded";

 // Add the new script to the page, thus
loading the remote JSON object
 document.getElementsByTagName("head")[0
].appendChild(script);
}
```

To use the above function, and to load your 'padded' JSON structure from the server, your code would look something like this:

```
// Request the JSON object
loadJSON("http://someurl.com/json.js",
function(json) {
    for ( var i in json )
        alert( json[i].title );
});
```

This technique has both its benefits and detriments. To its benefit, it's really fast – even faster than parsing an XML file of comparable size. To its detriment, it requires that the person implementing the server-side code add additional information to their output (which may not always be possible). Also, the validity and security of the code being returned cannot always be verified and may, in fact, be malicious in nature.

More information concerning On-Demand Javascript and JSONP can be found here:

- On-Demand Javascript
- JSONP
- Example of Yahoo's JSON API

In the case of our blogging application, we make use of the above techniques for most of our database queries, allowing us to quickly retrieve and traverse the results of a SQL operation. The only time in the application where we do not use On-Demand Javascript is when we're inserting new information into the database (e.g. Adding a new blog post). This is due to the fact that a user's post may be very long - long enough to not fit in a normal URL (using a HTTP GET Request). This means that we have to fall back to using XMLHttpRequest to send a POST with our SQL statement in it. This isn't a huge problem, but just something that we need to be aware of when working with On-Demand Javascript.

Formatting JSON

Up until recently, converting JSON into viewable HTML had been a relatively painful process, and a black mark on using JSON as an XML alternative. However, that time seems to have passed, with the recent release of a templating library called JSONT. The creator of JSONT, Stefan Goessner (goessner.net), wanted a templating system that was similar in nature to the combination of XML and XSLT (where the templating language, XSLT, was itself written in XML). As to with JSONT, the JSON templates are written in JSON itself.

JSONT is still very, very, new, but I think it shows a lot of promise, considering how powerful, and yet simple, it is. I'm going to start off by showing you a simple example of how JSONT can be used. First, we'll call the jsonT function, which takes two arguments. The first argument is the data that we want to transform. The second argument is the set of transformation rules. The function will then return the final transformed string.

```
jsonT(
  // An array of animals.
  [ "cat", "dog", "mouse" ],

  // Rules for formatting.
  {
    "self": "<ul>{$}</ul>",
    "self[*]": "<li>{$}</li>"
  }
);
```

In regards to the two rules, the 'self' rule applies to everything, and it's a great way to wrap the results. Having a {$} inside of a rule means "Keep applying rules to everything inside of me." The 'self[*]' rule matches each item in the global array. In this particular case, all we have is a string inside of the item, so we just print

that. The result of running this query is:

```
<ul><li>cat</li><li>dog</li><li>mouse</li></ul>
```

Now that we've seen a simple example of how JSONT can be used, let's take a look at how we use it in our application. The one time that we receive JSON objects, from the server, that we want to transform is when we do a SQL SELECT statement, requesting blog posts.

```
// Our generic JSON Template for transforming our
// blog post data structure into a viewable set of posts
var tmpl = {
    // Wrap all the posts in a <ul> ... </ul>
    "self": "<ul class='posts'>{$}</ul>",

    // Foreach post, display it as an <li> with the title in an <h2>
    // and the contents in a <div>. Also, transform the text
    // to textile on the fly.
    "self[*]": "<li><h2>{$.title}</h2>" +
        "<div class='content'>{@text($.content)}</div></li>",

    // Allow for the auto-conversion of text to textile
    // 'textile' is a function that converts text to textile
    "text": textile
};
```

What I've shown here is simply the rules used to transform our JSON blog post data structure (which was shown when talking about JSON). I didn't mention it explicity, but Textile is a formatting language for marking up text (so that you can stuff like put stars '*' around words to make them bold). Being able to do this sort of on-the-fly text conversion using other forms of templating is generally very difficult, and something that JSONT, thankfully, simplifies. So, using the provided blog post data structure, and our set of rules, we can get a transformation like this:

```
// Doing the following JSON Transform, on a single blog post
// Using our predefined template
jsonT([{
    title: "New Post",
    content: "Welcome to my *wonderful* blog!",
    date: "20060324122514"
}], tmpl );

// Gives us this as a result
<ul class='posts'>
 <li>
   <h2>New Post</h2>
     <div class='content'>Welcome to my <b>wonderful</b> blog!</div>
 </li>
</ul>
```

Using this defined template we can now quickly convert our incoming JSON strings to viewable HTML, virtually instantaneously. More information about JSONT can be found on the project site.

- The original JSONT article
- Lots of samples of JSONT in action
- Textile Overview
- Javascript Textile Implementation (used in this project)

To fully understand what JSONT is capable of, I recommend pouring through Stefan's examples and doing some reading on XSLT (if you haven't done so already).

The Server-Side Code

The server-side portion of this application has a very simple, and re-

producible, flow. The overall goal is to get a SQL query from the client, execute it against a SQLite database, and return the results formatted as a JSON string. To put this application into practice, I decided to refine the application flow and produce duplicate, working, versions of it in all the popular scripting languages (Perl, PHP, Python, and Ruby).

I assume that you have, at least, tinkered with one of those languages; and if not, this is a great time to learn by example. When building each individual application, there were a number of requirements that needed to be met in order to create very-similar programs.

• There needed to be support for SQLite (either built in to the language, or as a module or library).

• The ability to format objects/arrays/hashes as JSON (to format and return the SQL results, to the client).

• Support for handling CGI requests and Regular Expressions are required (getting the SQL query from the client and making sure that the database doesn't have any malicious characters in it).

• Finally, some level of Exception handling is needed (try/catches) to watch for errors in the client's SQL statements.

Demonstration

JavaScript Blog

Welcome to my Javascript-powered blog! This blog engine was written using nothing but Javascript. The backend is just a thin proxy to a SQLite database, which can be written with Perl, Python, Ruby, or PHP.

Thankfully, every one of the selected languages matches these points, to some degree. Rather than walk you through each individual application (which are each highly documented, and even use the same variable names) – I'd like to discuss some of the features that I found to be particularly interesting.

SQLite

SQLite is an incredibly lightweight, but full-featured, SQL database. SQLite works by running off of a single file on your system, so you can have as many databases as you have files. In addition to being really fast, SQLite serves as a really quick and simple way of setting up a database for simple applications or testing. Rather than require a large database install (such as MySQL, PostgreSQL, or Oracle), SQLite will serve our needs just fine.

Every language that I explored (Perl, PHP, Python, and Ruby) had support for SQLite, in one way or another.

• Perl has the DBD::SQLite module. This module is particularly notable because the developers decided to completely implement the SQLite specification inside of the module itself – meaning that no additional downloads are required to get a database up and running.

• PHP 5 was the only language to have built-in SQLite support. Unfortunately, the support was only for SQLite 2 (which is fine for some applications) – but I wanted complete compatibility between the different code bases, so I had to install a SQLite 3 library instead.

• Python and Ruby both have SQLite libraries that hook in directly with the official SQLite installation.

I highly recommend that you explore using SQLite in some of your smaller projects; it's a great way to get up-and-running when you don't need the overhead of a large database installation.

JSON Formatting

As with SQLite, every popular scripting language has an implementation of JSON serialization (taking a native object and converting it to a JSON string). Additionally, since the output is very lightweight, as is the case with most JSON, it becomes very easy to output to the browser. Incidentally, each language's implementation ends up being very similar to one another, simplifying the transition process from one language to another.

• Each implementation is available in the form of a library or module.

• They all accept native language objects (e.g. String, Arrays, and Hashes).

• Each implementation makes it easy to get at a string of the JSON-formatted object.

However, the one language that implements JSON serialization particularly elegantly is Ruby. Below is an example of how you would convert an object to a JSON string (after loading in JSON library):

```
# Convert the object (r) to a JSON
string, and print it
print r.to_json
```

Working with SQLite and JSON are really the only two, potentially new, topics that you'll have to worry about when looking at the server code. I highly recommend that you look over the code for the server implementation, in your favorite language, and start learning from there.

John Resig is a programmer and entrepreneur who's currently authoring the book 'Pro Javascript Techniques,' which is due out in the Fall of 2006. When he's not programming, he enjoys watching movies, writing in his web log (ejohn.org), and spending time with his girlfriend Julia.