**Dependency Language in JS / Shade2 in HTML5**

**Development & debugging:**

The project is available at <https://twkdv01.nam.nsroot.net:8443/svn/repos/HTML-Shade>.

[WebStorm 8](http://www.jetbrains.com/webstorm/) was used for development and debugging of the application on the server and browser. The Chrome Dev Tools were also used to debug the application in the browser. Firebug should work as well.

In addition to JavaScript, parts of the project (mostly the Angular.js parts) are written in CoffeeScript, which compiles to JavaScript and offers a different style (similar to Python) as well as enforcing some practices and offering extra features. Jade, which does the same for HTML, is used for the index page.

**Description of the project:**

There are 5 components:

1. Server/client-side JavaScript implementation of DL
2. Client-side JavaScript Shade-to-HTML compiler.
3. Client-side JavaScript Angular application that renders the HTML output of the compiler.
4. An editor application that takes DL and Shade and renders HTML.
5. A node.js server that serves the application.

**Dependency Language in JavaScript**

This is a basic implementation of DL. We use compilers described in YACC/LEX and created in JavaScript using Jison. Adding core syntax to the implementation is as simple of editing the YACC/LEX file and giving it to Jison. There are two separate compilers for the initial parsing of the DL file and for evaluating an expression to find a variable’s value, but the same compiler can be used for both. Most of DL’s features, other than dependencies are not implemented as focus was shifted to Shade after the basic dependency functionality of DL was implemented. However, when designing the structure of the DL compiler I took future improvements into consideration and development should be relatively easy.

The implementation was originally done on the node.js server, but using browserify I migrated it to the client. This can be done seamlessly, by running browserify on the startup .js file in the folder and using the output .js file in the application.

**Client-side JavaScript Shade-to-HTML compiler**

We convert Shade XML to HTML in 3 steps:

1. Convert XML Shade to JSON using x2js
2. Convert Shade controls to HTML elements described in JSON
3. Use a templating engine to convert the JSON object to HTML.

There is a list of attribute name conversions from Shade to HTML in an object. Adding new attributes to these objects is all that’s needed to have them appear in the output HTML. For example:

This means that if the parser sees a ‘Bg’ attribute in a control, it will add a class to the element converted from that control that corresponds to a CSS class with a 'background-color' attribute. Conversions can be made on the attribute/style names as well as their values, if needed. For example:



Converts to:

By appending ‘px’ to the value in the <Height> tag.

**Client-side Angular**

Using Angular we can define reusable, custom HTML elements, with whatever functionality we need. For example, instead of using HTML’s <select> element, which is not as versatile as the drop-downs used in WPF/Shade, an Angular directive called ‘dropDown’ is used, which mimics a drop down using a selection field and a table.

**An editor application that takes DL and Shade and renders HTML.**

The editor is modeled after the Shade Editor in .NET. It has a DL panel and Shade panel for code input, output panels for each of these and a panel to show the resulting HTML page. It’s built using Angular.

**A node.js server**

Right now, this simply serves the application index.HTML, all logic is done on the client (although the DL part was originally made to run on the server, and later moved to the client using browserify). In the future calls to external resources such as SQL queries can be done using node.js and SSH tunneling, for example.

**Testing**

Jasmine was chosen as the testing environment for both end-to-end and unit tests. Some tests are implemented, but the angular part of the application (the Editor and Shade components) are not currently tested. Jasmine was developed alongside angular so setting up tests for angular shouldn’t be difficult. Karma is another testing framework that allows to test on virtual machines, imitating different browsers and platforms rather than having to test on each browser separately.

**Future development**

Adding the rest of DL’s core functionality – things like lazy evaluation and option explicit are not implemented. The implementation using JavaScript’s native types rather than the ones defined in the DL documentation.

Adding DL’s built in functions – All that is necessary is adding the functions to the graph-dev/functions.js file and they will be usable in DL. Currently only 3 functions are implemented.

Adding more controls to Shade – Several controls are implemented, namely the ones in the basics mock-up in the Shade editor help files. New controls can be added to Shade/shadeCompiler/shadeHandlers.js directly or using an external file, like is done with Grid.js and DropDown.js.

SQL queries – The application is served with node.js. The idea was to have developers deploy the application to the server, and clients would access the deployed version from the browser. SQL queries can be made with an SSH tunnel (a proxy to the SQL server), a functionality that exists for node.js as an external library.

.NET Shade libraries – To implement functions from the Shade libraries for use with #bind, we can choose one of two approaches. One is to set up a server that can accept HTTP requests and run the functions, returning the result, or re-implementing the libraries in JavaScript.

Responsive design – Using CSS media queries and optionally some JS libraries such as twitter bootstrap, it is possible and desirable to implement responsive designs – ones that change to accommodate the type of device they are run on. Finding a library that fits Shade’s exact requirements is unlikely however, and an implementation of such a library specific to Shade is likely to be required.

Error handling – Currently, any errors reported are reported through whatever native error handling is available (such as Angular’s or the Jison compilers’ native error handling).

List of resources used:

1. [Jison](http://zaach.github.io/jison/)
2. [Node.js](http://nodejs.org/) (with [express.js](http://expressjs.com/))
3. [Angular.js](https://angularjs.org/)
4. [Jasmine](http://jasmine.github.io/)
5. [X2JS](https://code.google.com/p/x2js/)
6. [Browserify](http://browserify.org/)
7. [Lo-Dash](http://lodash.com/)
8. [CoffeeScript](http://coffeescript.org/)
9. [Jade](http://jade-lang.com/)