Sync JS

Proposed Spec for JavaScript Web framework

Goals…

* Simple, Markup Driven
* Based on conventions
* Attempt to Reduce bandwidth 50%-90%
* Minimal code count
* Single Page / Full Ajax – No browser requests, No reloads
* Minimal development effort/costs
* Back/Front button support
* Deep linking, New Tab/Window support
* Low learning curve
* Downgradable, SEO Support
* Multiple update types (Windows, Table Rows, etc.)
* Unobtrusive JavaScript
* HTML5 Compliant
* Web Server Agnostic
* Client-side Template support (Coming Soon)
* Local Storage / Caching support (Coming Soon)
* Dependency Management (Coming Soon)
* MVC Support (Coming Soon)

Example of current framework can be viewed at [WorkPlex.com](http://workplex.com).

The purpose of the “Sync” framework (working title) is as a light-weight, easy-to-use, markup driven framework built on top of jQuery. The intent is to make building highly ajaxified web application much easier and cost effective by employing a meta-programming methodology to greatly reduce the overall code count.

The goal is to create a framework that embraces both JavaScript and HTML. It makes the need for models and controllers completely optional on an as-needed basis. It is highly dependent on conventions, but can be completely customized. It heavily offsets functional behavior from the server to the client, while retaining the dynamic, on-demand nature of the web. HTML and scripts are loaded when and only when needed.

Alternative Names:

* jSync
* Updater

**Hello World**

Full page is only loaded once, and plugin is initialized. The default content area has the id “content”, but can be configured.

<html>

<head>

<script type="text/javascript" src="/Scripts/jQuery.js"></script>

**<script type="text/javascript" src="/Scripts/Sync.js"></script>**

**<script type="text/javascript">**

**$(function () {**

**Sync.init();**

**});**

**</script>**

</head>

<body>

<div **id="content"**>

**<a href="/Samples/HelloWorld">Hello World</a>**

</div>

</body>

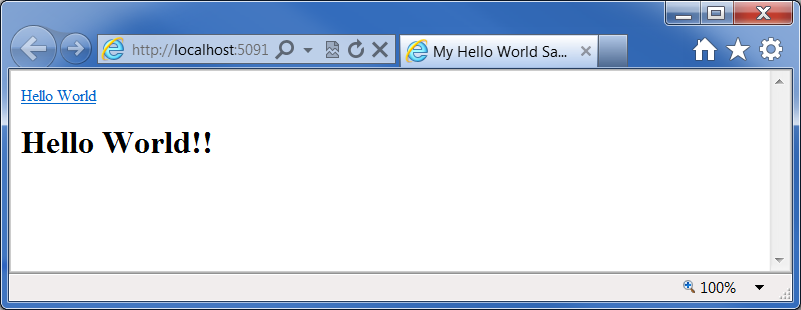
</html>

Links and Forms are automatically ajaxified and partial HTML updates returned from server are automatically updated depending on HTML5 metadata.

<div **data-update="content" data-title="My Hello World Example"**>

<h1>Hello World!!</h1>

</div>



**More Update Types**

Show content in a model window.

<div **data-update="window" data-title="My Window" data-model="true" data-width="500"**>

<h1>My Window</h1>

</div>

Replace an existing element

<div id="my-div" **data-update="replace"**>

<h1>My Updated Text</h1>

</div>

Insert content in an element

<div **data-update="insert" data-target="#parent-div"**>

<h1>My Updated Text</h1>

</div>

Add or replace a row in a table

<tr id="my-row" **data-update="row"**>

<td>My Updated Row</td>

</tr>

Other update types currently include **append**, **prepend, top, bottom** and **subrow**.

**Automatic URL Management**

Automatic support for deep linking, open in new window/tab and back/front button support is built in. Currently hash URL’s “#” are used to manage linking. HTML5 pushState will be used in the future. An address for a particular view can be defined in the markup.

<div data-update="content" **data-address="/MyCustomUrl"**>

<h1>My Updated Text</h1>

</div>

**Markup -Driven Behavior**

Behaviors can be defined using metadata rather than writing code.

<div **data-request="/MyUrl/View"**>Click Me to Make a Request</div>

<div data-update="append" data-target="main-div" **data-hide="#main-menu" data-show="#user-menu"**>

<h1>My Updated Text</h1>

</div>

**Script and Dependency Management**

Inline scripts are automatically executed **after** updates are made.

<div id="my-div" data-update="content">

...

</div>

**<script>$("my-div").doSomthing();</script>**

Script dependencies are automatically handled also. The dependencies are downloaded **before** updates are made.

Defined via script reference tags…

**<script src="/Scripts/Dependency.js"></script>**

<div id="my-div" data-update="content">

...

</div>

Defined via metadata…

<div id="my-div" data-update="content" **data-load="Dependency.js"**>

...

</div>

**Client-side Templates / Caching** (Coming Soon)

The framework would use a default templating engine, but could be configured to use others. Models would effectively be passed to the templates via the “this” keyword. For example, when a link is clicked, say “/Employee/Details/123”, a window with the employees skills would be displayed.

First, the framework checks to see if any template is cached in memory or local storage associated with the route “/Employee/Details/{id}”. If not, the template is requested by making a request for “text/html”. Once the framework has the template, every other request to the route is made for “application/json” to ensure that only the data is retrieved. Templates id’s are defined with the data-template attribute, and associated with a route via the data-route attribute.

<div id="employee<%= this.EmployeeID %>" **data-template="employee-details" data-route="/Employee/Details/{id}"** data-update="window" data-title="<%= this.FullName %>>

<label>Name: </label> <%= this.FullName %>

<label>Title: </label> <%= this.Title %>

<label>Skills:</label>

<ul>

<% $(this.Skills).each(function(i) { %>

<li><%= this %></li>

<% }); %>

</ul>

</div>

**Caching** (Coming Soon)

Updates and templates can be cached in either local storage or session, since the page normally doesn’t refresh. Templates are automatically stored automatically, and default settings can be configured.

Cache in session memory …

<div data-update="content" data-cache="session">

…

</div>

Cache in local storage for 10 days…

<div data-update="content" data-cache="local; 240">

…

</div>

**High-Level Events** (Coming Soon)

Simple events such as making an Ajax requests after link clicks, form posts and URL changes are fully automatic. Higher level events can be attached to elements though. Pages could have global events (document level) or local events (element) level. Here are some examples of these events.

* requestStarted – A request was started
* success – A request was successful
* invalid – A form post was submitted with invalid values
* beforeUpdate – An element in the response is about to be updated in the DOM
* afterUpdate – An element in the response was just updated in the DOM
* complete – A request was completed
* error – A request returned an error response

//Form submit was successful

$(form)**.success**(function (result) {

//Show growl alert

alert("Your employee details have been saved", "success");

//Render details and display

$.**render**("employee-details", result);

});

**Configurable Attribute Prefixes** (Coming Soon)

Hate the “data-“ syntax? The attributes should be configured to use either “data-“ or custom namespaces.

<div **data-update="window" data-title="My Window"**>

<h1>My Window</h1>

</div>

<div **x:update="window" x:title="My Window"**>

<h1>My Window</h1>

</div>

**Configuration**

The framework will be based on conventions, but will be highly customizable.

//Initialize sync

Sync.init({

autoEvents: true, //Automatically hijax every link and form

autoCorrectLinks: true, //Change standard URL's to ajax (#) URL's

contentId: "content", //The main content area where content is rendered

scriptLocation: "/Scripts", //Path to download dependent scripts from

topContentId: "content-top", //The content area right above the main content

bottomContentId: "content-bottom", //The content area right below the main content

progressId: "progress", //Progress indicator id

progressText: "One Moment...", //Progress indicator text

progressCss: "progress", //Progress indicator CSS style

pageTitlePrefix: "", //Prepend to title of each page

submitFilter: ".placeholder", //Don't submit any form elements that match this

//Request event

onRequest: function (url, sender, formData) { },

//Success event

onSuccess: function (result) { },

//Before update event

onBeforeUpdate: function (update, meta) { },

//After update event

onAfterUpdate: function (update, meta) {

//Initialize any scripts or plug-in

InitView(update);

},

//Complete event

onComplete: function (navKey) {

//Set menu

SetMenu(navKey);

},

//Error event

onError: function (result) {

alert("An unexpected error has occurred.");

}

});

**Other Features**

* Automatic progress indicator
* Automatic disabling of forms/links
* Double submit Prevention
* Change standard links to Ajax links
* Set page title
* Manage navigation keys/values (for setting menus)
* Form autofocus, controlled focus

**Sample Use Case – Task List**

Initial page is loaded and Sync.js is initialized. Links/Forms are hijaxed.

<html>

<head>

<script type="text/javascript" src="/Scripts/jQuery.js"></script>

<script type="text/javascript" src="/Scripts/Sync"></script>

<script type="text/javascript">

$(function () {

Sync.init();

});

</script>

</head>

<body id="content">

<a href="/Tasks">Task List</a>

</body>

</html>

When the “Task List” is clicked, the first request is made to “/Tasks”, and templates for the task list are returned. Once the templates are returned, another request is made to the route sending the “application/json” header so that only data is returned. The templates are then automatically rendered and updated.

<div data-template="task-list" data-update="content" data-title="My Task List" data-route="/Tasks/{?page}">

<a href=" /Task/Add">Add Task</a>

<table>

<%= $.render("task-row", this) %>

</table>

</div>

<tr data-template="task-row" id="task-form<%= this.id %>">

<td><a href="/Task/Edit/<%= this.TaskID %>"></a></td>

<td><%= this.Title %></td>

<td><%= this.Priority %></td>

</tr>

When the user clicks the “Add Task” link, the client requests the form template and automatically displays it in a model window. The TaskController.js dependent script file is also automatically loaded. After the users fills in the form and hits submit, the POST is automatically handled.

<form data-template="task-form" data-function="TaskController.form" data-load="TaskController" data-update="window" data-title="Add Task" data-route="/Task/Add|/Task/Edit/{id}">

<input name="Title"/>

<input name="Priority"/>

<button type="submit">Save Task</button>

</form>

When the form is added to the DOM, the action method on the TaskController is run, which adds high-level events, “success” and “invalid” to the form. On a successful submit, a success alert is shown and the new row is rendered and updated. In the controller and the action functions, “this” always refers to the view.

var TaskController = function () {

var form = function () {

//Successful submit

this.success(function (result) {

alert("Task has been saved.", "success");

$.render("task-row", result);

});

//Invalid submit

this.invalid(function (errors) {

var summary = $.render("validation-summary", errors);

this.append(summary);

});

};

}

If the submit is invalid, the “invalid” event renders and updates the validation summary with the list of errors.

<div data-template="validation-summary" class="error">

<b>Please correct the following errors</b>

<ul>

<% $(this.Errors).each(function() { %>

<li><%= this %></li>

<% }); %>

</ul>

</div>