CHAPTER 12
Professional Tools

[TODO intro. Background material about testing. Section summaries.]

12.1 Enabling Debug Logging

Problem

You want to capture log output from your code in a flexible manner, instead of resorting to embarassing old-school `alert()` dialogs.

Solution

Use `Y.log()` to log messages, warnings, and errors to your browser console. By default, `Y.log()` is included in the core YUI object. To enable logging to the browser console, create a `YUI` instance and call the `log()` method:

Example 12-1. Basic Debug Logging

```
<!DOCTYPE html>
<title>Basic Debug Logging</title>

<script src="http://yui.yahooapis.com/3.3.0/build/simpleyui/simpleyui-min.js"></script>
<script>
var Y = YUI();

Y.log("This is an informational debug message.", "info", "example_app");
Y.log("This is a warning message.", "warn", "example_app");
Y.log("This is a critical error message.", "error", "example_app");
</script>
```

Loading Example 12-1 into a browser displays a blank page, but the page is generating log messages. By default, your browser's error console captures YUI log output. As an example, in Firefox with Firebug installed, the Firebug console displays:

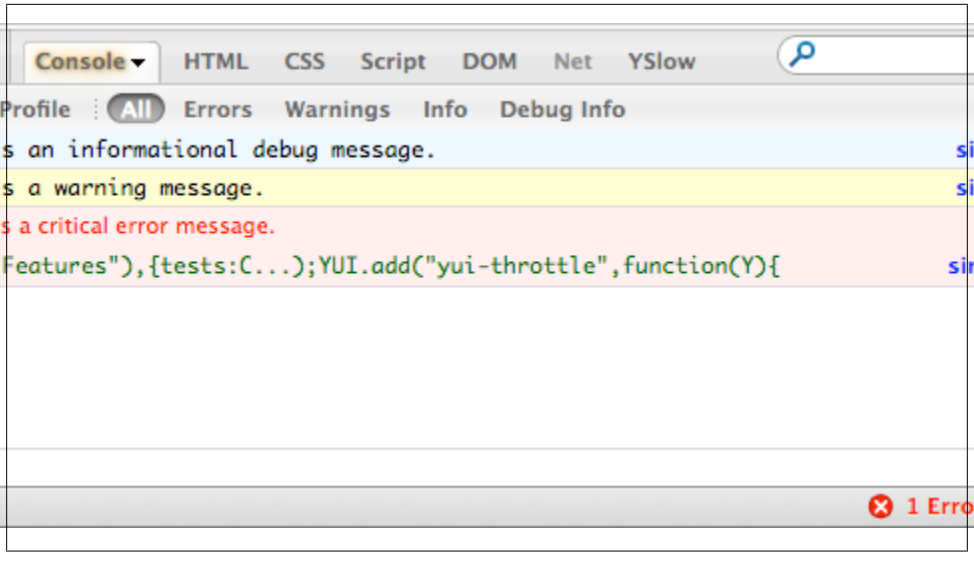

Figure 12-1. Sending Log Output to the Browser Error Console

Discussion

YUI's logging facilities are a sophisticated mechanism for inspecting the workings of both your own code and any other YUI library component. Calling the `log()` method itself is straightforward. The method has the signature `Y.log(msg, cat, src, silent)`, where:

- `msg` — Specifies the message to log.
- `cat` — (optional) Specifies a message category. Some browser error consoles provide a special display mode for the categories `"info"`, `"warn"`, and `"error"`, as does the YUI Console widget. However, you can provide any category name that makes sense for the object that is consuming the event.
- `src` — (optional) The source of the message. This is useful if you have multiple components that are firing log events and you need some way to distinguish the source.
- `silent` — (optional) If `true`, prevents the `yui:log` event from firing. This enables you to decide on-the-fly whether JavaScript components such as YUI Console are able to catch and display that particular message. Interestingly, the `silent` parameter does not affect whether the browser error console displays the log message — this is controlled by the YUI instance's `useBrowserConsole` configuration setting, as described below.

When `Y.log()` executes, it emits a custom `yui:log` event. The `yui:log` event bubbles up to an `EventTarget` named `Y.Global` that is available in every YUI instance. By default,

a `Y.Console` listens for `yui:log` events that occur within the local YUI instance. However, you can change this; more on how to do this later.

The YUI object provides several configuration settings that affect `Y.log()`'s behavior:

- `debug` — When set to `false`, disables `Y.log()` entirely. The default is `true`. Logging statements have the potential to slow an application down, so YUI makes it easy to disable logging in production. Alternatively, YUI provides a number of filtering capabilities that enable you to do things like turning on only error log messages, for a single module. If you need to leave logging enabled, you can be very fine-grained about how you do it.

  Also note that if you really want to disable all logging in production, the best approach is to use your build process to strip debug statements out of production code entirely, which reduces code paths and minimizes bytes over the wire. In fact, the YUI Builder tool that the YUI team uses is designed to do exactly this. For more information about how you can use YUI Builder with your own code, refer to Recipe 11.13.

- `filter` — Sets the default form for YUI modules to load: "min", "raw", or "debug". The default is "min". All native YUI modules ship in three forms:

  — *`module`*`-min.js` — The minified version, with variable names minimized and comments and whitespace stripped. This is the version suitable for production use. For more information about how to generate your own minified JavaScript, refer to Recipe 12.12.

  — *`module`*`.js` — The raw version, which includes full variable names, comments, and whitespace. This is the version to use if you want to hack on YUI directly, or learn about how YUI works by reading the source code.

    But wait, why would you ever care about reading YUI library source code? Well, as Trotsky might have said, you might not be interested in YUI's internals, but YUI's internals are interested in *you*. For example, the details of the internals start to matter if you make a simple usage error such as typoing a YUI method name. The production YUI modules yield the best performance, but minified variables and no whitespace make it very hard to debug your code. Using the "raw" filter enables you to track down exceptions and errors using the full human-readable YUI source code. Just don't forget to remove the filter when you're ready to ship.

  — *`module`*`-debug.js` — The debug version, which includes all `log()` statements that the YUI developer team have included for their own debugging purposes. This is the version to use if you want to learn about how YUI works by loading YUI components and watching log messages fly by. If you are trying to track down an error, stepping down from raw to debug might give you the extra information you need figure out where your problem lies. (Or it might just be confusing.)

- `filters` — Specifies the form to load for specific modules, overriding the overall `filter` setting for those specific modules. For example,

```
YUI({
    filter: "raw",
    filters: {
        autocomplete: "debug",
        io: "min"
    }
}).use(...);
```

would retrieve the debug version of AutoComplete, the minified version of IO, and the raw version of all other modules.

- `logExclude` — Disables logging only for the specified modules. For example, if you set:

```
YUI({
    filter: "debug",
    logExclude: {
        autocomplete: true,
        io: true
  }
}).use(...);
```

then logging for AutoComplete and IO would be disabled, but all other modules could emit log messages.

- `logInclude` — Enables logging only for the specified modules. For example, if you set:

```
YUI({
    filter: "debug",
    logInclude: {
        autocomplete: true,
        io: true
  }
}).use(...);
```

then only AutoComplete and IO could emit log messages, and logging would be disabled for all other modules.

- `useBrowserConsole` — When set to `true`, enables `Y.log()` to write to the browser error console. By default, this value is `true`. However, this value gets disabled when you invoke a YUI Console on the page. For more information, refer to Recipe 12.2.

[TODO show Y.log on the server side using Node.js. Show the beginnings of writing something that listens for yui:log on the server side.]

12.2 Rendering Log Output in the Page

Problem

Instead of relying on the browser error console, you'd rather monitor and manipulate log output directly in a web application.

Solution

The most convenient way to render log output is the YUI Console widget. Used together, `Y.Console` and `Y.log()` are a powerful combination for developing and debugging your code.

Example 12-2. Debug Logging to a YUI Console

```
<!DOCTYPE html>
<title>Debug Logging to a YUI Console</title>
<body class="yui3-skin-sam">
<div id="console"></div>

<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use("console", function (Y) {
var console = new Y.Console().render("#console");

Y.log("This is an informational debug message.", "info", "example_app");
Y.log("This is a warning message.", "warn", "example_app");
Y.log("This is a critical error message.", "error", "example_app");
});
</script>
</body>
```

The presence of a YUI Console causes YUI to automatically redirect log output to the YUI Console widget instead of the browser error console.

Console actually has its own local `useBrowserConsole` setting, defaulted to `false`, that passes through to the YUI object that it is listening to. Thus, creating a Console widget on a page automatically disables logging to the browser error console for log events in that YUI instance. To preserve browser error console logging, just set `useBrowserConsole` back to `true` in the Console constructor:

```
var console = new Y.Console({ useBrowserConsole: true }).render("#console");
```

Discussion

It's important to understand that the Console is actually a generic widget for displaying messages. By default, a Console widget is wired to listen for `yui:log` events, but you can easily reconfigure and extend the Console for wildly different purposes. Key customization points for the `Y.log()` -> `Y.Console` system include:

- changing the `logSource` that the Console is listening to
- filtering the messages the Console displays
- hooking into the Console's internal `entry` event, which fires whenever the Console transfers a message to the print loop buffer
- manipulating the Console and changing its display behavior
- changing the event type that the Console listens for from `yui:log` to a custom event

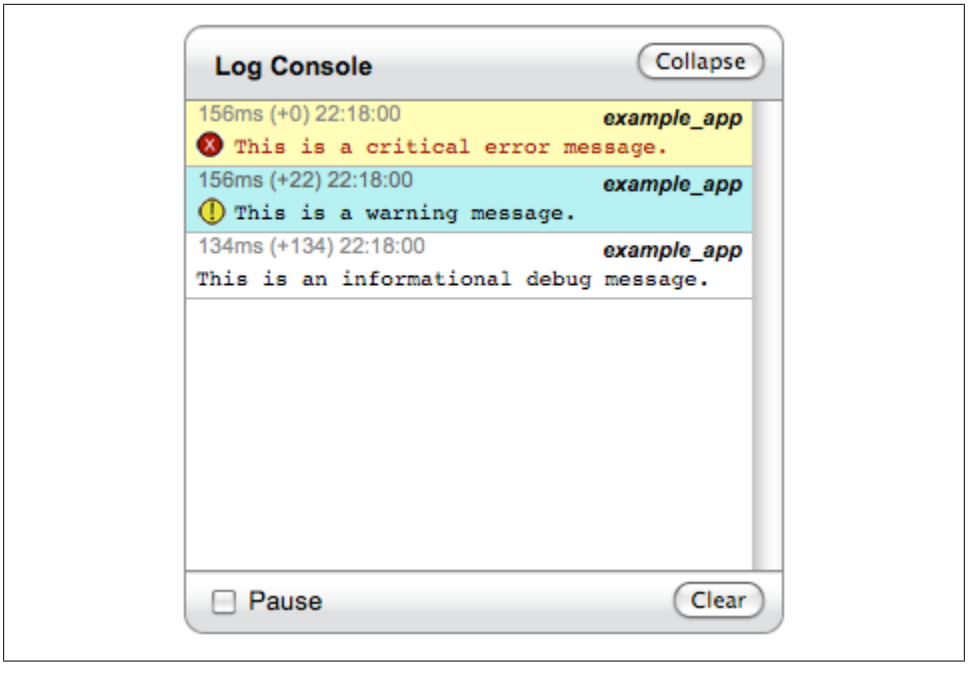

Figure 12-2. Rendering Log Output in a YUI Console

Within the Console, a message gets normalized to a simple `message` object with the properties `message`, `category`, `source`, `localTime`, `elapsedTime`, and `totalTime`. The first three properties correspond to the log message's original `msg`, `cat`, and `src` properties. The last three properties get added by the Console, and represent the time the message was received, the time since the previous message was received, and the time since the Console was instantiated. The `message` object gets attached to the `entry` event, which enables you to inspect the message in any `entry` handlers you write. The Console also uses the expected structure of this object to render individual messages using a template, which is another customization point.

For full documentation on how to customize YUI Console, refer to the Console API. Some of the interesting things you can do with Console include:

- *Creating a universal console*. By default, a Console receives log events only from the YUI instance it belongs to. However, since `yui:log` events bubble up to the `Y.Global EventTarget`, you can create a "universal console" that captures all log events simply by configuring the console's `logSource` to listen to `Y.Global`:

  Example 12-3. Universal Console

  ```
  <!DOCTYPE html>
  <title>Logging to a Universal Console</title>
  <body class="yui3-skin-sam">
  <div id="console"></div>
  ```

```
<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use('console', function (Y) {
    var console = new Y.Console({ logSource: Y.Global }).render("#console");
    Y.log("A log message that is local to the YUI console instance.");
});

YUI().use('node', function (Y) {
    Y.log("No log message has ever escaped... the universal YUI console!");
});
</script>
</body>
```

Setting `logSource` to `Y.Global` has two effects. First, both log messages appear in the YUI Console, as expected. Second, both log messages appear in the browser error console as well! Why is that? As described earlier, creating a Console disables `useBrowserConsole` for the YUI instance the Console is listening to. However, if you reconfigure the Console to listen to `Y.Global`, then Console's own YUI instance remains undisturbed and continues to pass log messages to the browser error console.

Try removing the `logSource` configuration in Example 12-3 to see the default behavior return. Once you do this, the log message in the first YUI instance will display in the YUI Console, but not the browser error console. For the log message in the second instance, it will be the other way around.

- *Enabling highly specific log filtering.* While you can control `Y.log()`'s behavior by setting configurables in the YUI object, you can also filter messages at the Console level. An obvious first place to start reducing noise is by setting the Console's `logLevel`:

  ```
  var console = new Y.Console({ logLevel: "warn" }).render("#console");
  ```

  The standard log categories have a hierarchy: `"error"` is greater than `"warn"` which is greater than `"info"`. Filtering on a `logLevel` of `"warn"` includes error and warning messages, but excludes informational messages. You can also apply a `logLevel` to the YUI object itself, which affects all consoles (including the browser error console). Log level filtering has no effect on custom categories, which lie outside the hierarchy.

  Taking filtering one step further, you can suppress messages with custom event logic. Example 12-4 is a Console designed to display error messages specifically related to uploads. It uses the previous `logLevel` technique to filter out `"info"` and `"warn"` messages, then hooks into the `entry` event to filter on the message content itself. If the message content does not contain the specified string, the `entry` handler prevents Console from printing the message.

Example 12-4. Filtering Log Messages on Console Entry

```
<!DOCTYPE html>
<title>Filtering Log Messages on Console Entry</title>
```

```
<body class="yui3-skin-sam">
<div id="console"></div>

<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use("console", function (Y) {
var console = new Y.Console({ logLevel: "error" }).render("#console");
console.on('entry', function (e) {
    if (! /upload/.test(e.message.message)) {
        e.preventDefault();
    }
});

Y.log("This is a critical error message.", "error", "example_app");
Y.log("File upload successful", "info", "example_app");
Y.log("File upload failed: server refused connection.", "error", "example_app");
});
</script>
</body>
```

As mentioned earlier, the `entry` event carries a `message` object representing the log message. This object in turn has a property named `message`.

Hooking into the `entry` event is powerful because it enables you to suppress events based on any criteria you like. Of course, this behavior is not just limited to filtering; you could use this event to perform some custom work, or relay the message to some other listener with `Y.fire()`. However, be careful not to do too much work in an `entry` handler — if you have large message volumes, executing code here can get expensive.

- *Manipulating Console's behavior and appearance.* You can:

  — Set Console's appearance using configurables:

```
var console = new Y.Console({
    style: "block",       // change the CSS: block, inline, or separate
    newestOnTop: false,   // change new entries to appear on the bottom
    width: "250px",
    height: "400px",
    collapsed: true,      // create the Console in a collapsed state
    paused: true,         // create the Console paused
}).render("#console");
```

  — Manipulate Console's state by calling methods:

```
console.expand();         // Expand the Console from a collapsed state
console.collapse();       // Collapse the Console
console.hide();           // Remove the Console from the UI entirely
console.show();           // Show the Console again

console.clearConsole();   // Empty the message pane and flush unprinted
                          // messages from the buffer
console.reset();          // Same as clearConsole(), plus unpauses the
                          // console and resets internal timers.
```

— Rewrite the HTML used to display messages by setting the `entryTemplate` configurable. This is a raw string of HTML markup interspersed with template variables, designed to be operated on by `Y.substitute()`. A fragment of the string might resemble:

```
...
<span class="{entry_cat_class}">{category}</span>
...
```

If CSS is not powerful enough to customize the message display, you can supply your own template to rewrite messages entirely. For examples of how to use `Y.substitute()` with templates, refer to Recipe 9.6 and Recipe 9.7.

— Enable end users to filter Console output by augmenting Console's behavior with the ConsoleFilter plugin:

```
YUI().use('console-filters', function (Y) {
    var console = new Y.Console();
    console.plug(Y.Plugin.ConsoleFilters);
    console.render();
});
```

This creates a convenient control pane with checkboxes that toggle messages on and off by category and source type.

- *Listening for a different custom event.* Consider a case where we have some kind of HTTP server written in JavaScript. Whenever the server handles an HTTP request, it fires off a custom `http:req` event. We can use a Console to monitor these events simply by changing the `logEvent`:

Example 12-5. Capturing Custom Events in the Console

```
<!DOCTYPE html>
<title>Capturing Custom Events in the Console</title>
<body class="yui3-skin-sam">
<div id="console"></div>

<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use("console", "event-custom", function (Y) {
var console = new Y.Console({ logEvent: "http:req" }).render("#console");

Y.fire("http:req", {"msg": "GET /foo HTTP/1.1 404 123", "cat": "error", "src": "404"});
Y.fire("http:req", {"msg": "GET /bar HTTP/1.1 200 3430", "cat": "info", "src": "200"});
});
</script>
</body>
```

Since we are far too lazy to write a complete working HTTP server here, Example 12-5 merely simulates incoming HTTP requests with a couple of hardcoded `Y.fire()` methods. The payload for each `http:req` event supplies an object with a `msg`, `cat`, and `src`, just like an ordinary log message. Since YUI Console is designed

to operate on messages with this structure, your custom events should at least include a `msg` property, and ideally a `cat` and `src`.

12.3 Writing Unit Tests

Problem

You want to create some assurance that a function works the way it was intended, and that it will continue to work properly even if you change its internal implementation later on.

Solution

Create unit tests for your function using the YUI Test unit testing framework. In example Example 12-6, we:

1. Provide a `trim()` function to test. This function could have been pulled in as a module, but in this case it's just defined locally.
2. Create a test case and populate it with an object literal containing tests. The tests are designed to evaluate a range of possible outcomes: leading whitespace, trailing whitespace, and trailing *and* leading whitespace. Each test case uses an assert method to validate results.
3. Create a Console to capture and display the test results, as we did earlier for simple log events in Recipe 12.2. Invoking YUI Test with Console together pulls in additional CSS resources that enable the Console to display passed test results in green and failed test results in red.
4. Add the test case to a `Y.Test.Runner` and invoke `run()` to execute the tests.

Example 12-6. Writing Unit Tests for a trim() Function

```
<!DOCTYPE html>
<title>Writing Unit Tests</title>
<body class="yui3-skin-sam">
<div id="console"></div>

<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use('console', 'test', function (Y) {

// Implementation is not quite right; regex is missing a /g
function trim(text){
    return text.replace(/^\s+|\s+$/,  "");
}

var testCase = new Y.Test.Case({
    name: "trim() Tests",

    "Leading white space should be removed": function(){
```

```
        var result = trim("     Hello world!");
        Y.Assert.areEqual("Hello world!", result);
    },

    "Trailing white space should be removed": function(){
        var result = trim("Hello world!     ");
        Y.Assert.areEqual("Hello world!", result);
    },

    "Leading and trailing white space should be removed": function(){
        var result = trim("     Hello world!     ");
        Y.Assert.areEqual("Hello world!", result);
    }
});

var console = new Y.Console({ newestOnTop : false }).render("#console");

Y.Test.Runner.add(testCase);
Y.Test.Runner.run();
});
</script>
</body>
```

Because the `trim()` function's regex is written incorrectly, the third test actually fails. This is actually great news — our tests have prevented us from shipping a poorly implemented feature! To fix the broken test, change the regex to `/^\s+|\s+$/g` and reload the page.

Discussion

In testing terms, a *unit* is an isolated chunk of code that you can test independently. In JavaScript and other languages that support objects, a unit corresponds to a method or function. To create effective JavaScript unit tests, your code should:

- Keep method bodies small. Each method should have a single purpose, or correspond to a single operation. This enables unit testing to make direct method calls and more easily compare state before and after.
- Avoid overusing true private variables (JavaScript variables hidden by closures). Creating truly private state and functions can be a powerful technique, but always keep in mind that private items are hard to unit test. At best, you can only test them indirectly.
- Never use `alert()` for debugging. As described in Recipe 12.1, professional JavaScript developers have far better techniques to debug output, techniques that do not require blocking for user input.

The idea behind traditional unit testing is that you are testing the implementation of a public interface. Your tests describe a set of input and output conditions that the method must satisfy, but the actual implementation of the method is irrelevant. This is called *black box testing*. Black box testing provides two key advantages. First, they enable you

to formally describe your input sets ahead of time. This not only helps you plan how to write your code, but also serves as a form of technical specification for others to read. Second, they provide confidence that you are free to fix bugs or even swap out the implementation without breaking other code that depends on that method. Of course, the trick is in developing enough high-quality tests to gain that level of assurance. For more information, refer to Recipe 12.4 and Recipe 12.9.

When YUI Test first debuted in 2008, the JavaScript testing landscape looked very different than it does today. Few developers paid much attention to JavaScript testing. The testing tools that did exist were faithful ports of frameworks designed for other languages, and were therefore poorly adapted to JavaScript. YUI Test evolved quickly into a much more natural JavaScript testing framework, complete with advanced features such as cross-browser event simulation and support for asynchronous testing. These days, developers take JavaScript testing much more seriously, and every major JavaScript library now ships with unit tests and a test runner.

More recently, the trend has been for JavaScript library test suites to evolve into standalone tools, and YUI Test is no exception. You are free to download and use YUI Test separately to test any JavaScript code you like, without any dependencies on the YUI libraries. This standalone package is referred to as "YUI Test Standalone." It includes a JavaScript library for testing, as well as self-contained documentation, examples, and some command-line utilities (discussed later in this book).

If you are working with the YUI Library sandbox, it is convenient to use YUI Test as we did in Example 12-6, where `YUI.use()` loads the test framework as a YUI 3 module. When operating as a YUI module, YUI Test is referred to as "YUI 3 Test." Either way, you get the same behavior, as YUI 3 Test is essentially just a YUI-friendly wrapper for the core JavaScript libraries provided by YUI Test Standalone.

[TODO Actually, YUI Test Standalone does not currently back YUI 3 Test, but this will almost certainly change by the time this book is published. This whole section is written assuming this change will happen.]

One feature YUI Test provides is friendly test names, similar to "Behavior Driven Development" test frameworks such as Vows. Many old school test frameworks rely on a simple naming convention to identify test functions, such as requiring any test function name to start with the string `"test"`. This results in functions like `testLeadingWhitespaceRemoved()`.

The YUI Test Runner supports this traditional convention. However, it also supports a second, more flexible convention:

```
"Leading white space should be removed": function() {
   ...
}
```

If you bind a function to a string property that contains a space, the YUI Test Runner identifies this as a test function as well. These friendly test names are useful because they encourage you to think of your tests as behaviors that need to be satisfied, and

enable you to display and discuss test results using natural language. Just imagine — test results so easy to read, even a product manager can understand them! (Maybe this isn't such a great feature after all.)

Older versions of YUI Test supported friendly test names by requiring the sentence to include the word "should." It turned out that this convention was suboptimal, because as it happens, some people around the world speak and think in languages other than English.

[TODO not quite right. YUI 3 Standalone does this, but the YUI 3 Test module requires "should" to be in the method name. This should be fixed by 2012.]

In addition to following a naming convention, a test function must contain one or more *assert methods* to run during testing. An assert method tests whether a condition is valid. For example, the popular `areEqual` assert method compares whether two values are equal, with an optional failure message as the third argument:

```
var name = "Pat";
Y.Assert.areEqual(name, "Pat", "The name should be Pat");
```

If all of the conditions within a test function are satisfied, the test passes. If any condition is invalid, the assert method throws an error, which skips all remaining assertions and causes the test to fail. If a test function executes zero assert methods during testing, the test function is an empty test, which YUI Test also treats as a failure. This is an intentional safeguard — if you have a buggy test function that is not actually executing any assertions, YUI Test will never fool you into thinking the test is passing.

[TODO also not quite right. YUI 3 Standalone does this, but YUI 3 Test treats empty tests as passing. This should be fixed by 2012.]

YUI provides a full spectrum of assert methods, including:

- Equality and sameness assertions. In addition to `areEqual()` and `areNotEqual()`, YUI also provides `areSame()` and `areNotSame()`, which test equality without doing any type coercion by using the `===` and `!==` operators. For example:

  ```
  Y.Assert.areSame(2, "2", "This assertion fails because a number is not a string.");
  Y.Assert.areEqual(2, "2", "You will never see this failure msg. Thanks, type coercion!");
  ```

- Data type assertions, including `isArray()`, `isBoolean()`, `isFunction()`, `isNumber()`, `isObject()` and `isString()`. There are also two generic type assertions, `isTypeOf()` and `isInstanceOf()`, which rely on the `typeof` and `instanceof` operators respectively. These last two operators in JavaScript are quirky, so be careful how you use them.
- Special value assertions, including `isTrue()`, `isNaN()`, `isNull()`, `isUndefined()`, and their opposites. These assertions do not perform any type coercion.
- Specialized assertion classes such as `DateAssert` and `ArrayAssert`, which provide methods such as `DateAssert.isTimeEqual()` and `ArrayAssert.containsItems()`.

[TODO Not yet in YUI 3 Test, but call out for special attention the newer YUITest.ObjectAssert methods: inheritsKey, inheritsKeys, ownsOrInheritsKey, ownsOrInheritsKeys. hasKey() included prototype properties, then changed to not include prototype properties, ... So now we have inheritsKeys (passes if on the prototype) and ownsOrInheritsKeys (passes if owns or on the prototype)]

For a complete list of available assert methods, refer to the API documentation.

There is no hard rule about the number of assertions that comprise a good test function. The goal is simply to verify a method's contract, to test each expected output for each given input. That said, here are some guidelines for using assertions and writing test functions:

- Decouple unrelated assertions into different test functions. As an example, we could have written Example 12-6 with a test function that evaluated both leading whitespace and trailing whitespace. However, this approach would have tested two unrelated input-output sets in the same function. If the first assertion fails, the second assertion is masked and never gets tested. The better approach is to test both input-output sets separately, so that they don't interfere with each other.
- Provide well-written failure messages with each assertion. In YUI Test, every assert method takes an optional failure message as the last argument. Avoid writing failure messages that tell you *what happened*:

  ```
  Y.Assert.areEqual("Hello world!", result, "The result wasn't 'Hello world'");
  ```

  This is a bad failure message, because you already know that the test failed. A good failure message tells you *what was expected*:

  ```
  Y.Assert.areEqual("Hello world!", result, "Leading white space should be stripped.");
  ```

  With this approach, a list of failures become a list of unfulfilled requirements for you to evaluate. This way of thinking about test failures dovetails nicely with YUI Test's friendly test names feature.
- Make sure that individual test functions run quickly. Most test functions should simply contain one or more assert methods with just enough machinery to run those assertions. In particular, test functions should avoid dragging in real dependencies or performing expensive operations.

  The reason for this is that testing is only effective when done on a regular basis, so you must be able to run your tests early and often. Ideally, you should run tests after making any substantial change, and you should definitely run them before committing changes to source control. If your tests are slow, you won't run them, which leads to tests that become less and less relevant to your actual code, which eventually leads to the total decay of the test suite.
- When necessary, create mock or fake dependencies. Unit testing is about testing small units of code in isolation. This means that you must have total control over object data that serve as inputs to your functions under test. Using mock objects

rather than real dependencies also has the side effect of speeding up your unit tests. For more information about mocking objects, refer to Recipe 12.6. There is a place for using real dependencies — this is called *integration testing*, and is discussed in Recipe 12.9.

12.4 Organizing Unit Tests into Suites

Problem

You've discovered that you have non-trivial number of tests to support. You want to group your tests into suites to support easier organization of test code and more meaningful test reports.

Solution

Create test cases to represent individual methods, and then group your test cases into a test suite to represent an entire object. In example Example 12-7, we:

1. Pull in the YUI JSON module as our code to test, along with the YUI 3 Test module and the Console module.
2. Create two test cases and populate them with object literals containing tests. The tests are designed to evaluate a range of possible outcomes: parsing a JSON object into JavaScript object, parsing a JSON array into a JavaScript array, and the reverse operations (serializing JavaScript to JSON). Each test case uses assert methods to validate results.
3. Create a Console to capture and display the test results.
4. Create a test suite with a suitable name and add the test cases to the test suite.
5. Add the test suite to a `Y.Test.Runner` and invoke `run()` to execute the tests.

Example 12-7. Grouping Tests into a Test Suite

```
<!DOCTYPE html>
<title>Grouping Test Cases into a Test Suite</title>
<body class="yui3-skin-sam">
<div id="console"></div>

<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use("json", "console", "test", function (Y) {

var parseTests = new Y.Test.Case({
    name: "parse() Tests",

    "JSON object should be parsed to JS object" : function(){
        var result = Y.JSON.parse("{\"value\":\"YUI3\"}");

        Y.Assert.isObject(result);
        Y.Assert.areEqual("YUI3", result.value);
```

```
    },

    "JSON array should be parsed to JS array" : function(){
        var result = Y.JSON.parse("[100, 200, 300, 400, 500]");

        Y.Assert.areEqual(5, result.length);
        Y.Assert.areEqual(100, result[0]);
        Y.Assert.areEqual(200, result[1]);
        Y.Assert.areEqual(300, result[2]);
        Y.Assert.areEqual(400, result[3]);
        Y.Assert.areEqual(500, result[4]);
    }
});

var stringifyTests = new Y.Test.Case({
    name: "stringify() Tests",

    "JS object should be serialized to JSON object": function(){
        var object = { value: "YUI3" };
        var result = Y.JSON.stringify(object);

        Y.Assert.isString(result);
        Y.Assert.areEqual("{\"value\":\"YUI3\"}", result);
    },

    "JS array should be serialized to JSON array": function(){
        var array = [1, 2, 3, 4, 5];
        var result = Y.JSON.stringify(array);

        Y.Assert.isString(result);
        Y.Assert.areEqual("[1,2,3,4,5]", result);
    }
});

var console = new Y.Console({ newestOnTop : false }).render("#console");

var testSuite = new Y.Test.Suite("Y.JSON Tests");
testSuite.add(parseTests);
testSuite.add(stringifyTests);

Y.Test.Runner.add(testSuite);
Y.Test.Runner.run();
});
</script>
</body>
```

Discussion

Like other test frameworks, YUI Test supports a hierarchy of test cases and test suites. Each test suite can contain test cases and other test suites. However, only test cases can contain actual tests (method names that contain a space or that begin with the word "test").

So how many tests should you write, and how should you organize your tests? The best way to organize your test hierarchy is to follow a simple pattern:

1. For each object you need to test, create one test suite.
2. For each method in the object to test, create one test case and add it to the object's test suite.
3. For each test case, create one test function for each input-output set in the method under test.

In this way, your test hierarchy mirrors the code you're testing. You can look at your code or your specification and figure out where you are missing test suites, and within those suites, where you are missing test cases.

The tricky part is the third step. How do you know how many test functions are enough? If you have too few tests, you're not actually verifying that your method works the way you think it does. But there is also such a thing as too many tests. After all, we can't *exhaustively* test every possible combination of inputs — for most objects, that would lead to a test suite so mindbogglingly large that you probably would need to learn special mathematical notation just to write the number of tests down. Even if we don't take things to absurd extremes, each line of test code you write is code that could itself be incorrect, code that you have to support. So somewhere between the number '0' and the unfathomably large, there is some number of tests that is "good enough." Presumably we can look at our program, reason intelligently about the number of tests that would actually illuminate something useful about the code, and proceed to write those tests.

One approach you can use to find the minimum number of tests is to use *structured basis testing*. In his book [Code Complete], Steve McConnell describes structured basis testing as the idea that you must test each statement in a program at least once. Thus, to determine the minimum number of test cases for a function:

1. Start with the number 1. This represents the straight path through the function.
2. For each of the following tokens, increment by 1: `if`, `while`, `for`, `&&`, and `||`.
3. Add 1 for each `case` in a `switch` statement.

For example, if you have a function that contains a for loop and an if statement, structured basis testing indicates that you need a minimum of three tests:

- One test for the straight path through the function (all boolean conditions are true).
- One test for the case where the initial `for` condition is false (the for loop is never executed).
- One test for the case where the `if` statement is false.

Of course, this is just a minimum, and you might very well need to add more. But this at least provides a starting point.

Let's look at our previous examples in turn. In Example 12-6, our `trim()` function is just a one-line regex with no flow control statements to be found. This means we need a *minimum* of one test. In reality, we almost certainly need more than that, as the regular expression is a mini-language all by itself. Given what we know of the regex, the three tests we actually have seem reasonable. If the `trim()` function contained more branches and loops, structured basis testing would indicate which additional tests we would need to write.

Let's turn our attention to Example 12-7, specifically the `stringify()` tests. Even before doing any analysis, it should be intuitively obvious that we need more tests — the test case only tests arrays and objects, and we know for a fact that JSON supports more syntax than that. Looking at the YUI JSON source code and tallying paths in the main `_stringify()` method alone, it is clear we are off by over an order of magnitude. If we really want to test the YUI JSON module adequately, we have our work cut out for us.

12.5 Testing Event Handlers by Simulating Events

Problem

You've written a click handler function that needs testing. Despite your lightning-quick ninja reflexes, it is impractical to test this code by always manually clicking the button yourself.

Solution

Use YUI's event simulation utility to test event handlers. Using event simulation, you can exercise the DOM in an automated fashion. In example Example 12-8, we:

1. Pull in the Node Event Simulation module, along with the YUI 3 Test module and the Console module.
2. Provide a controller with a click handler to test. This controller could have been pulled in as a module, but in this case it's just defined locally.
3. Create a test case and populate it with an object literal containing a setup function, a test, and a teardown function. The test simulates clicking the button and then evaluates the results. The setup and teardown functions do the necessary work of attaching and detaching the click handler.
4. Create a Console to capture and display the test results.
5. Add the test case to a `Y.Test.Runner` and invoke `run()` to execute the tests.

Example 12-8. Testing a Click Handler with Event Simulation

```
<!DOCTYPE html>
<title>Testing a Click Handler with Event Simulation</title>
<body class="yui3-skin-sam">
<div id="button"></div>
<div id="console"></div>
```

```
<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use("node-event-simulate", "console", "test", function (Y) {

var controller = {
    handleClick: function(event){
        event.target.addClass("clicked");
    }
};

var testCase = new Y.Test.Case({
    name: "Test Click Handler",

    setUp: function(){
        Y.one("#button").on("click", controller.handleClick, controller);
    },

    tearDown: function(){
        Y.one("#button").detachAll();
    },

    "handleClick() should add the class 'clicked' to the button": function(){
        var button = Y.one("#button");
        button.simulate("click");

        Y.Assert.isTrue(button.hasClass("clicked"), "Button should have a class of 'clicked'");
    }
});

var console = new Y.Console({ newestOnTop : false }).render("#console");

Y.Test.Runner.add(testCase);
Y.Test.Runner.run();
});
</script>
</body>
```

Discussion

Since the typical JavaScript program is tied tightly to the context of a browser environment, this creates unique challenges for testing. Methods that interact with the DOM are difficult to unit test outside a real brower. Further complicating matters is that a substantial part of many JavaScript applications involves event handling for mouse clicks and other human user interactions.

The good news is that YUI enables you to unit test DOM event handlers using event simulation. However, as your event testing grows more sophisticated, this starts crossing over from unit testing into *functional testing*. Functional testing tests the user's experience with the product rather than input-output sets for code. If you find yourself needing to test that the user interface responds in a specific way due to user interaction, then you really want to write some functional tests rather than unit tests.

If you are unsure whether something should be a unit test or a functional test, consider whether the test can be written before the code to test actually exists. Unit tests can be written ahead of time; functional tests generally cannot, because they are so closely tied to the particulars of the user interface.

While you can use YUI Test with YUI event simulation to write some basic functional tests, at some point you should consider driving your application with a dedicated functional testing tool. YUI Test includes command line utilities designed to integrate with Selenium, a popular open source functional testing tool. For more information about how this works, refer to Recipe 12.9.

Returning to our concrete example, when Example 12-8 in a browser, you might be wondering, "Hey, where's the button?" All the browser displays is a white page with a Console instance displaying test results, similar to previous examples. The "button" is just an empty `<div>`, impossible for a human user to click on. And yet... the test passes just fine. The test code "clicks" the button, the click handler adds a class to the button, and the assert method verifies that the class appeared as expected.

Of course, when you design your test pages, there's nothing stopping you from creating visible buttons. Your test buttons could even have attractive gradients and rounded corners. But as far as an automated test system is concerned, adding that extra markup and CSS is irrelevant fluff. Our event simulation code directly simulates a click in the DOM. The actual button aesthetics (or lack thereof) doesn't matter.

In Example 12-8, we use both a `setUp()` and a `tearDown()` method. For each test function in a test case, the YUI Test Runner runs the `setUp()` method before each test and runs the `tearDown()` method after each test, regardless whether the test passed for failed. These methods are perfect for creating and destroying necessary data objects or attaching and detaching event listeners.

You can also configure `setUp()` and `tearDown()` methods for test suites. The YUI Test Runner calls the test suite's `setUp()` method before executing the `setUp()` method of the first test in the first test case. Likewise, the YUI Test Runner calls the test suite's `tearDown()` method after all tests in all child test cases and test suites have executed, including the last test's `tearDown()` method. Test suite `setUp()` and `tearDown()` methods are useful for creating global resources that are meant to be shared by all tests in a suite.

12.6 Mocking Objects

Problem

You have an application that has several components that are expensive to load and render. To provide a smoother user experience, you have cleverly taken advantage of an asynchronous function queuing mechanism to render your UI in chunks. Unfortunately, this makes your master `renderUI()` function hard to unit test, because `ren`

derUI()'s job is to execute a queue of other rendering functions that inherently take a long time to run.

Solution

Use YUI Test's mock object facilities to create a fake asychronous queue. In your unit test, you don't actually care that the "queue" `renderUI()` doesn't contain any real rendering functions. You just care about walking through `renderUI()`'s code paths.

In example Example 12-9, we:

1. Provide a `renderUI()` function to test. This function could have been pulled in as a module, but in this case it's just defined locally.
2. Create a test case and populate it with an object literal containing a test function.
3. Populate the test function with a mock object representing the asynchronous queue of functions.
4. Define the methods on the mock object that we expect to call in this test, and the arguments that we expect to supply for each method.
5. Call `renderUI()` and verify that the methods on the mock object that we expected to get called actually did get called.
6. Create a Console to capture and display the test results.
7. Add the test case to a `Y.Test.Runner` and invoke `run()` to execute the tests.

Example 12-9. Testing with a Mock Object

```
<!DOCTYPE html>
<title>Testing with a Mock Object</title>
<body class="yui3-skin-sam">
<div id="console"></div>

<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use("console", "test", function (Y) {


function renderUI(renderQueue, renderFnToSkip) {
    if (typeof renderFnToSkip == "function" ) {
        renderQueue.remove(renderFnToSkip);
    }
    renderQueue.run();
}

var testCase = new Y.Test.Case({
    name: "UI Rendering Tests",

    "App should start rendering, but with no Twitter widget" : function() {
        var mockQueue = Y.Mock();
        var renderTweetWidget = function() {};

        Y.Mock.expect(mockQueue, {
```

```
            method: "remove",
            args: [renderTweetWidget]
        });

        Y.Mock.expect(mockQueue, {
            method: "run",
            args: []
        });

        renderUI(mockQueue, renderTweetWidget);

        Y.Mock.verify(mockQueue);
    }
});

var console = new Y.Console({ newestOnTop : false }).render("#console");

Y.Test.Runner.add(testCase);
Y.Test.Runner.run();
});
</script>
</body>
```

Discussion

Mock objects eliminate test dependencies on other objects. If your code depends on an object that is simple and fast, it is acceptable to create that object directly, perhaps in the `setUp()` method of your test case or test suite, perhaps in the test function itself. However, if your code depends on an object that:

- relies on a network connection
- performs some sort of expensive operation
- returns unpredictable results

then you should use a mock object. Mock objects ensure that despite complex, unpredictable, or slow dependencies, you maintain rigorous control over all test inputs, and your tests continue to be reproducible and run quickly.

The classic case for a JavaScript mock object is for simulating XHR or some other kind of call over the network. Network operations are a great thing to remove by mocking, since networks are slow *and* unpredictable. Example 12-9 takes a slightly different tack, using a mock object to simulate an expensive series of rendering operations. To make the test code a little more interesting, the `renderUI()` function has an additional feature; it allows the caller to pass in a second, optional argument representing a component that should be removed from the queue.

So how does mocking work? Creating a mock object might seem mysterious at first. If your code really depends on making an expensive call over the network... don't you actually need to make that call? When unit testing, the answer is no. Keep in mind that in a unit test, all you're really trying to do is exercise code paths through the function

under test. Fake input data and fake responses are fine, as long as your function responds to the fake data the way you expected.

Thus, when you use use a mock object, you don't care at all about its internals. What you *do* care about are the methods that get called on the mock object, including inputs and possibly any return values. You can think of `Y.Mock.expect()` as a kind of assertion about how you expect your code to exercise the object. Instead of asserting, "This variable should have the value 3," or "This variable should be a number," you are asserting, "This object should have this method called, with this argument list, and return this value."

To verify these expectations, you pass the mock object as input into the function you want to test, then call `Y.Mock.verify()` on the mock object. Calling `verify()` inspects the mock object to make sure that its methods were called correctly. If your test failed to call an expected method, passed in an incorrect value, or generated an incorrect return value, the test fails.

In our example, we called `renderUI()` with two arguments: the mock queue object to "run", and a (fake) function to "remove" from the queue. Looking at the implementation of `renderUI()`, we expect our test should end up calling two methods on the mock object: `remove(renderFnToSkip)` and `run()`. We therefore define two expectations:

1. the `remove()` method will be called with a single function argument named `renderTweetWidget`
2. the `run()` method will be called with an empty argument list

Since our code doesn't use any return values, we don't bother to check for them in the expectation. Your expectation object may include:

- `name` — The string name of the method you expect to be called (**required**).
- `args` — An array representing the list of arguments you expect to be supplied to the method (**required**).

  If you only care about the type of the argument and not its value, you can provide a special `Y.Mock.Value` in place of an actual value. Supported types include `Y.Mock.Value.String`, `Y.Mock.Value.Number`, `Y.Mock.Value.Boolean`, `Y.Mock.Value.Object`, `Y.Mock.Value.Function`, and even `Y.Mock.Value.Any`, if you don't care about the type of the value at all. In Example 12-9, our code doesn't actually depend on the value of the argument passed into `remove()`, just the type. So we could rewrite the expectation as:

  ```
  Y.Mock.expect(mockQueue, {
      method: "remove",
      args: [Y.Mock.Value.Function]
  });
  ```

  The reason the test uses a function specifically named `renderTweetWidget` is simply to make the example a little more concrete. We're not just choosing to remove any

old function from the render queue, we're choosing to remove a widget that displays tweets.

- `returns` — The value you expect the method to return.
- `error` — An error you expect the method to throw.
- `callCount` — The number of times you expect the method to be called with the given arguments.

If any of these defined expectations are not satisfied, calling `verify()` throws an assert error that fails the function, exactly like an ordinary assert method. In case you were wondering, you are of course free to use ordinary assert methods right alongside `verify()`.

Note that in Example 12-9, there is only a single test. An obvious second test is missing — the case where we *don't* pass in a function to skip. In that case, we only expect the `run()` method to get called. What would testing that second code path like? Something like this:

Example 12-10. Adding a Second Mock Object Test

```
"The entire app should start rendering" : function() {
    var mockQueue = Y.Mock();

    Y.Mock.expect(mockQueue, {
        method: "run",
        args: []
    });

    renderUI(mockQueue, null);

    Y.Mock.verify(mockQueue);
}
```

We expect that passing `null` in as `renderUI()`'s second argument will cause `renderUI()` to only call `run()` with no arguments on the queue object. The test in Example 12-10 verifies that this is exactly what happens.

12.7 Testing Asynchronously using wait

Problem

You need to pause your tests and resume them later, in order to evaluate some state that will be present in the future.

Solution

Use YUI Test Case's ability to pause and resume a running test with `wait()` and `resume()`. In example Example 12-11, we:

1. Provide a `changeToRed()` function to test. This function could have been pulled in as a module, but in this case it's just defined locally.
2. Create a test case and populate it with an object literal containing a test function. This test case is special in that it is designed to suspend until after the `changeToRed()` function completes its work.
3. Set an event handler that listens for the `example:red` event. When the event handler triggers, it resumes the suspended test case and checks an assertion about the `<body>`'s current background color.
4. Call `changeToRed()`, immediately followed by `wait()`. The `wait()` method immediately suspends the test case until something calls `resume()`.
5. Create a Console to capture and display the test results.
6. Add the test case to a `Y.Test.Runner` and invoke `run()` to execute the tests.

Example 12-11. Testing Asynchronously with wait()

```
<!DOCTYPE html>
<title>Testing Asynchronously with wait()</title>
<body class="yui3-skin-sam">
<div id="console"></div>

<script src="http://yui.yahooapis.com/3.3.0/build/yui/yui-min.js"></script>
<script>
YUI().use("console", "test", function (Y) {

function changeToRed(timeout) {
    setTimeout(function(){
        Y.one("body").setStyle("backgroundColor", "rgb(255, 0, 0)");
        Y.fire("example:red");
    }, timeout)
}

var asyncTest = new Y.Test.Case({
    name: "Asynchronous Transition Test with Events",

    "After 2 seconds, body should change to red" : function(){
        var resume = this.resume;

        Y.on("example:red", function() {
            resume(function(){
                var bgColor = Y.one("body").getStyle("backgroundColor");
                Y.Assert.areEqual("rgb(255, 0, 0)", bgColor);
            });
        });

        changeToRed(2000);
        this.wait();
    }
});

var console = new Y.Console({ newestOnTop : false }).render("#console");
```

```
Y.Test.Runner.add(asyncTest);
Y.Test.Runner.run();
});
</script>
</body>
```

Discussion

Because `changeToRed()` doesn't complete its work until some timeout has occurred, we can't verify its results immediately. What we need to do is suspend the test case until conditions are right. There are two basic approaches:

1. Suspend the test case until some event occurs, then resume the test case and run some assertions.
2. Suspend the test case for some fixed period of time, then resume the test case and run some assertions, as illustrated in Example 12-12.

Since `changeToRed()` conveniently fires an event as soon as its work is complete, the first option seems like the way to go. The listener is a bit complicated, so let's break down its structure step by step:

1. Using `Y.on()`, we set a listener for the `example:red` event.
2. When the listener detects an `example:red` event, it triggers the event handler. The event handler does one thing: calls the test case's `resume()` method.
3. The `resume()` method accepts a single argument, and does two things:
   - Wakes up the test case from its suspended state.
   - Executes the function that was passed into `resume()`. This function should be responsible for running any assertions

Within the event handler function, `this` is now set to refer to the `Y` instance, not the test case instance. Since the test case's `this.resume()` is no longer in scope, the event handler can't refer to it directly. Fortunately, we can enable the event handler to call `this.resume()` simply by saving `this.resume` to a variable. The event handler function closes over this variable, and everything works fine.

After setting the listener, we call the `changeToRed()` function, followed immediately by `this.wait()`, which suspends the test case. The test case remains suspended until the `example:red` event fires 2000 milliseconds later, which triggers the event handler, which calls `resume()` to resume the test case.

You can see this in action if you load Example 12-11 in a browser. The Console runs some test startup log messages, halting at the message:

```
Test case "Asynchronous Transition Test with Events" started
```

There the Console waits until 2000 milliseconds pass, the background changes to red, and the test case resumes from suspension.

If you are testing code where there is no convenient event to listen for, you can still run an asynchronous test by using `wait()` all by itself, without `resume()`. Previously, we called `wait()` with no arguments, which causes the test to suspend until some other function specifically calls `resume()`. However, similar to `setTimeout()`, the `wait()` method can actually take two arguments: a time period for which to suspend the test case and a function to execute when the time period elapses. As with the function you would pass into `resume()`, the function you pass into `wait()` should contain assertions to run.

As an example, if `changeToRed()` did *not* happen to fire an event on completion, we could cope with that by rewriting our asynchronous test case to look like this:

Example 12-12. Testing Asynchronously with No Event to Listen For

```
var asyncTest = new Y.Test.Case({
    name: "Asynchronous Transition Test",

    "After 2 seconds, body should flip to red" : function(){
        changeToRed(2000);

        this.wait(function(){
            var bgColor = Y.one("body").getStyle("backgroundColor");
            Y.Assert.areEqual("rgb(255, 0, 0)", bgColor);
        }, 2100);
    }
});
```

Now instead of listening for an event, we are simply waiting a fixed period of time for the body to change to red. After 2100 milliseconds, the test case resumes, and we execute a function that contains an assertion. This assertion executes in the context of the test case object, so it still has access to all of the same data as the test that called `wait()`, including any properties and methods on the test case object itself. Note that if we had guessed wrong for some reason and waited only 1800 milliseconds, the background color would not yet be red and so the test would fail.

12.8 Collecting and Posting Test Results

Problem

Rather than just viewing test results in a Console, you want to collect raw test data and store it somewhere for later analysis.

Solution

Use the YUI Test's Test Reporter to post test results to a remote server. Our example reuses the code from Example 12-7, replacing the last two lines

```
    Y.Test.Runner.add(testSuite);
    Y.Test.Runner.run();
```

with the code in Example 12-13. Instead of simply running tests, we are now listening for the event that signals the end of testing. The event handler collects the results in JSON format and posts the data to some kind of URI endpoint.

Example 12-13. Collecting and Posting Test Results

```
var testRunner = Y.Test.Runner.add(testSuite);

testRunner.subscribe(Y.Test.Runner.COMPLETE_EVENT, function() {
    var results = testRunner.getResults();
    var reporter = new Y.Test.Reporter("http://localhost/report.php");
    reporter.report(results);
});

testRunner.run();
```

Discussion

In previous examples in this chapter, the YUI Console was our mechanism for storing test results. As soon as you close the browser window, those test results disappear. This is okay for one-off testing, but if you are running tests on a regular basis, you will almost certainly want to store the result data somewhere.

Since the test data is trapped on the web page, what's the best way to get at it? The good news is that you don't have to roll your own XHR solution or worse, scrape the data off the test page. YUI Test provides a handy utility in the YUI Test Reporter. You just need to get a results object from the Test Runner, invoke a Test Reporter instance, and call the `report()` method. When you call `report()`, Test Reporter creates a form and POSTs the data to the endpoint you specified with these fields:

- `results` — The results object serialized to a string.
- `useragent` — Your browser's user agent string.
- `timestamp` — The date and time the report was sent.
- *`custom_field`* — An extra custom field you added by previously calling `addField(`*`custom_field, value`*`)` on the Test Reporter instance. You can add multiple additional fields, but you cannot override the three built-in fields.

The form submission does not cause your test page to navigate away. Note that Test Reporter does not receive a response back from the endpoint, so there isn't a simple way to ensure that your POST succeeded.

You obtain a raw test results object by calling `getResults()` on the Test Runner. When you instantiate the Test Reporter, you can specify the format the test data should be in by passing one of these constants in as the second constructor argument:

- `Y.Test.Format.XML` — A YUI-specific representation of the test results in XML. This is the default.

- `Y.Test.Format.JSON` — A YUI-specific representation of the test results in JSON.
- `Y.Test.Format.JUnitXML` — A representation of the test results in JUnit XML, a format that many testing tools heroically manage to support despite the fact that nobody is quite sure what the specification is.
- `Y.Test.Format.TAP` — A representation of the test results in TAP (Test Anything Protocol) format.

[TODO: setting Y.Test.Format.JSON does not seem work in getResults(), but does work when creating a test reporter.]

If you call `getResults()` before tests have completed running, the method returns `null`. In other words, you can't do this:

```
testRunner.run();
var results = testRunner.getResults();
```

because when `getResults()` executes, there is no chance that the Test Runner has completed its work yet. That's why Example 12-13 relies on subscribing to the `Y.Test.Runner.COMPLETE_EVENT` event, which indicates that all testing is over. When that event fires, it is safe to extract test results and post them.

So where exactly should you post the report data? If you are running some kind of third-party test harness or continuous integration tool, there's a good chance that it already supports the JUnit XML or TAP formats. You can use YUI Test Reporter to POST data directly to some endpoint in the tool — or if your tool's API doesn't support POSTing report data directly, you can always write a small proxy script that accepts POST data and then turns around and feeds the tool properly.

Alternatively, you can write a custom report display screen. This is more work, but it enables you to slice and dice the data any way you please. It also affords you the opportunity to flex your server-side JS skills and implement an elegant solution in pure JavaScript. Or you can just bludgeon the problem into submission with PHP.

Example 12-14. Quick and Dirty PHP Test Report Script

```
<?php
if ($_SERVER['REQUEST_METHOD'] == "POST") {
    file_put_contents("/tmp/test_results.xml", $_POST['results']);
}
else {
    $results = simplexml_load_file("/tmp/test_results.xml");
    $tests = $results->xpath("//test");
?>

<!DOCTYPE html>
<title>Test Report for <?php echo $results['name']; ?></title>
<style>
    td { border: 1px #000 solid; padding: 2px }
    td.fail { background: #f33; }
</style>
<h1>
```

```
    Test Report for <?php echo $results['name']; ?>:
    <?php echo $results['total']; ?> total,
    <?php echo $results['failed']; ?> failed
</h1>
<table>
<tr>
    <th>Test</th>
    <th>Result</th>
    <th>Message</th>
</tr>
<?php
    foreach ($tests as $test) {
        $class = $test['result'] == 'fail' ? 'class="fail"' : '';
?>
<tr>
    <td><?php echo $test['name']; ?></td>
    <td <?php echo $class; ?>><?php echo $test['result']; ?></td>
    <td><?php echo $test['message']; ?></td>
</tr>
<?php } ?>
</table>
<?php } ?>
```

You can drop Example 12-14 into almost any webserver running PHP 5. If the script does not run properly, check your server's INI settings: `magic_quotes_gpc` must be off, and `open_basedirs` must permit scripts to write to `/tmp` (or wherever you want to write the test result data). To see the report script in action, modify Example 12-13 so that `Y.Test.Reporter` points to your PHP report script, open the HTML page in a browser to run the tests and post the data, and then open the PHP script in a browser to view the results.

This minimal, self-contained report script lacks certain desirable features such as reproducibility of results, security, and aesthetics. However, it does illustrate some basic principles:

1. You need a component that handles incoming POST requests.
2. That component must store the data persistently somewhere.
3. On a GET request, the app must fetch the requested report.
4. Finally, you must format the raw data for display.

Our report script ignores the nested test suite / test case structure in the data and just lists the tests. If you have a large, well-organized collection test suites, you will almost certainly want to design a more sophisticated visual representation.

12.9 Using Continuous Integration

Problem

You want to incorporate your unit tests into a fully automated system that continuously builds your code, runs your tests in multiple browsers, and reports any failures.

Solution

Use the YUI Selenium Driver to direct Selenium to ...

[TODO example of running the Selenium Driver]

Discussion

[discussion of automation / CI]

12.10 Testing Multiple Browsers Before Committing

Problem

Before committing your changes, you would like to quickly run a series of automated tests to verify that your recent changes work in multiple browsers.

Solution

[TODO: Example of running Yeti]

Discussion

[TODO: Diveintoyeti]

12.11 Testing on Mobile Devices

Problem

You want to test your code in mobile browsers, but you can't seem to get Node.js and Yeti to run on your iPhone.

Solution

[TODO show how to use Yeti with localtunnel]

Discussion

[TODO how to install localtunnel, other details of localtunnel + Yeti, other mobile testing tips]

12.12 Compressing Your Code

Problem

You want to make your code as small as possible so that it loads quickly over the network. You already plan to GZIP your code, but perhaps there's some way to actually shrink the code that GZIP is acting upon?

Solution

Use the YUI Compressor tool to safely remove whitespace and minimize local symbols where possible. The easiest way to do this is to use the online YUI Compressor:

1. Paste your code into the text box (or alternatively upload your JavaScript as a file or point YUI Compressor at your code online.
2. Click Compress. YUI Compressor converts your code into equivalent but smaller JavaScript. For example, the function

   ```
   function trim(text){
       return text.replace(/^\s+|\s+$/g,  "");
   }
   ```

   compresses to:

   ```
   function trim(a){return a.replace(/^\s+|\s+$/g,"")};
   ```

   which is 25% smaller.
3. GZIP the compressed results.

The online version of YUI Compressor is very easy to set up, but it is designed for one-off conversions. To integrate YUI Compressor into your build process, download the latest release of YUI Compressor so that you can run it on the command line:

[TODO real life command line example]

Discussion

When people speak of *compression* on the web, they're often referring to HTTP compression. In HTTP compression, the web server compresses source files with a scheme such as GZIP, and web clients retrieve these files using HTTP, uncompress them, and use them as ordinary resources. HTTP compression is an important tool in your toolkit, and performance scoring utilities such as YSlow take this into account.

Although GZIP only works well on text files such as HTML, CSS, and JavaScript, it is still possible to reduce binary image file sizes using other techniques. For more information, refer to the YSlow Smush.it utility.

There is, however, a second kind of compression that complements HTTP compression nicely. This approach is called *minification*. If you consider a typical hand-authored JavaScript file, it contains all sorts of extra characters that are useful for humans responsible for *authoring* the code, but that aren't useful for JavaScript engines responsible for *executing* the code. So as a first pass, you might imagine writing a post-processor that strips extra whitespace and comments, resulting in a smaller file that still has the same behavior in the browser. Although there's a bit of complexity here given that the JavaScript language has significant newlines, this kind of minification algorithm is pretty straightforward and does in fact save some bytes, on top of what HTTP compression saves. Success! But can we do better?

With a more sophisticated approach, it is possible to minify even more aggressively. JavaScript engines do not require function names, variable names, and other symbols have long human-readable names. Converting those symbols to single letters would provide even more savings; in fact. However, blindly converting every variable name is dangerous! Our algorithm has to account for which variables are hidden safely in a scope, and which are accessible outside the program. At this point, the algorithm can't just operate on JavaScript code as a giant string, but as a stream of tokens. In other words, we need an algorithm able to parse JavaScript itself.

The YUI Compressor is a Java command line tool that relies on Mozilla's Rhino JavaScript engine to tokenize the source JavaScript file. Once Rhino tokenizes the JavaScript source file, YUI Compressor analyzes the source and determines which symbol names are in a local scope and are therefore safe to minify. Variable renaming alone saves more bytes on average than simple whitespace stripping. YUI Compressor also runs a battery of other techniques to minify code, such as:

- concatenating large strings by safely removing the + operator (which might have the side effect of slightly speeding up your code execution)
- removing comments and whitespace
- safely replacing bracket notation with dot notation where safe (`foo["bar"]` becomes `foo.bar`)
- safely replacing quoted literal property names (`{"foo":"bar"}` becomes `{foo:"bar"}`)
- removing semicolons in places where JavaScript's automatic semicolon insertion would terminate the statement anyway
- using regular expressions to minify CSS

The result is minification that is efficient, but conservative and safe. Keep in mind that the minification process is slow enough that you cannot do it on the fly. Instead, you should incorporate it into your build process.

Some of these techniques are actually bad practice when authoring code, precisely because they make the code hard for humans to read. But here, this is okay — minified code is for JavaScript engines, not humans.

Finally, because YUI Compressor is sophisticated enough to actually parse JavaScript, it is able to include an optional "lint" feature that warns about bad code practices, similar to JSLint. Unlike JSLint, YUI Compressor limits its advice to bad practices that specifically inhibit compression, such as:

- using the `eval()` or `with()` statements (which have unpredictable access to local variables, defeating renaming)
- whether a variable is used but never defined (which creates a global variable that cannot be safely renamed)
- whether a variable is defined but never used
- whether a variable is defined multiple times

You can display this advice with a `-v` on the command line.

The good news is that many of the techniques that make JavaScript code work well with YUI Compressor are good programming practices in general. These practices include:

- Avoiding the `eval()` statement. There are already other well-known reasons to avoid using `eval()`, such as security problems and performance concerns. But `eval()` also interferes with code minification. The code executed in an `eval()` statement has access to local variables in the scope where `eval()` is called. Since there is no way to know if the `eval()` will actually use any of these local variables, YUI Compressor cannot safely rename any variables in this scope.
- Avoiding the `with()` statement. Like `eval()`, there are already reasons to avoid using `with()`. And like `eval()`, one more reason to avoid `with()` is that it interferes with code minification. Since `with()` confuses the distinction between variables and object properties, YUI Compressor cannot safely rename any variables in the scope where `with()` is called.
- Minimizing the use of global variables. This is good advice in any language. In the situation we're concerned about there, YUI Compressor cannot rename global variables for the obvious reason, that other code on the page might be using those variables.
- Using constants to represent repeated literal values. Again, this is good advice in general; if the string `"Great Underground Empire"` appears multiple times in your

code, storing this string in a variable makes it easier to update this value later on. As a bonus, this helps out YUI Compressor. YUI Compressor does not replace literal values, so defining common literal values as constants enables YUI to compress values that it couldn't compress before.

- Storing local references to objects and values. YUI Compressor can't rename global variables or multi-level object references. However, storing these references in local variables within a function enables YUI Compressor to freely rename them. For example, in Example 12-15, YUI Compressor cannot minify `MyApp.MyModule.MyClass`.

  Example 12-15. Cannot Minify Multilevel Object Reference

  ```
  function frobozz(){
      if (MyApp.MyModule.MyClass.hasSword()) {
          MyApp.MyModule.MyClass.killTrollWithSword();
      }
      else {
          MyApp.MyModule.MyClass.death();
      }
  }
  ```

  However, if you store this object reference in a local variable, `MyApp.MyModule.MyClass` need only appear once at the top of this function, and YUI Compressor can then proceed to compress all the instances of the variable `myClass`.

  Example 12-16. Can Minify Local Variable of Multilevel Object Reference

  ```
  function frobozz(){
      var myClass = MyApp.MyModule.MyClass;
      if (myClass.hasSword()) {
          myclass.killTrollWithSword();
      }
      else {
          myClass.death();
      }
  }
  ```

Finally, YUI Compressor supports special syntax that provides fine-grained control over how minification works.

First, there are situations where you need to preserve comments even in minified production code, such as copyright statements. In fact, you might be using code under a license that *requires* including the copyright statement. To preserve a multiline comment, include an exclamation point as the first character:

```
/*! Copyright 2012, Frobozz Inc. This is a very important license
 *  statement. Failure to include this license can result in...
```

Second, you can prevent individual local variables, function names, or function arguments from being renamed by using a *hint*. A hint is a string at the very beginning of

the function definition. The hint string should contain a comma separated list of *variable*:nomunge tokens, as in:

```
function zorkmid(frobozz, xyzzy) {
    "xyzzy:nomunge, grue:nomunge, plugh:nomunge";

    var grue = true;
    function plugh() {...}
    ...
}
```

This would permit YUI Compressor to rename `frobozz`, but preserve `xyzzy`, `grue`, and `plugh`.

12.13 Documenting Your Code

Problem

You want to make sure other people can figure out how to use your JavaScript API — without having to ask you directly.

Solution

Use YUI Doc to create documentation from comments in source code. Like YUI Test, YUI Doc is a standalone project. YUI Doc can generate documentation for non-YUI code. In fact, YUI Doc can generate documentation for languages other than JavaScript.

In Example 12-17, we've taken the method from Recipe 12.7 and enhanced it with some comments to generate API documentation. To show off the capabilities of YUI Doc a little better, we've reimagined how `changeToRed()` works. Instead of a standalone function, `changeToRed()` is now a method that belongs to a class `ColorChange` that in turn belongs to an `Example` module.

Example 12-17. Documenting a Method

```
/**
 * Provides toy classes and methods used to show how unit testing
 * works in YUI Test.
 * @module example
 * @submodule example-async
 * @for ColorChange
 */

/**
 * Provides simple timed style changes that you can use to learn how
 * to write asynchronous tests.
 * @class ColorChange
 */
Example.ColorChange = {

    /**
```

```
     * After the specified timeout, changes the body of the document
     * to "rgb(255,0,0)" and fires an "example:red" event. This is a toy
     * function used to illustrate asynchronous testing.
     * @method changeToRed
     * @param {int} timeout The number of ms to wait before changing
     *                      the body to red.
     */
    changeToRed: function(timeout) {
        setTimeout(function(){
            Y.one("body").setStyle("backgroundColor", "rgb(255, 0, 0)");

            /**
             * Indicates that the background has changed to red. This is
             * the event to listen for when running an asynchronous test
             * with wait() and resume().
             * @event example:red
             * @type Event:Custom
             */
            Y.fire("example:red");
        }, timeout)
    }
};
```

To convert this into documentation:

1. Download YUI Doc from *yuilibrary.com*.
2. Unpack the files into a directory, and follow the installation instructions in the *INSTALL* file.
3. Edit the *example.sh* UNIX shell script or the `example.bat` Windows batch file to match the paths on your system.

The main YUI Doc script is called *yuidoc.py*. You can always run *yuidoc.py* directly, but `example.sh` and `example.bat` are convenient because *yuidoc.py* requires a large number of commandline parameters to work properly. In practice, you should eventually drive *yuidoc.py* using a Makefile, an Ant script, or whatever else you are using to build your software.

4. Execute `example.sh` or `example.bat`.

YUI Doc then generates HTML files and raw parsed files, in the directories you specified in `example.sh` or `example.bat`.

Discussion

YUI Doc is a *documentation generator*, a tool that creates API documentation from specially formatted doc comments. The most famous tool of this type is probably Javadoc, but there are many others, including Doxygen, phpDocumentor, and JsDoc Toolkit.

Most documentation generators are designed to actually parse source code in a particular language or set of languages. This helps keeps the generated documentation in sync with the actual classes and method signatures in the API. Even if there are no doc comments, it's still possible to at least generate a friendly HTML "view" of the API.

YUI Doc is different. Unlike most documentation generators, YUI Doc does not actively parse JavaScript or any other language — it just parses YUI doc comments. The advantage of this approach is that YUI Doc is language-neutral; you can use it to document an API in nearly any language, not just JavaScript. The disadvantage is that you must explicitly declare every parameter in every method. Unlike other tools, YUI Doc cannot generate "skeleton" API documentation for components that you forget to document. If you prefer to use a JavaScript-specific documentation generator, try JsDoc Toolkit.

Similar to Javadoc, YUI Doc's syntax relies on multiline comment blocks that contain "tags" that start with an @, such as `@param` or `@returns`. A YUI Doc comment block:

- must contain one and only one of these primary tags: `@module`, `@class`, `@method`, `@property`, `@event`, or `@attribute`
- may contain a plain text description
- may contain one or more secondary tags such as `@param`, `@type`, and `@returns`

For more information about the available tags and how to use them, refer to the YUI Doc documentation.

YUI Doc is designed to represent a JavaScript library as an organized structure of modules and classes. A *module* in YUI Doc is an overarching piece of a larger library, somewhat analogous to a Java package. JavaScript does not have an intrinsic notion of modules, but they are an important way to organize your library, and in fact YUI Doc requires you to define at least one module. Each module contains one or more *classes*. JavaScript is a prototypical language and does not have a strong notion of classes, so in YUI Doc, a class is simply an object with methods. Within classes, you can also define properties, events, and attributes (object configuration values).

[TODO The YUI Docs say @attribute and @config are secondary tags, but I think they might actually be more like primary tags. At least, it looks like you can create a block that just has an @attribute...]

The core feature of all documentation generators is that they ensure the documentation's source exists right next to the corresponding source code. This treats API documentation more like source code, which in turn helps keep the documentation accurate. If you add a parameter to a method, you are more likely to remember to update the documentation if that documentation source sits just a line or two above the method signature.

Although documentation generators are the right tool for writing API documentation, crafting effective doc comments is surprisingly tricky. The problem is that you are serving two audiences: people who read the generated API documentation, and people

who read the source code. This means you need to be careful about cluttering your source with lots of "line noise" comments that make it hard to read your code. Doc comments must be informative, terse, and clean.

Perhaps the most common mistake people make is writing meaningless doc comments that simply restate the name of the method. For example, documentation like:

```
getColor(): Gets the color.
```

There is almost always *something* more useful to say than this. Focus on the "how" and the "why" rather than the "what." Is there an important pattern (or anti-pattern) when using this method or class? Edge cases to consider? Your doc comments should highlight the kinds of things that would be hard to figure out simply by knowing the method signature or by reading the source code. If all else fails, keep in mind that poor doc comments are worse than empty doc comments.

On the other extreme, avoid writing very verbose doc comments, sprinkled with lots of special tags and metadata. As with any type of comment, don't let doc comments overrun your source code and become clutter. One of the nice things about YUI Doc is that its syntax is small, limited to a useful set of tags that describe components of the API and how they relate to each other. Other documentation generators offer dozens of additional tags that control the formatting or that duplicate metadata that is already in your version control system. These tags seem like nifty features, but actually lead to worse documentation.

Because YUI Doc does not actually parse JavaScript, it enables you to write doc comments far away from the method, class, or other component being documented. Unfortunately, this obviates the key feature of documentation generators, which is improving maintainability by keeping documentation close to the source. Always try to keep YUI doc comments close to the corresponding source code. The exception is modules, which do not map directly to a particular snippet of source code.

Don't abuse YUI Doc to write tutorials, conceptual guides, or other long form documentation. This kind of material is useful, but it belongs in a user manual or developer guide, not an API reference. Instead of using a documentation generator here, use a general documentation system such as DocBook or Sphinx. The good news is that YUI Doc output is well-suited for incorporation into a larger developer guide. YUI Doc uses Cheetah as its template engine, and if the Cheetah template language isn't powerful enough, you also have access to the raw JSON output. This flexibility enables you to transform YUI Doc output into another format such as DocBook XML or reStructuredText, suitable for inclusion as a chapter or appendix.