Fast and efficient

Developing jQuery plugins with ease

Jakob Westhoff <jakob@php.net>

WebTech Conference
12. October, 2010

Jakob Westhoff

@jakobwesthoff

`http://westhoffswelt.de`

About You?

Questions?

Questions!

What comes next?

jQuery

jQuery about itself

From the jQuery Website:

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.

jQuery about itself

From the jQuery Website:

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.

jQuery about itself

From the jQuery Website:

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.

jQuery about itself

From the jQuery Website:

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.

jQuery about itself

From the jQuery Website:

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript.

Introduction to jQuery

- Compact
  - only 71kb minified
  - 24kb minified and gzipped
- Cross-browser compatible
  - IE 6.0+, FF 2+, Safari 3.0+, Opera 9.0+, Chrome
- Easily extendable

Working with jQuery

```
$(".tooltip").addClass("highlight").fadeIn("slow");
```

Working with jQuery

```
$(".tooltip").addClass("highlight").fadeIn("slow");
```

- Document centric
- Operates on sets accessed using `$` or `jQuery`
  - `$(css selector).operation`
  - `jQuery(css selector).operation`
- Fluent interface paradigm
  - `operation().operation().operation()`

What comes next?

Plugins

Plugintypes

- Set methods (`jQuery.fn`)
- Utility functions (`jQuery`)
- Special event handler
- jQuery.UI widgets
- jQuery.UI behaviors
- jQuery.UI effects
- CSS-Selectors
- Easing functions (Animation)

Plugintypes

- Set methods (`jQuery.fn`)
- Utility functions (`jQuery`)
- Special event handler
- jQuery.UI widgets
- jQuery.UI behaviors
- jQuery.UI effects
- CSS-Selectors
- Easing functions (Animation)

Set methods

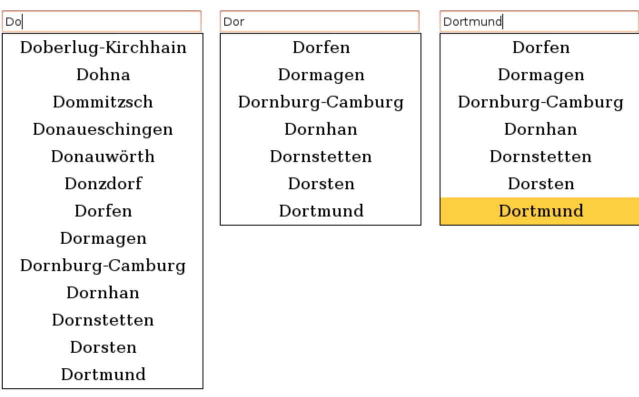
- The most common plugin type
- Extends the capabilities of every jQuery set
- Think of it as a new functionality in the style of:
  - `addClass`
  - `css`
  - `attr`
  - ...

What comes next?

Case study

A simple Livesearch example

- Livesearch enhancement
  - Type ahead suggest, Autocomplete, ...
- Used in nearly all Web2.0 applications



What comes next?

Requirements

Plugin Requirements

- Applicable to every `input` element of type `text`
- Configurable by different options
  - Target URL to communicate with
  - Minimum length to start a search
  - ...
- Work in conjunction with `jQuery.noConflict()`
- Obey to the fluent interface
- Handle keypress events to:
  - execute the search
  - navigate and select the results

Plugin Requirements

- Applicable to every `input` element of type `text`
- Configurable by different options
  - Target URL to communicate with
  - Minimum length to start a search
  - ...
- Work in conjunction with `jQuery.noConflict()`
- Obey to the fluent interface
- Handle keypress events to:
  - execute the search
  - navigate and select the results

What comes next?

Filenames

Filename conventions

- Name of this plugin: `livesearch`
- All plugins should follow a certain filename convention:
  - `jquery.livesearch.js`
  - `jquery.livesearch.css`
  - `jquery.livesearch/image.png`
  - ...

What comes next?

Building blocks

The problem with the $ shortcut

- $ should not be used in plugins
  - jQuery allows redefining of this alias for compatibility reasons
  - `$.noConflict()`
  - May break your plugin
- Always use `jQuery`

The problem with the $ shortcut

- Or use nifty workaround:

```
1 (function($) {
2     ...
3     // Your plugin code goes here
4     ...
5 })(jQuery);
```

Registering the Plugin Method

- Register new methods on `jQuery.fn`
- Available on all sets automatically
- Stick to one method per plugin
- Pluginname == methodname

Registering the Plugin Method

```
1 jQuery.fn.livesearch = function(options) {
2     ...
3     // Plugin method
4     ...
5 }
```

Handling Options

- Most plugins need configuration options
- Use a default if a certain option is not provided
- Options are supplied as associative array / object

Default Options

- Defaults should exist for every available option
- Make defaults accessible from the outside
- Allows for different levels of customization
  - Coarse grained: On the application level
  - Fine grained: For each plugin call
- Use the `defaults` property on the plugin function

Default Options

```
1  jQuery.fn.livesearch.defaults = {
2    "source": [],
3    "minLength": 2,
4    "searchDelay": 600,
5        ...
6  };
```

Handling Options

- Utilize `jQuery.extend`
- Easy way to combine object properties
- Prioritizes input by argument order
- First given object will be modified
- Returns merged object

Handling Options

```
1  options = jQuery.extend(
2    {},
3    jQuery.fn.livesearch.defaults,
4    options
5  );
```

Access to the targeted set

- Access the targeted elements using `this`
- points to a `jQuery` set
- may contain zero, one or multiple element(s)

Handling an arbitrary amount of elements

- Use `each` iterate sets
- Correct handling of zero, one or multiple entries
- Return value obeys fluent interface and can be passed-through
- Use `filter` to remove unwanted elements

Handling an arbitrary amount of elements

```
1  return this
2    .filter( "input[type=text]" )
3    .each(function( index, value ) {
4      ...
5      // this  == current handled element
6      // value == current handled element
7      // index == current index in the set
8      ...
9    });
```

What comes next?

Foundation

What we have got so far

```
1   // Compatibility with $.noConflict()
2   (function($) {
3
4     // Register new method for every set
5     jQuery.fn.livesearch = function(options) {
6
7       // Option handling
8       options = jQuery.extend(
9         {},
10        jQuery.fn.livesearch.defaults,
11        options
12      );
13
14      // Ensure the fluent interface paradigm.
15      // Filter unwanted elements
16      // Handle sets correctly.
17      return this
18        .filter( "input[type=text]" )
19        .each(function() {
20          // "real" plugin code goes in here
21          ...
22        });
23    }
24
25    // Default options accessible from the outside
26    jQuery.fn.livesearch.defaults = {
27      ...
28    };
29
30  })(jQuery);
```

What comes next?

Event namespaces

Event namespaces - a mostly unknown feature

- Group registered events by a certain identifier
- As usual use `bind` for listening to events
- Namespaces and event types separated using a dot (`.`)
  - eg. `click.namespace`
- Namespaces allow easy de-registration/management
  - `.unbind("click.namespace")`
  - `.unbind(".namespace")`

Event namespaces in plugin development

- Always use event namespaces inside your plugins
- Plugin name makes a good identifier
- No conflict between your event handlers and others
- Easy de-registration possible without remembering the callback

Register events with namespaces

```
1 $(this).bind(
2   'keydown.livesearch',
3   function(event) {
4     switch( event.keycode ) {
5         ...
6     }
7   }
8 );
```

What comes next?

Scopes

Scopes in JavaScript

- Scopes in JavaScript are maintained on a function level
- A variable declared using `var` is local to its function
- Variables are assumed local even if they are used before being declared with `var`
- A variable used without `var` is assumed to be global
- Functions are always local to the scope they are defined in
- Variables from an outside scope can be overridden with local ones

Scopes in JavaScript

```
1   // Global variable
2   foo = 42;
3
4   // Local variable
5   var bar = 23;
6
7   // Functions are always local to the scope they reside in
8   function outside() {
9     // Local to scope of function "outside"
10    function inside() {
11      ...
12    }
13  }
14
15  // Position of var statement is not important
16  function hello() {
17    // Variable world is local to this function
18    world = "Hello World!";
19    var world;
20  }
21
22  // Inner scopes are always prioritized
23  var baz = 42;
24  function theAnswer() {
25    // baz is local to this function and contains a string
26    var baz = "Answer to the Ultimate Question of Life, the Universe, and
          Everything";
27  }
28  // baz is still 42 outside this function
```

Scopes in JavaScript

```
1  // Global variable
2  foo = 42;
3
4  // Local variable
5  var bar = 23;
6
7  // Functions are always local to the scope they reside in
8  function outside() {
9    // Local to scope of function "outside"
10   function inside() {
11     ...
12   }
13 }
14
15 // Position of var statement is not important
16 function hello() {
17   // Variable world is local to this function
18   world = "Hello World!";
19   var world;
20 }
21
22 // Inner scopes are always prioritized
23 var baz = 42;
24 function theAnswer() {
25   // baz is local to this function and contains a string
26   var baz = "Answer to the Ultimate Question of Life, the Universe, and
       Everything";
27 }
28 // baz is still 42 outside this function
```

Scopes in JavaScript

```
1  // Global variable
2  foo = 42;
3
4  // Local variable
5  var bar = 23;
6
7  // Functions are always local to the scope they reside in
8  function outside() {
9    // Local to scope of function "outside"
10   function inside() {
11     ...
12   }
13 }
14
15 // Position of var statement is not important
16 function hello() {
17   // Variable world is local to this function
18   world = "Hello World!";
19   var world;
20 }
21
22 // Inner scopes are always prioritized
23 var baz = 42;
24 function theAnswer() {
25   // baz is local to this function and contains a string
26   var baz = "Answer to the Ultimate Question of Life, the Universe, and
       Everything";
27 }
28 // baz is still 42 outside this function
```

Scopes in JavaScript

```
1   // Global variable
2   foo = 42;
3
4   // Local variable
5   var bar = 23;
6
7   // Functions are always local to the scope they reside in
8   function outside() {
9     // Local to scope of function "outside"
10    function inside() {
11      ...
12    }
13  }
14
15  // Position of var statement is not important
16  function hello() {
17    // Variable world is local to this function
18    world = "Hello World!";
19    var world;
20  }
21
22  // Inner scopes are always prioritized
23  var baz = 42;
24  function theAnswer() {
25    // baz is local to this function and contains a string
26    var baz = "Answer to the Ultimate Question of Life, the Universe, and
          Everything";
27  }
28  // baz is still 42 outside this function
```

Scopes in JavaScript

```
1   // Global variable
2   foo = 42;
3
4   // Local variable
5   var bar = 23;
6
7   // Functions are always local to the scope they reside in
8   function outside() {
9     // Local to scope of function "outside"
10    function inside() {
11      ...
12    }
13  }
14
15  // Position of var statement is not important
16  function hello() {
17    // Variable world is local to this function
18    world = "Hello World!";
19    var world;
20  }
21
22  // Inner scopes are always prioritized
23  var baz = 42;
24  function theAnswer() {
25    // baz is local to this function and contains a string
26    var baz = "Answer to the Ultimate Question of Life, the Universe, and
          Everything";
27  }
28  // baz is still 42 outside this function
```

Impact of scopes on plugins

- Create private functions and variables
- Don't pollute the global scope
- Allow certain properties to be overridden from the outside
  - Default options (`jQuery.fn.livesearch.defaults`)
- Structure your code inside the plugin by creating
  - objects
  - functions
  - plugin visible variables

Impact of scopes on plugins

```
1  (function($) {
2
3    // Everything in here is global to the plugin,
4    // but confined to its own scope
5    var foo = 42;
6    function bar() {...};
7    var baz = {...};
8
9    jQuery.fn.livesearch = function(options) {
10     // Everything in here is local to the plugin function.
11     // This allows for example access to the options variable
12     var foo = 42;
13     function bar() {...};
14     var baz = {...};
15   }
16
17     // Export something to the outer scope without polluting its scope
18     jQuery.fn.livesearch.defaults = {...};
19     jQuery.fn.livesearch.foo = 42
20     jQuery.fn.livesearch.bar = function() {...};
21
22 })(jQuery);
```

Impact of scopes on plugins

```
1   (function($) {
2
3     // Everything in here is global to the plugin,
4     // but confined to its own scope
5     var foo = 42;
6     function bar() {...};
7     var baz = {...};
8
9     jQuery.fn.livesearch = function(options) {
10      // Everything in here is local to the plugin function.
11      // This allows for example access to the options variable
12      var foo = 42;
13      function bar() {...};
14      var baz = {...};
15    }
16
17      // Export something to the outer scope without polluting its scope
18      jQuery.fn.livesearch.defaults = {...};
19      jQuery.fn.livesearch.foo = 42
20      jQuery.fn.livesearch.bar = function() {...};
21
22  })(jQuery);
```

Impact of scopes on plugins

```
1   (function($) {
2
3     // Everything in here is global to the plugin,
4     // but confined to its own scope
5     var foo = 42;
6     function bar() {...};
7     var baz = {...};
8
9     jQuery.fn.livesearch = function(options) {
10      // Everything in here is local to the plugin function.
11      // This allows for example access to the options variable
12      var foo = 42;
13      function bar() {...};
14      var baz = {...};
15    }
16
17      // Export something to the outer scope without polluting its scope
18      jQuery.fn.livesearch.defaults = {...};
19      jQuery.fn.livesearch.foo = 42
20      jQuery.fn.livesearch.bar = function() {...};
21
22  })(jQuery);
```

Impact of scopes on plugins

```
1   (function($) {
2
3     // Everything in here is global to the plugin,
4     // but confined to its own scope
5     var foo = 42;
6     function bar() {...};
7     var baz = {...};
8
9     jQuery.fn.livesearch = function(options) {
10      // Everything in here is local to the plugin function.
11      // This allows for example access to the options variable
12      var foo = 42;
13      function bar() {...};
14      var baz = {...};
15    }
16
17      // Export something to the outer scope without polluting its scope
18      jQuery.fn.livesearch.defaults = {...};
19      jQuery.fn.livesearch.foo = 42
20      jQuery.fn.livesearch.bar = function() {...};
21
22  })(jQuery);
```

Impact of scopes on plugins

```
1  (function($) {
2
3    // Everything in here is global to the plugin,
4    // but confined to its own scope
5    var foo = 42;
6    function bar() {...};
7    var baz = {...};
8
9    jQuery.fn.livesearch = function(options) {
10     // Everything in here is local to the plugin function.
11     // This allows for example access to the options variable
12     var foo = 42;
13     function bar() {...};
14     var baz = {...};
15   }
16
17     // Export something to the outer scope without polluting its scope
18     jQuery.fn.livesearch.defaults = {...};
19     jQuery.fn.livesearch.foo = 42
20     jQuery.fn.livesearch.bar = function() {...};
21
22 })(jQuery);
```

Formatting

CSS formatting in plugins

- Use CSS rules for visual formatting if possible
- Decoupling of functionality and representation
- Store CSS in appropriate position `jquery.livesearch.css`
- Always use classes not ids (multiple invocation)
- Prefix all used identifiers with the pluginname (`livesearch-container`)
- Follow the hierarchy of your elements (`livesearch-container-element`)

What comes next?

Preparations

Before you start writing your own plugin

- Before you write your own plugins
  - Check `http://plugins.jquery.com/`
- Read Plugins/Authoring documentation
  - `http://docs.jquery.com/Plugins/Authoring`
- jQuery docs are always a good resource
  - `http://docs.jquery.com`

Want to learn more?

- Buy "Pluginentwicklung mit jQuery" (German)
- `http://entwicklerpress.de/jquery`

Conclusion

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`
2. Prefix your CSS classes with the plugin name, follow the hierarchy

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`
2. Prefix your CSS classes with the plugin name, follow the hierarchy
3. Obey the fluent interface

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`
2. Prefix your CSS classes with the plugin name, follow the hierarchy
3. Obey the fluent interface
4. Handle sets of zero, one or multiple elements (`each`)

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`
2. Prefix your CSS classes with the plugin name, follow the hierarchy
3. Obey the fluent interface
4. Handle sets of zero, one or multiple elements (`each`)
5. Be aware `$` might not be what you think it is

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`
2. Prefix your CSS classes with the plugin name, follow the hierarchy
3. Obey the fluent interface
4. Handle sets of zero, one or multiple elements (`each`)
5. Be aware `$` might not be what you think it is
6. Handle events: Always use namespaces

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`
2. Prefix your CSS classes with the plugin name, follow the hierarchy
3. Obey the fluent interface
4. Handle sets of zero, one or multiple elements (`each`)
5. Be aware $ might not be what you think it is
6. Handle events: Always use namespaces
7. Make default options accessible from the outside

8 Golden Rules of Plugin Development

1. Name your file `jquery.[insert name of plugin].js`
2. Prefix your CSS classes with the plugin name, follow the hierarchy
3. Obey the fluent interface
4. Handle sets of zero, one or multiple elements (`each`)
5. Be aware $ might not be what you think it is
6. Handle events: Always use namespaces
7. Make default options accessible from the outside
8. Structure your code without polluting outside scopes (Scoping)

Thanks for listening

Questions, comments or annotations?

Slides: http://westhoffswelt.de/portfolio.htm

Contact: Jakob Westhoff <jakob@westhoffswelt.de>
Follow Me: @jakobwesthoff

What comes next?

Unit testing

Unit testing with QUnit

- Unit testing? YES you want to!

Unit testing with QUnit

- Unit testing? YES you want to!
- QUnit
- jQuerys unit testing framework
  - `http://docs.jquery.com/QUnit`
- No stable release, yet
- However core features are extremely stable

QUnit test runner

- Runner executed by opening a webpage
- Results printed to this page
- Needed files: `qunit.js`, `qunit.css`
- Container elements to store the results need to be present

QUnit test runner

```
1  <html>
2
3    <head>
4      <link rel="stylesheet" href="qunit/qunit.css" type="text/css" />
5
6      <script type="text/javascript" src="qunit/qunit.js"></script>
7      <!-- Further scripts containing plugin and tests -->
8    </head>
9
10   <body>
11     <h1 id="qunit-header">Livesearch Plugin</h1>
12     <h2 id="qunit-banner"></h2>
13     <h2 id="qunit-userAgent"></h2>
14     <ol id="qunit-tests"></ol>
15   </body>
16
17 </html>
```

Tests and modules

- Unit tests are defined using the `test` function
- Tests can be structured in modules (`module` function)
- A certain environment can be defined to be used for each test
- `setup` and `teardown` functions possible

Tests and modules

```
1  module( "Module identifier", {
2       'setup': function() {...},
3       'teardown': function() {...},
4       ...
5  });
6
7  test( "some unit test", function() {
8      // Use assertions here to ensure correct plugin
           behaviour
9  });
```

Assertions

- Assertions in QUnit do not follow xUnit pattern
- Argument order: actual, expected, reason
- Unusual naming of functions
  - `ok` == `assertTrue`
  - `equal` - Compare scalar values, check for same references on objects
  - `deepEqual` == `assertEquals` (recursive)
  - `raises` - Check for thrown exception
  - ...

Asynchronous tests

- QUnit does support asynchronous tests (`start`, `stop`)
- Only use these to test really asynchronous tasks, like an event dispatcher
- Most asynchronous calls can be mocked to be tested synchronously

Mocks, Stubs and Spys

- Use Mocks, Stubs and Spys in your tests
- Synchronize asynchronous tasks
  - Faster tests
  - No dependencies on components
  - Full control over response data (`XMLHttpRequest`)
- Sinon.JS - A Mock, Stub and Spy framework

Sinon.JS

```
1  // Simple mocks
2  var mock = sinon.mock( XMLHttpRequest.prototype );
3  mock.expects( "open" ).once().withArgs( "GET", "some/
       url", true );
4  mock.expects( "send" ).once().withArgs( null );
```

Sinon.JS

```
1  // Fake XMLHttpResponse Server
2  this.server = sinon.fakeServer.create();
3  this.server.respondWith(
4      [200, { "Content-Type": "text/plain" }, "some␣
           response"]
5  );
6
7  var spy = sinon.spy();
8
9  // Send request with spy as callback
10
11 this.server.respond();
12
13 ok( spy.called );
14 ok( spy.calledWith( "some␣response" ) );
15
16 this.server.restore();
```