MODERN WEB ESSENTIALS

using

JavaScript & HTML5

David Pitt

InfoQueue
ENTERPRISE SOFTWARE DEVELOPMENT SERIES

INTRODUCTION

Mobile device proliferation is forcing enterprise IT departments to change. They must now support mobile devices, which further extends to the need for the development of mobile-friendly applications. As users get more and more savvy, simply accessing existing applications via a mobile browser just won't fly.

Enter the next phase of web development: single page application (SPA) development using HTML5 with JavaScript.

SPAs allow the enterprise to provide users with rich, responsive applications through the browser. HTML5 with JavaScript has opened the door to device-independent user interfaces. This creates extensive cost savings for the enterprise; gone is the necessity to develop multiple platforms to reach users on various operating systems. Implementing JavaScript and HTML5 with responsive design principles solves a pain point - how to reach users on various platforms without diminishing the user experience.

Another viable reason for this shift is that it eliminates a weak spot - reliability upon browser plug-ins. The plug-in approach of the past has worked well enough, but with users frequently experiencing compatibility problems, lengthy loading times, and performance issues in some circumstances, a plug-in free approach can create improvement in the user experience. Combining that with browsers' improved speed for JavaScript execution, HTML5 and JavaScript solves many web problems that currently plague the enterprise.

But, as always, change is complicated. Software architects and developers must have the right toolset and correct combinations of experience and knowledge to successfully take the enterprise through this paradigm shift.

This mini-book includes three sub-sections: JavaScript Essentials, HTML5 Essentials, and Responsive Design. They introduce you to the features of the languages and concepts, how you can leverage them to assist in enterprise development, and tips to be successful. I've successfully implemented these strategies and technologies with my teams, and hope that my experience can help you to avoid the pitfalls.

Thanks,
David Pitt

TABLE OF CONTENTS

Keyhole Software, LLC. 8900 State Line Road, Suite 455 Leawood, KS 66206 Tel 877-521-7769 http://keyholesoftware.com/

Section One: JavaScript Essentials

This section covers:

- ✔ JavaScript execution environment
- ✔ The structure of the JavaScript language
- ✔ The importance of objects
- ✔ Prototypes and inheritance
- ✔ Functions and closures
- ✔ AJAX

If you've been developing enterprise web applications, it's likely that you have applied JavaScript in some fashion – probably to validate user input with a JavaScript function that validates a form control, manipulate an HTML document-object model (DOM) for a user interface effect, or even to use AJAX to access the server to eliminate a page refresh.

Single page application (SPA) architectures allow rich, responsive application user interfaces to be developed. There are many frameworks and approaches available, excluding plug-in technologies that are JavaScript-based. This means that developers need a deeper understanding of the JavaScript language features. This mini-book section assumes you have programming experience in a traditional object-oriented language like Java or C#, and introduces features of JavaScript that allows it to be a general purpose programming language. You may be surprised by its expressiveness and object-oriented capabilities.

ENVIRONMENT

One clue that JavaScript was not originally intended to be a general-purpose language is the fact that a browser is required to execute it. The snippet below shows how an HTML page loads a JavaScript function defined inline. Normally this assumes the HTML page and JavaScript file reside on a web server.

Listing 1 – HTML page loading a JavaScript function

```
<script>
  function sayhello() {
      alert('hello world');
  }
</script>

...

<input type="button" value="say hello" onclick="sayhello();"/>
```

The `sayhello()` function defined above can invoked and executed in a variety of ways, including:

- Putting inline JavaScript tags at the beginning or end of the file when the HTML form button is clicked.
- Calling the function when a form button is clicked.
- Putting `<script> </script>` elements at the beginning or end of an HTML document, depending on the browser you're using.
- Executing JavaScript on a page load using the jQuery framework.

As you can see, there is no main method or entry-point mechanism like other languages, so a browser and an HTML page load of some kind is required to execute JavaScript. Some server-side Java solutions have recently become available, but generally speaking JavaScript for UI development requires a browser.

Open Source Steps Up

Luckily, innovations of the open-source community have filled this need. Environments have been created that allow JavaScript to be executed outside of a browser, commonly referred to as "headless," or server-side JavaScript.

Node.js is one popular open-source framework that provides a JavaScript runtime environment outside of a browser. With Node.js, JavaScript can be executed from a command line or by specifying files. Node.js is also available for most operating systems. Phantom.js is another viable option on the market. Although similar, the intent of the headless environments differs between the two. Phantom.js has HTML DOM (document-object model) available while Node.js does not. But both still provide a way to develop and test code outside of a browser and web server. Here are links to these projects:

- **Node.js** – http://nodejs.org/
- **Phantom.js** – http://phantomjs.org/

The examples presented in this tutorial can all be typed into and executed with a headless JavaScript environment. Assuming Node.js or Phantom.js binaries have been installed on your operating system, you can execute the previous JavaScript file with the expressions that follow:

```
// Java Script defined
// in HelloWorld.js text file

function helloWorld() {
  console.log("hello world")
}
helloWorld();

// Execute JavaScript
$node helloworld.js
$phantomjs helloworld.js
```

```
// Executing with a node console
```

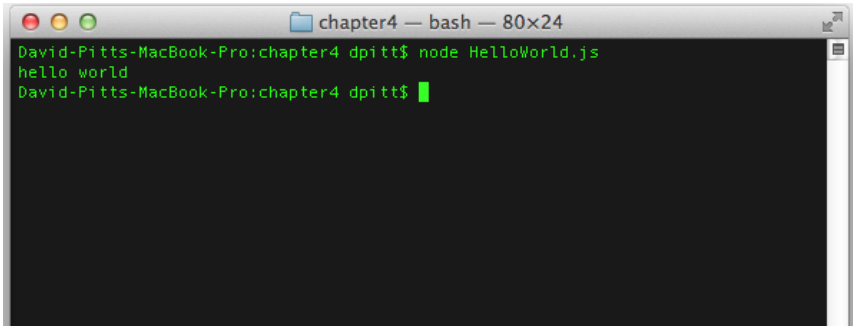


Figure 1 – Executing JavaScript from the command line

MODULARITY/STRUCTURE

JavaScript does not have a lot of structural elements like other languages. Part of this is due to its original origins as a dynamic prototype-based language. Modularity is accomplished by partitioning JavaScript functionality into separate files. Typically JavaScript libraries are defined in one giant file which can be painful to maintain and comprehend.

Upcoming sections will present modularity workarounds that are necessary for developing large applications with JavaScript, but it is still important to understand primitive JavaScript to establish a foundation of understanding.

Memory

Like when they use other object-oriented languages, developers don't specifically need to worry about or perform allocation and deallocation of memory. Since everything is an object that is dynamically created, the runtime environment will utilize a garbage-collection mechanism to reclaim objects that are no longer visible or reachable by the current execution context.

In theory, developers should not to worry about memory reclaiming or leaks. However, there are ways that object references become zombied or unreachable by the garbage collector. Closures are one way object references can become unreachable, causing a memory leak.

Global Variables

When a page with JavaScript is loaded, objects and variables created and defined with the page consume memory. The garbage collector will track and reclaim memory by objects that are no longer referenced by anything. However there is a way to define global objects that is visible during the lifetime of the browser executable within which JavaScript is executing. A runtime window variable is visible and references a globally available object. You can freely add/attach objects to the window variable. The following code shows an example of a global-variable definition.

```
window.userId = "jdoe";                          ← Global variable

var userId = "jdoe;                              ← Local variable
```

It's important to note that if you define a variable without VAR, then it's attached to the global window object. The following code shows this:

```
userId = "jdoe";                                 ← Attached to window
```

BEST PRACTICE: You should rarely need to define global variables by attaching to the window property. For an SPA, most frameworks will provide a pattern for defining global objects.

Whitespace and Semicolons

Previous tutorials have described the origins of JavaScript and pointed out its features as a dynamic object-based language. The name "JavaScript" likely came from the similarity with the Java syntax. Like Java, JavaScript syntax is simple, free-form, and case-sensitive. Expressions are terminated with semicolons.

Listing 2 – An example of JavaScript syntax

```
var abc    =    'a'  +    'b' + 'c';
var def =
          'd'   +
         'e'   +
         'f';

console.log(abc + def);
```

Semicolons are required to terminate expressions, but JavaScript cuts slack to lazy developers who forget to do so. However, it's best practice to always terminate expressions with a semicolon.

Comments

Comments are non-executable lines of code that can help document your code. You can define block and line comments.

```
/*
  Block comments
*/

...

// line comments

...

var a = "abc"; // end of line comment
```

Arithmetic Operators

Available operators are those you would expect for arithmetic operations (+, -, /, %). The + is overloaded to support string concatenation. JavaScript supports unary increment and decrement operators in the same fashion as C, C#, and Java. Listing 3 shows some operators in action.

Listing 3 – An example of arithmetic operators

```
var count = 5;                                  ← Increment/decrement
console.log( --count ); // logs 4
console.log( ++count ); // logs 5
console.log( count-- ); // logs 4
console.log( count++ ); // logs 5

var x = 5;                                      ← Assignment
var y+= 5; // y = 10;
var y-= 5; // y = 5;
var y*= 5; // y = 25;
var y/= 5; // y = 5;
var s1 = "hello";                               ← String concatenation
var s2 = "world";
var s3 = s1 + s2;
```

== and ===

The assignment operation == is used for equality checks. It will perform type conversions before checking equality. JavaScript also introduces the === operator which checks for equality. It will not perform type conversion. Study the expressions in listing 4 below and you'll see this nuance.

Listing 4 – An example of ==

```
console.log( 0 == '' );                         ← Logs true, performs type conversion
console.log( 0 === '' );                        ← Logs false, no type conversion

console.log ( 5 == '5');                        ← Logs true, performs type conversion
console.log ( 5 === '5');                       ← Logs false, no type conversion

console.log ( true == '1');                     ← Logs true, performs type conversion
console.log ( true === '1');                    ← Logs false, no type conversion.
```

BEST PRACTICE: For safety, always use the === operation for equality checks.

Flow Control

Execution paths are controlled using if/else and looping commands. They are fairly straightforward, basically the same as you have used in almost all languages.

Listing 5 – An example of execution-path control

```
var list = [1,2,3,4,5];
for (item in list) {                            ← For loop
  console.log(item);
}

for each (item in list) {                        ← Same as for, deprecated, don't use
   console.log(item);
}

for (var i=0;i<list.length;i++)
{
    console.log(list[i]);                       ← For loop with index
}

if (1 + 1 == 2 )  {                             ← If/else
    console.log("The world makes sense...");
 } else {
    console.log("The chaos ensues...");
 }

var count = 10;
while (count > 0) {                             ← Do loop
   console.log("Count = "+count--);
}
```

Code Blocks

JavaScript expressions can be enclosed in code blocks that can be attached to function definitions or defined to delineate conditional and looping expressions. Code blocks are defined using {} characters.

```
function() {...}

If/Else
if (<condition>) {…}

For Loop
for (< expression>) {…}
```

Scope

Scoping around code blocks with JavaScripts differs from what you would expect with other languages. Variables defined within conditional and looping constructs are not locally scoped to an enclosing code block. This is important to understand in order to prevent possible side effects.

What would you expect the console output to be if the expressions that follow are executed?

```
var a = 'abc';

if (true) {
      var a = 'def';
}

console.log(a);
```

You may be surprised to learn that the output would be `def`, and not `abc`. The variable defined in the conditional code block overrides the outer variable definitions. Scoping behavior within function code blocks behaves as you would expect.

Here is another example. What would you expect the console output to be when these expressions are executed?

```
var a = 'abc';

function s() {
    var a = 'def';
}

s(); A
console.log(a);                    ← Execute function
```

Variable `a` is visible and scoped to function `s()` so the log output would be `abc;`. Did you answer correctly?

AMD/CommonJS Module Specifications

Efforts have been made to make JavaScript a viable server-side language. Modularity was one area that needed to be addressed, so open-source projects created a common module API. One popular open-source project is CommonJS, which defines an API for defining modules and dependencies. Another popular module API is AMD, which stands for asynchronous module definition. Both have their advantages.

RequireJS implements the AMD specification. Loading JavaScript modules asynchronously, instead of synchronously, allows modules to load in an on-demand manner, and helps with performance, debugging, and other issues, especially in the browser environment.

The module pattern shown in this section is essentially part of the implementation used for the specifications. In practice you should use an existing proven module dependency framework to support your SPA, but understanding what is going on under the covers is helpful.

DATA TYPES

JavaScript has a dynamic-type system. This is in contrast to static-type languages such as Java and C#, which require data variables to be typed. While this provides type safety and, arguably, software that is

easier to maintain, it also requires a compilation step that JavaScript does not, as expressions are interpreted at runtime. There is much debate in the IT community regarding the agility and flexibility of dynamic versus typed languages. Both offer advantages, so you will have to make your own decisions on this debate.

Primitive

Like other object languages, JavaScript provides built-in system primitive types. Primitive types exist for performance, but some primitive types are also defined as objects that allow method calls and can be extended. As in classical languages such as C# and Java, primitive types are passed by value instead of by reference. Since primitive types are typical to most languages, the expression in listing 6 should provide enough usage information.

Listing 6 – Example primitive data types

```
var i = 100;f = 100.10;                    ← Numbers, stored as 64-bit base-10 floating point
var s = 5.98e24;                           ← Very large or small number scientific notation
var s = "hello World";                     ← String, double or single quotes
var b = true;                              ← Boolean
```

BEST PRACTICE: Use a math library when arithmetic values need to be exact. JavaScript's float has issues (e.g. 0.1 + 0.2 = 0.30000000004) and inexactness in enterprise applications is a problem.

Arrays

Arrays can be created literally or using a constructor approach. As in other languages, in JavaScript they are zero-based. Also, since JavaScript is dynamic, initial size does not have to be declared. It just has to be defined. Listing 7 shows some implementation examples:

Listing 7 – Array implementation example

```
var states = ['ks','mo','ne','co'];
console.log(states);

var countries =
countries[0] = "United States";
countries[1] = "Canada";
console.log(countries);

var mixed = ['text',true,10.00];
console.log(mixed);
```

Array Operations

Besides containing a list of data objects, arrays have methods defined to help manipulate and iterate over their contents. There are many methods. Listing 8 shows some interesting ones and how you can iterate over them.

Listing 8 – Array operations

```
var a = ['ks','mo','ne','co'];
var b = ['az','ok','tx'];

console.log( a.concat(b) );                ← New array concatenated

console.log( a.join(b) );                  ← Joins arrays into a string

a.push('me');                              ← Push elements to end of array, returns length

console.log( a.pop() );                    ← Removes element from end of array and returns it

for (var ele in a) {                       ← Loop over array
     console.log(ele);
}
```

Undefined and Null

JavaScript introduces a "not defined" data type. This is not to be confused with a null value, as they are not the same. Again, compiled languages do not need an undefined distinction, since anything not defined will result in a compilation error. Undefined data types are set to variables that do not have a value assigned. The null data type represents a value of null. The snippet below illustrates the differences:

```
var v;
console.log(v);                    ← Undefined
var v = null;
console.log(v);                    ← Null
```

Undefined and null causes confusion as many assume that variables and object properties are automatically assigned a null value when defined. The example above shows that they are assigned an undefined value or type. Since undefined equals nothing, JavaScript provides a shortcut mechanism to check for undefined variables, as with the expression below. This works for undefined and null.

```
var v;
if (v) {console.log{true)}         ← True
var v = null;
if (v) console.log(true);          ← True
```

Since undefined is assigned by default, it's safer to use the shortcut method to check for empty data values, instead of doing null checks, as shown below:

```
var v = null;
if (v == null) { console.log("value is null"); }
```

If the developer forgets to initialize a data value with null, a bug in logic could occur.

BEST PRACTICE: Don't initialize your variables and properties with null. Rely upon JavaScript's undefined default and perform checks using the `if(value)` default value.

OBJECTS

Everything in JavaScript is an object. Strings, numbers, arrays, and even functions are objects that have properties and methods. System objects supplied by the runtime environment implement objects for primitive types. They are sometimes called "wrapper objects," as they wrap their respective primitive data types.

Here are some expressions that send methods calls to some high-level primitive objects:

```
var s = 'hello world';                          ← String
console.log(s.length);                          ← Number, displays 11

var amount = 100.12345;                         ← Number
console.log(amount.toFixed(2));                 ← Number, displays 100.12
```

Like object-oriented languages, JavaScript also has a new operator. It can be used to create primitive object instances.

Listing 9 shows some examples that use the new operator. There's more about the new operator further along in this section, so stay tuned.

Listing 9 – New operators in JavaScript

```
var s = 'hello world'                           ← String
console.log(s.length);                          ← Displays 11

var amount = 100.12345;                         ← Number
console.log(amount.toFixed(2));                 ← Displays 100.12

var s = new String('hello world');              ← String Object
console.log(s.length);                          ← Displays 11
```

```
var amount = new Number(100.12345);              ← Number Object
console.log(amount.toFixed(2));                  ← Displays 100.12

var d = new Date();                              ← Date Object
console.log(d.getMonth());                       ← Displays current date month (0-11)
```

Built-In Objects

Everything in JavaScript is an object provided by the JavaScript runtime environment. Here is a list of available object types:

- **String** – Array of character values
- **Boolean** – Conditional true/false
- **Date** – Represents date time value
- **Number** – Represents all integral and floating point numeric values
- **Math** – Provides methods for mathematical functions such as abs, log, tan, etc.
- **Function** – Executable block of code that can accept parameters and return a value
- **Object** – Base object prototype for all objects
- **RegExp** – Perform pattern-matching, search, and replace on strings

You've already seen how some of these objects are used for primitive data types and arrays. What may not be obvious is that functions are also objects, or "first-class objects," meaning function objects can be created using JavaScript syntax. More details about function objects are coming up, but first let's jump into some details about JavaScript objects.

Creating Objects

You've seen system objects provided by the JavaScript runtime. As in other object-oriented languages, customer or user-defined objects can be created and used. However, JavaScript objects differ from classic "class"-based object-oriented languages which have inheritance, encapsulation, and polymorphism constructs built into the language. JavaScript objects are dynamic and to support this dynamic behavior, JavaScript takes a prototype-based approach to object creation. The next sections should give you a good idea of how this works.

There are two ways to create objects with JavaScript: literally or with a constructor function.

Literal Objects

Literal JavaScript objects are defined using JavaScript Object Notation (JSON). Some may think that JSON is just a data format used to transmit data from a server or remote system. It is, but its real purpose is to define JavaScript objects that have properties and executable methods – actually functions.

As an example, consider an object that models an account with an ID, name, and balance properties, with methods that debit and credit the account. Figure 2 shows an account-object model and source code using JSON for its JavaScript implementation.

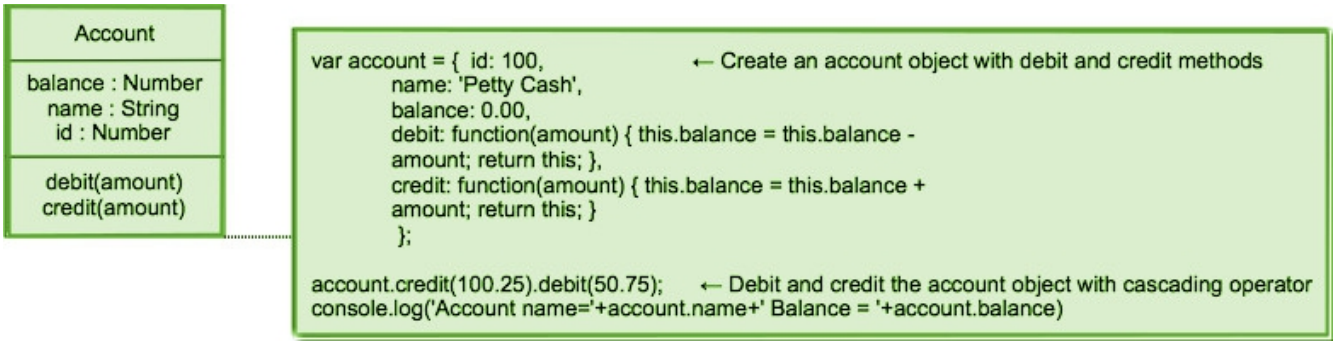


Figure 2 – Petty-cash account-object model and source code using JSON

Additionally, listing 10, below, shows a "literal" account-object definition with debit and credit methods and relevant properties:

Listing 10 – Literal account-object definition

```
var account = {  id: 100,
                 name: 'Petty Cash',
                 balance: 0.00,
                 debit: function(amount) { this.balance = this.balance - amount; return this; },
                 credit: function(amount) { this.balance = this.balance + amount; return this; }
              };
```

Notice in the listing above that the account object has properties representing account ID, balance, and name, along with methods defined for debiting and crediting. Methods are evaluated against the object instance using the "." operator. Notice how methods can cascade since the method implementations return `this`**.** The expression below shows the cascading debit and credit calls to the account object:

```
account.credit(100.25).debit(50.75);
console.log('Account name='+account.name+' Balance = '+account.balance);
```

The dynamic nature of JavaScript surfaces when adding new properties or methods, as you simply add them to the object. Here's how a `close()` account method can be dynamically added to the account object:

```
account.close = function() { this.balance = 0;};          ← Add new close function to account object
account.credit(100.25).debit(50.75);                                ←  Debit/credit account
console.log('Account name='+account.name+' Balance = '+account.balance);
account.close();                                                      ← Close the account
```

All objects are instances of the JavaScript system's object type, and each is simply comprised of an associative array and a prototype. We'll talk more about that later. Listing 11 provides some insight into how properties of an object are stored an associative array.

Listing 11 – Properties stored as an associated array

```
var object = { x: 100,                          ← Literal object
               y: 200,
               add: function() { console.log(this.x + this.y)}
               }

for (key  in object) {
     console.log(key);                      ← Outputs property names x y add to the console
}
```

Property elements can be accessed using the property name. Listing 12 shows a literal object created as a mechanism to map state abbreviations to state name.

Listing 12 – An example of array subscript syntax

```
var map = { ks: "Kansas",                       ← Object map of states
            mo: "Missouri",
            ca: "California"
          };
 console.log(map["ks"]);                        ← Access by key, outputs Kansas
```

Notice how the object property is accessed using array-type access, but instead of an index number the name of the property is specified.

Constructor-Function Objects

An alternative way to define and create an object is called an object constructor. This approach feels a little more like the classic approach, as the classical "new" operator is used to create an instance. Also, constructor objects allow an instance to be initialized with supplied values.

In listing 13, the account object is defined and used with the constructor approach. Notice how the initial balance is initialized and the instance is created with the new operator.

Listing 13 – An example of object constructor

```
var Account =                                          ← Constructor account object
        function(initialBalance) {
              this.id = 200,
              this.name = 'R&D',
              this.balance = initialBalance
              this.debit =       function(amount) { this.balance =
               this.balance - amount; return this; },
              this.credit = function(amount) { this.balance = this.balance
               + amount; return this; }
            };

var object = new Account(5123.25);                      ← Create instance
console.log('Account name = '+object.name+' Balance = '+object.balance);
```

BEST PRACTICE: Variables that reference constructor functions are typically camel-cased with the first character capitalized. Create a factory that hides the "new" keyword since it's easy to forget.

Functions are discussed in an upcoming section. But, although you have seen constructor objects look like functions, look closely. They aren't really functions; they provide a way to initialize objects, enclose the structure of an object, and provide a way to "construct" objects when needed. This is as opposed to defining them literally.

Prototypes

Now that we have explored a couple of ways to define and create objects, let's dive under the covers and see whats going on with objects.

JavaScript is referred to as a prototype-based language. This can be contrasted to the classic class-based languages in which classes contain methods and properties are defined. At runtime, class metadata is turned into a type-system-object model. However, available classes must be defined at construction time. JavaScript's dynamic nature applies a prototyped-based approach to objects. As indicated, JavaScript objects are instances of an object with an associative array (key/value) of other data objects or function objects. And, as we have seen, objects can be created at runtime, with methods and properties being added at will.

Every constructor-based object definition has a prototype property that points to the same prototype of the constructor function. Adding a new method to the constructor object is visible to all instances that have been created from it, as shown in listing 14.

Listing 14 – Adding a new method to the prototype

```
var Name =  function() {                          ← Name constructor-object definition
              this.first = null,
              this.middle = null,
              this.last = null};

  var nameA = new Name();                         ← Instance is created
  nameA.first = 'Jane';
  nameA.last = 'Doe';

  var nameB = new Name();                         ← Another instance is created
  nameB.first = 'John';
  nameB.last = 'Doe';

  Name.prototype.middle = 'Chris':                ← Middle name added to Name prototype

  Console.log(nameA.middle);                      ← Middle name is visible to both instances
  Console.log(nameB.middle);
```

Listing 14 shows the definition of a constructor-function object named `Name`. It has two properties: `first` and `last` name. Two instances are created using the `new` operator and properties are set. Notice these two instances reference the constructor-objects prototype property. Then, a middle-name property is added to the prototype reference. This makes the middle-name property visible to both instances, `nameA` and `nameB`.

Prototype Chaining/Inheritance

The previous section showed how methods and properties that need to be shared across all instances can be made global to all instances by adding properties to the constructor-objects prototype. When a property or method is sent to an object during execution, the runtime environment will look for the property/method in the current instance, then in the existing prototype, and up the chain until `Object.prototype` is encountered. This is shown in figure 3.

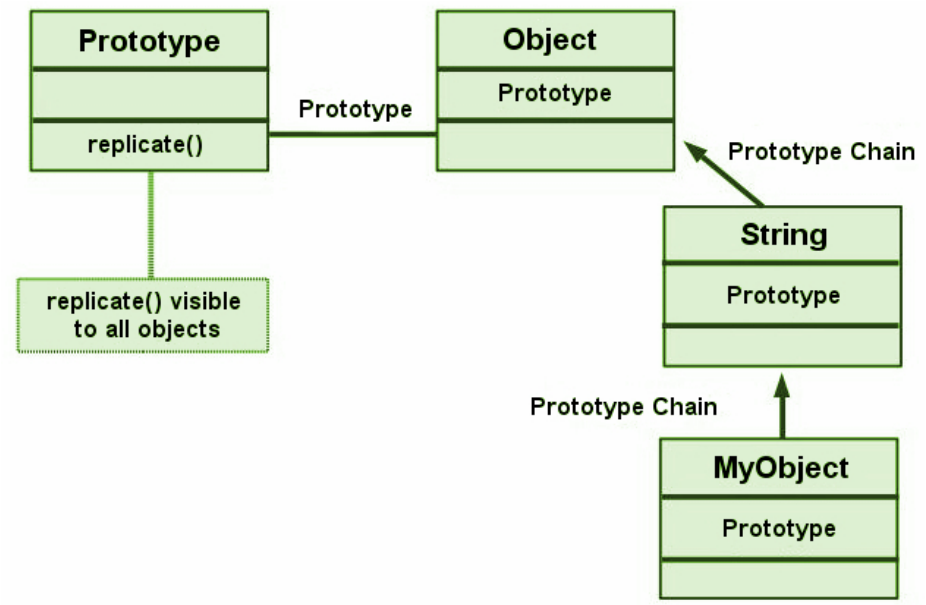


Figure 3 – Prototype-chaining inheritance

This is referred to as prototype chaining and is how inheritance works in JavaScript. The next expression demonstrates this chaining behavior by adding a `replicate()` method to the built-in objects prototype property.

```
Object.prototype.replicate = function(count) {                  ← All objects now have
    return count < 1 ?  '' : new Array(count + 1).join(this);     replicate() method
};

console.log("Hello".repeat(3));                                 ← Logs "HelloHelloHello";
```

This new `replicate()` is now visible to all objects. While it works for string objects, it will probably throw an execution error when run against other objects since the `split()` method expects a string object. Adding the `replicate()` method to the string-objects prototype makes it visible only to string objects.

```
String.prototype.replicate = function(count) {                    ← All string objects now have
    return count < 1 ?  '' : new Array(count + 1).join(this);       replicate() method
};

console.log("Hello".repeat(3));                                   ← Logs "HelloHelloHello";
```

As you can see, this is similar to inheritance. Let's apply this to an anonymous function, shown with these expressions in listing 15.

Listing 15 – Prototype-chaining inheritance with an anonymous function

```
var XY = function() { this.x = 100,
                      this.y = 200 };

var a = new XY();                                               ← #A Create instance
a.multiply = function() { return this.x * this.y;};             ← #B Add multiply method
console.log(a.multiply());                                      ← Logs 2000
var b = new XY();                                               ← #C Create another instance
```

```
console.log(b.multiply());                                  ← #D Error multiply() is not defined

XY.prototype.multiply = function() { return this.x * this.y;};   ← #E Add multiply to XY prototype
console.log(b.multiply());                                  ← #F Multiply is available and
                                                              displays 20000
```

The example defines an XY constructor object **#A**, then assigns a multiply function **#B** and executes it. Then another XY instance is created **#C** and the multiply method is issued, but it is undefined **#D**. This is because the `multiply()` method was added to an object, but not to its prototype. Adding the multiply function, the XY constructor-functions-objects prototype **#E**, makes it available to all instances created from the XY object **#F**.

Prototypes in Action – Implementing the Singleton Pattern

Prototype behavior can be seen in action when trying to implement the singleton pattern with JavaScript. Singletons are a pattern commonly seen in classical object languages. The intent is to implement a global, single instance of an object. With JavaScript it is easy, too easy, to make an object a global variable. Simply set an object reference to the globally visible window object. Here is how the current user of an application can be made global:

```
CurrentUser = {userId: 'jdoe', name:'John Doe' };
```

or

```
window.CurrentUser = {userId: 'jdoe', name:'John Doe' };
```

For a large application, putting this global module definition in its own JavaScript file will help make things more modular and maintainable. However, if this file is loaded or referenced multiple times, it will wipe out previous values. The application of the single pattern ensures that only a single instance of an object exists no matter how many times this file is loaded. The singleton pattern for a global `CurrentUser` is shown in listing 16.

Listing 16 – Singleton-instance implementation

```
var CurrentUser =  function() {
     var User = function() {
            var userid = '';
            var name = '';

            return {

                  getName : function() {
                        return name;
                  },
                  getUserId : function() {
                        return userid;
                  },
                  setName : function(newName) {
                        name = newName;
                  },
                  setUserId : function(newUserId) {
                        userid = newUserId;
                  },
            };
     };
     if (User.prototype._instance) {
            return User.prototype._instance;
     }
     User.prototype._instance = new User;
     return User.prototype._instance;
}();
                                                      ← Set singleton value
CurrentUser.setUserId("jdoe");
CurrentUser.setName("John Doe");

console.log(CurrentUser.getUserId());
```

JavaScript prototype behavior provides a convenient way to enforce the singleton instance of a current user no matter how many times the file is loaded. A closure is defined that returns the singleton instance.

The singleton instance is set to the constructor-functions prototype for the first request and subsequent calls just return the original instance.

Also, notice how getter/setter access methods were defined, as this is a practice not typically done in JavaScript OO development. In this particular example, it shows how access to the data can be encapsulated with methods.

FUNCTIONS

Everything in JavaScript is an object, and functions are no exception. JavaScript functions represent a modular unit of execution and are considered first-class objects, as they can be created literally and dynamically, assigned to variables, and passed around as data. Literal functions you have already seen in this tutorial. Here's a basic literal function definition you've probably seen before:

```
function helloWorld() {
    console.log("hello world");
}
                                                ← Execute function
helloWorld();
```

Literal functions are akin to method implementations in classic languages like Java and C#. JavaScript, being dynamic in nature, does not really have much in common with compiled-based languages. Comparisons are probably made to Java due to the "Java" in "JavaScript."

One advantage that dynamic-based languages like JavaScript have over their compiler-based competitors is that functional-programming capabilities are made possible by the ability to treat chunks of code like data. This can lead to elegant designs that do more with less code.

Anonymous/Closures

Anonymous functions or closures are powerful elements in JavaScript. Closures are a key element of functional-programming techniques. Other languages such as C# provide closures. Java has been promising closures for a number of releases, but has yet to provide this capability.

Closure functions are defined and assigned to a variable that can be passed around just like a piece of data, and then executed. You'll see closures commonly used to provide callback and event handling functionality. Functions can be defined and assigned to a variable that can then be passed around and executed.

Check out listing 17, below, as this example contrasts a literal function with one that is created anonymously, or as a closure.

Listing 17 – Literal function contrasted with closure

```
function helloWorld() {                          ← Literal-function definition
    console.log("hello world");
}

helloWorld();                                    ← Execute function, outputs "hello world"
var hello = function() { console.log("hello"); }; ← Define anonymous function and assign to variable
hello();                                          ← Execute function, outputs "hello"

var log = function(text) { console.log(text); };  ← Define anonymous function with argument
log("Hello World");                               ← Execute function, outputs "Hello World"
```

Let's make things a little more interesting with the next example. These expressions define an anonymous function that is passed into another function and then executed. Notice how arguments are handled in the following expressions:

```
var hello = function() { console.log("hello"); };        ← Define hello function
var executor = function(func) { console.log(func()); };  ← Define executor function

executor(hello);                                         ← Invoke executor function, pass in hello
                                                           function as an argument, outputs 'hello'
```

Memoizing

Since functions can be treated as data, an interesting feature becomes available, called "memoization." This feature provides the ability to hide or remember data. Variables scoped by an outer function and referenced by an inner function remember their values every time the function is invoked.

Memoization can be seen in listing 18. This example implements a literal function that returns anonymous functions for a specific operation. Each operation function can then be executed and results returned. Notice how the sum variable is remembered between operation calls.

Listing 18 – Memoization closure

```
function operationFactory(operation,initialValue) {      ← #1 Function that returns operation
                                                            closure function and sets initial value
  var sum = initialValue;                                ← Sum variable will be memoized by
                                                            operation closure functions
  if (operation == "+")
     { return function(x) { sum += x;return sum;} };
  if (operation == "-")
     { return function(x) { sum -= x;return sum;} };
  if (operation == "*")
     { return function(x) { sum *= x;return sum;} };
}

var add = operationFactory("+",0);                       ← #2 Get operation functions from factory
var subtract = operationFactory("-",200);                   and assign to variable
var multiply = operationFactory("*",10.0);

add(100);                                          ← #3 Execute add function, sum variable will be 100
console.log(add(200));                             ← #4 Add 200, output will be 300, as the previous
add variable is remembered
subtract(100);                                     ← Console output will be 50
console.log(subtract(50));

multiply(0.5);                                     ← Console output will be 2.5
console.log(multiply(0.5));
```

Let's walk through the code, as this is an important concept to see in action. Step **#1** defines an `operationFactory` function that accepts an operation identifier and an initial value. When called, a closure is returned that performs an arithmetic operation against the outer function `sum` variable. Step **#2** gets operation closure functions from the factory and assigns them to variables. Step **#3** then invokes the function with a value of 100. Step **#4** invokes the add function again with 200 and outputs 300 to the log.

Since the add function was executed two separate times, you might expect that the second add execution would output 200. Closure memoization remembers the sum variable across executions of the same function instance.

What do you think the output of the expression shown below will be?

```
var add = operationFactory("+",100);
add(50);
add(50);
console.log(add(50));
```

If you guessed 250, you are right. The initial value of the memoized sum variable is set to 100.

Execution Context

This is a concept that causes confusion, especially with closures. Classical object-oriented developers understand the concept of the `this` keyword, which provides a way to reference an existing object reference. This is especially useful when having to access properties/methods and in passing around object references.

However, in JavaScript `this` may not be the `this` you were expecting. Since JavaScript is dynamic code, it has the concept of an execution stack, and since JavaScript runs on a single thread, only one code block is visible in the execution context. Its `this` references that execution context.

Here are some common JavaScript execution contexts for the `this` operator:

- Window
- Document
- Function
- Method
- Constructor Method

Here is where context problems often occur. Say you are defining a literal-object method that defines a closure, one that performs a calculation and prints results when a button is clicked. It could look something like the source below:

```
var add =  {                                                    ← Literal object
     sum: 0,
     execute: function(x,y) {this.sum =  x + y; }               ← this is in object-method context
     print: function() {
      var btn = document.getElementById("print_button");      ← Get document button object
      btn.onClick = function() {
            this.execute(100,100):  ←  WILL FAIL, why? Context, or this will be in document context
            alert(this.sum);        ←  WILL FAIL, why? Context, or this will be in document context
       }
     }
  }
```

However, this will fail when the button is clicked with an error indicating that `this.execute()` and `this.sum` are undefined. Why? Because when the closure function is executed, `this` will be in the HTML document context.

How can this be fixed? Memoization is the answer. The correct context is preserved and referenced in the closure by defining a variable that references `this`. This variable is then referenced by the closure, preserving correct context reference, and not using the current context of `this`.

Here is a JavaScript snippet that works. Notice how the this context reference is memoized in the closure:

```
var add =  {                                                    ← Literal object
     sum: 0,
     execute: function(x,y) {this.sum =  x + y; }               ← this is in object-method context
     print: function() {

      var btn = document.getElementById("print_button");       ← Get document button object

      var _this = this;                                        ← Reference to this method context

         btn.onClick = function() {
             _this.execute(100,100):                  ←  Won't fail, memoized _this is correct context
             alert(_this.sum);                        ←  Won't fail, memoized _this is correct context
       }
     }
  }
```

You can also supply an execution context. It can also be specified and supplied to a function using the function call or apply methods.

Here's an example:

```
function add(a,b) {                                  ← Function definition
      return this.x + this.y + a + b;
}
var o = {x:100, y:100};                              ← Object definition
console.log(add.call(o,200,200));                    ← Invoke function with call, specifying a
                                                       context for this. Outputs 600 to console.
```

This is something that you will encounter often, especially in SPA development, when you will be writing a lot of JavaScript client logic to create user-interface elements, and for reacting to events in the HTML document context.

Function Closures in Action and Modularity Support

Modularity and dependency injection are not mechanisms built into JavaScript. With JavaScript, you can use folders/files to help modularize code. Compare this with other languages; enforcing modularity in Java is

accomplished using package definitions while C# uses namespaces. As dependency injection is not part of any language, frameworks have filled the gap. This is changing as both C# and Java have indicated a future in implementing built-in dependency-injection mechanisms in their language specifications.

Modularity and dependency injection are key for managing SPA-like applications that have rich user-interaction requirements. However, the goal of this section is to understand JavaScript functions and closures. Understanding how to apply modularity will help with this goal. Figure 4 illustrates the concepts of modularity and dependency injection, showing how modules can be used and dependent modules injected:

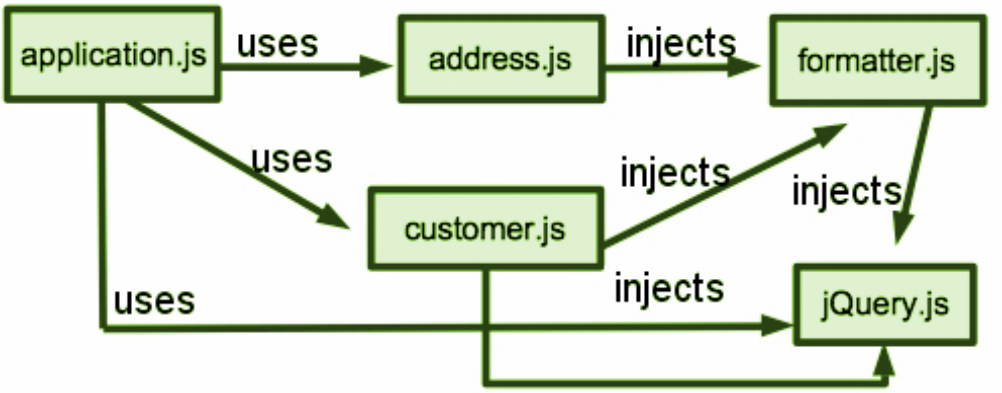


Figure 4 – Modularity and dependency injection

Available JavaScript modularity mechanisms are functions or code defined in separate files that are loaded using a `<script>` tag. Modularity lets you hide complexity and information from consumers of a module. Classical OO languages provide language-access visibility to methods and properties of objects. Access modifiers are available and can be specified to make properties and methods private. Private-access modifiers prevent developers from changing or accessing elements that are not a part of a module's public-access API. JavaScript does have an access modifier, but a pattern has been invented that allows methods and attributes to be hidden.

The module pattern evaluates a function closure that returns a literal-object-function method that accesses the private data. Listing 19 shows how an address object is modularized:

Listing 19 – Module pattern for address

```
var addressModule = (function () {                          ← Address-module reference to literal object
  var address = ['street','city','state','zip',];           ← Array holds address segments (street, city,
  return {                                                    state, ZIP, etc.)
   street: function (street) {
       address[0] = street;
      return this;
    },
    city: function(city)  {
       address[1] = city;
       return this;
   },
   state: function(state) {
       address[2] = state;
       return this;
   },
   zip: function(zip) {
        address[3] = zip;
        return this;
   },
   format: function () {
      return address[0]+"\n"+address[1]+','+address[2]+'  '+address[3];
    }
  };
})();                                                       ← Function closure evaluated on load
addressModule.street('123 Easy Street').city('Lawrence').state('KS').zip('123456');  ← Invokes module
                                                                                       methods
console.log(addressModule.format());                        ← Outputs formatted address to console
```

Notice how in listing 19, the address elements are stored in an array. Methods are defined to allow array elements to be set and to return a formatted address. Only the methods the literal object returns are visible to the user of the address object. This pattern effectively makes the address array visible or private to the returned literal object.

Dependency Injection

Another pattern commonly found in classical languages is dependency injection, a pattern for referencing other, "dependent" modules. Doing this in a consistent manner lets dependent modules communicate and provides a way to report or assert modules that are not present, which can help with maintenance and debugging. Listing 20 shows how the previous module pattern introduces a module that formats addresses:

Listing 20 – Injecting a dependent address module

```
var addressModule = (function (printer) {                ← Printer module supplied to address module
  var address = ['street','city','state','zip',];
  return {
   street: function (street) {
       address[0] = street;
      return this;
    },
    city: function(city)  {
       address[1] = city;
       return this;
   },
   state: function(state) {
       address[2] = state;
       return this;
   },
   zip: function(zip) {
        address[3] = zip;
        return this;
   },
   format: function () {
      return printer.format(address);                    ← Engage printer module and format address
    }
  };
})(printerModule);                                       ← Printer-module reference, assume this is a
                                                           global variable
addressModule.street('123 Easy Street').city('Lawrence').state('KS').zip('123456');
                                                         ← Check the counter value and reset, outputs 1
console.log(addressModule.format());
```

The address module is "injected" as an argument in module closure, and a reference to the injected module(s) are applied in the module-loaded function call.

EXCEPTIONS/ERRORS

When errors occur during JavaScript execution, an error object is thrown. You may be surprised to know that problems can be caught by errors being thrown or raised. Exceptions are an integral part of the Java, C#, and C++ languages. You don't see a lot of exception-handling code in JavaScript. The typed languages mentioned above have the advantage of exception types that can provide additional debugging information to explain why the exception occurred.

Exceptions in JavaScript are actually errors that have occurred during execution. Let's say you want to catch an undefined error. The following example will show how this is accomplished with a try/catch code block.

```
try {
     var x = 0;
     var z = x + y;

  } catch (error) {                                      ← Y not being defined will throw an exception
     console.log("Y is not defined, you big dummy :)");  ← Catch block outputs message to console
  }
```

You can also throw or raise errors in your code using the throw clause. You can throw any object type and this instance will be available in the catch block. Here are some examples of throwing an error with various object types:

```
Throw -1;                                          ← Throw -1 number
throw 'Error Message';                             ← Throw error message string
throw {code: 100, message: 'error message' );      ← Throw object literal instance with error info
```

Catch blocks will have access to the object instances that are thrown.

AJAX

Asynchronous JavaScript and XML (AJAX) is a technology supported by all browsers and is a simple mechanism that provides a profoundly improved user experience. Before AJAX, browsers and JavaScript code would be executed whenever an HTML page was requested from the web server, at which point the browser, along with JavaScript, would render an HTML user interface.

AJAX provides a way to request XML or string data from the web server and then process this data with JavaScript. The ability to update individual HTML elements at any granularity likely started the movement towards SPA applications we see today. You're probably already using AJAX if you develop web applications using JEE or .NET server-side MVC frameworks. Many UI components leverage AJAX to provide a more responsive user interface.

Let's dive into how an AJAX request is made in JavaScript and how its response is processed. You've probably noticed the XML emphasis with AJAX. This is due to expecting an AJAX server request to return an HTML DOM, which is in XML format. Listing 21 shows JavaScript that makes an AJAX call to return an HTML/XML document from the web server:

Listing 21 – XML AJAX request

```
var xmlhttp = new XMLHttpRequest();                          ← Create request instance

xmlhttp.onreadystatechange=function()                        ← Process server-side results from call
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    document.getElementById("myDiv").innerHTML=xmlhttp.responseXML; ← XML response
    }
  }

xmlhttp.open("GET","info.html",true);                         ← Server-side resource to access
xmlhttp.send();                                               ← Initiate AJAX request
```

The `xmlhttp` is a global object provided by the JavaScript runtime. The open method specifies a put, post, or get operation, the file to open, and returns "true" from the server if this an asynchronous call, which makes it AJAX. Otherwise it's a synchronous call. A callback function assigned to the `onreadystatechange` property will process the results.

AJAX requests can also return string values instead of XML from the server. Since a JSON string is easily turned into actual JSON with JavaScript, SPA applications will typically utilize server URL requests that return JSON strings containing only application data. SPA applications produce HTML on the client side with JavaScript, so JSON data will be merged with client-side dynamic HTML. Listing 22 shows how JavaScript invokes a server-side URL that returns a JSON string and then turns the string into a JSON object.

Listing 22 – JSON AJAX request

```
var xmlhttp = new XMLHttpRequest();                          ← Create request instance

xmlhttp.onreadystatechange=function()                        ← Process server-side results from call
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
   {
    var json = JSON.parse(xmlhttp.responseText);             ← Parse JSON text to JSON
    }
  }

xmlhttp.open("GET","info.do",true);                          ← Server-side resource to access
xmlhttp.send(); ← Initiate AJAX Request
```

The mechanism of AJAX is supported by all modern browsers and is essential to SPA-based applications. Upcoming tutorials will introduce SPA JavaScript frameworks. One mechanism some of these frameworks implement is a way to access server-side data in a RESTful manner. AJAX allows these frameworks to access server-side data then update a portion of the user interface, with no page refresh, reinforcing a rich user interface.

SUMMARY

This section introduced beginning and advanced JavaScript programming features and concepts. A thorough understanding of these topics is necessary for building web SPA applications, as much JavaScript will be developed, used, and applied throughout that process.

If some concepts are still fuzzy, I recommend that you play around with and modify some of the samples to see if it helps your understanding.

References

- Osmani, Addy. *Learning JavaScript Design Patterns.* Volume 1.5.2 http://addyosmani.com/resources/essentialjsdesignpatterns/book/
- Mozilla Developer Network. https://developer.mozilla.org/en-US/
- W3 Schools. http://www.w3schools.com/
- CommonJS Wiki Community. http://wiki.commonjs.org/wiki/CommonJS
- Asynchronous Module Definition (n.d.) In *Wikipedia*. http://en.wikipedia.org/wiki/Asynchronous_module_definition

Section Two: HTML5 Essentials for SPA Development in the Enterprise

This section covers:

- ✔ What is HTML5
- ✔ HTML5's role in an enterprise SPA
- ✔ Features compelling to enterprise SPAs
- ✔ Additional HTML5 features
- ✔ Mobile HTML5 support
- ✔ HTML5 browser support issues and resolutions

HTML5 is a technology topic that is most likely being discussed in your IT organization. Much press and hype have been made about the technology. Will it live up to the hype or will it not? I think it will, as its success will be tied how single page applications (SPA) can utilize HTML. And as long as web browsers remain ubiquitous, HTML will continue.

If there's a sudden shift away from the browser to native applications then HTML5 might be in jeopardy. Arguably, enterprises can cover more devices by implementing a browser-based SPA with JavaScript and HTML5. This tutorial will describe the new features of HTML5, making an attempt to point out features that are compelling to enterprise SPA development.

WHAT IS HTML5

HTML5 is just the next version of HTML. The HTML5 specification was finalized in 2013 by the W3C. For the previous few years, the specification had been in flux, still being worked on by the W3C working groups. Browser manufactures have chosen to implement some of the specifications, but some high-profile companies are backing off the technology (Facebook for example). Coupling this with the specification previously not finalized, adoption has been slow.

With the finalization of the specification, many of the features are already supported by recent versions of all the major browser manufacturers. You can assume that all of the features described in this section are supported by the browser versions shown in figure 5, unless otherwise noted.

Figure 5 – Browser HTML5 support includes Google Chrome, Mozilla Firefox, Safari, Opera, and version nine of Internet Explorer.

There are many sites that you can visit with a browser to test HTML5 features and report how compatible your browser and others are. One site is aptly called HTML5 Test and is available at www.html5test.com.

COMPELLING SPA-RELATED FEATURES

There are many useful HTML5 features, but not all provide a compelling feature for enterprise SPA development. While this is arguable, most enterprise applications don't readily use multimedia features. If you remember, enterprise applications typically have heavily-used create, read, update, and delete (CRUD) requirements. These are the considerations used to determine compelling enterprise SPA features. Remaining sections will cover other cool features of HTML5.

UI Elements

Standard HTML form elements have always been an afterthought. Form elements were not an original part of the HTML specification but were eventually introduced to allow user interaction with web sites and applications. JavaScript and CSS magic have been used to specialize form elements for UI metaphors common on many applications. HTML5 has recognized this and introduced the following new elements.

PROGRESS

How many progress UI indicators have you implemented? Many libraries and utilities provide progress whirligigs and indicators. In HTML5, progress or activity can now be implemented using the following element, as shown in figure 6.

```
<progress max="100"></progress> Indeterminate</
<progress max="100" value="50"></progress> 50%
<progress max="100" value="100"></progress> 100%
```

Figure 6 – Progress indicators available through this HTML5 feature and the code for each

METER

The meter element allows increments or unit of measure to be tracked and displayed. High, low, max, and min values can be defined. Figure 7 shows a list of examples with screen shots:

```
<label>Preferred usage</label> <meter min="50" max="250"
low="100" high="200" value="120"></meter></br>
<label>Too much traffic</label> <meter min="1024" max="10240"
low="2048" high="8192" value="9216"></meter> </br>
<label>Optimum value</label> <meter value=".5"
optimum=".8"></meter>
```

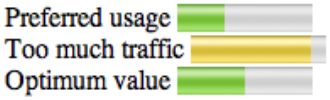
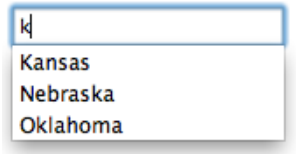


Figure 7 – Meter element examples

DATALIST

This is a form element that is long overdue. Datalist elements define a set of option elements that can be tied to an input field. When a user enters input into the field, it matches against the data list. Figure 8 shows a sample datalist UI and element.




```
<input list="states" />
<datalist id="states">
  <option value="Kansas">
  <option value="Missouri">
  <option value="Colorado">
  <option value="Nebraska">
  <option value="Oklahoma">
</datalist>
```

Figure 8 – Datalist element example

This is a common user-interface requirement that, up until now, had to be emulated using JavaScript and CSS magic. Here is a caveat: it is currently supported by all browsers except Safari.

KEYGEN

Securing web traffic with SSL is the standard way to secure browser-to-server communications. The keygen tag generates public/private keys. The private key is stored in a local keystore and the public key is sent the server. Figure 9 shows an example that defines a keygen element with a form.

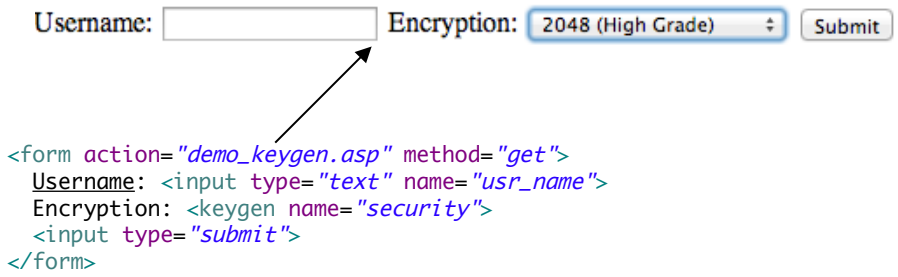


Figure 9 – Keygen element example

The keygen element has attributes that toggle a challenge dialog and algorithm-key type selection, such as RSA, DSA, or EC.

OUTPUT

When implementing an HTML form, output often needs to be displayed within the form. Elements within the form are all elements that accept input from the user. Yes, you can set values to an input text field and disable it or make it read-only. However, why not have an element to display form output? The output element provides this capability.

The example in figure 10 shows a form with two input values. In this case, the output element displays the sum of these two input values.

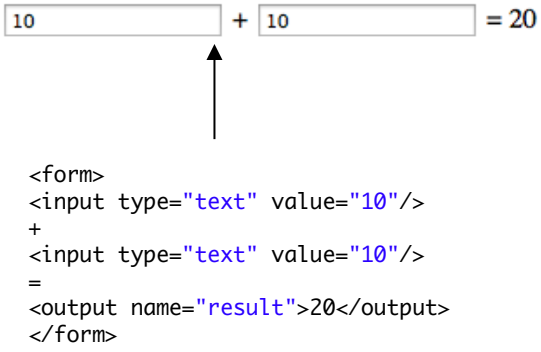


Figure 10 – Example of a form with two input values and an output element, with the code to achieve this feature

Data input using a text input element is the primary way to implement data-entry forms in browser applications. Lots of JavaScript/CSS mastication and UI framework libraries have been created to emulate input fields for specific data types, such as dates, numbers, email addresses, and telephone numbers.

New Input Types

HTML5 has eliminated the need for this workaround by specifying built-in input types. Figure 11 shows an example of an input element's date type definition.

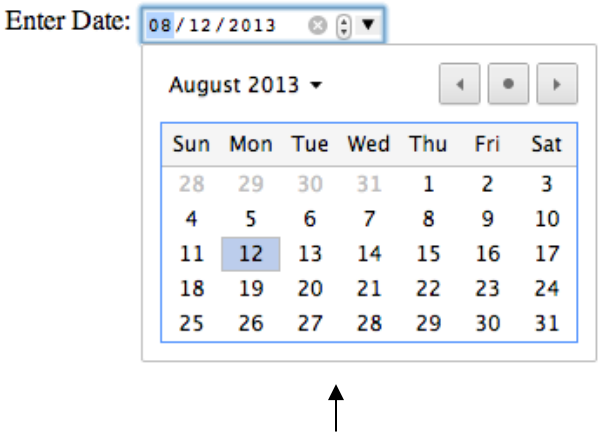


Figure 11 – Example of the date type definition in an input element and code required to achieve this feature

It's nice to have built-in support for a date picker as you no longer have to introduce a date-picker component. There are many more useful input types available in the specification. The downside is that not all browsers currently support these types. If a browser does not support a type, it falls back to a plain input-text field.

Here is a list of the new elements:

- tel – Telephone number
- search – Displays typical search-specific input field
- email – single or multiple email formatted entries
- datetime – UTC date time
- date – Date only
- month – Year and month
- week – Week of year in the format 2014-w04 means week 4 of 2014
- time – Hour, minutes, seconds
- datetime-local – Date time and no time zone
- number – Numerical value
- range – Numerical range
- color – Color chooser

As of late 2013, not all browsers support all of these input types. But now that the HTML5 specification has been completed, eventually these tags will be as common as the `<button>` tag.

New Input Attributes

Much SPA functionality revolves around form input and processing. The previous sections showed new input types. Another behavior that every SPA, or every web application for that matter, takes advantage of is form validation.

In SPA architecture, the type of data that can be input is defined on the client and is validated when a form is submitted. All web applications have to implement form validation. Many frameworks and approaches have been defined to validate input forms over the years. A lot of effort has gone into validating forms in client-side browsers. HTML5 has recognized this requirement and provides built-in features for form validation and more. The figure below shows a sample form that uses the new input types and form-validation attributes.

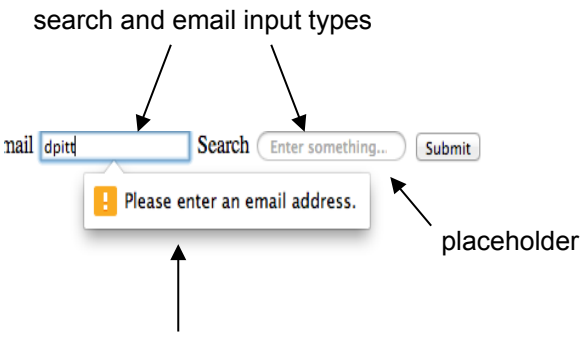


Figure 12 – New form attributes and input types

The HTML in figure 12 implements autofocus, validates that a valid email address has been entered, and reports an error message on form submit. It prevents form submission unless valid and displays a search input type with placeholder information.

```
<form>
 <label>Email</label> <input type="email" required autofocus/>
 <label>Search</label> <input type="search" placeholder="Enter something..."/>
  <input type="submit"/>
</form>
```

Before these new HTML5 features, the developer would have to implement a lot of JavaScript and CSS. Now it's native to the browser, and less code means fewer bugs with better performance.

Custom Data Attributes

A common practice in SPAs is to query an HTML page's object model for elements and then operate on these elements. Querying using the built-in document API can be done against elements or attributes. The HTML5 specification introduces the ability to define a custom attribute for any HTML element. Custom attributes are defined through an attribute prefixed with `data-`. Here is an example of a custom attribute defined to indicate a role for a `<div>` tag:

```
<div data-role="admin">
..
</div>
```

This attribute can be used when the page is rendered to, for example, only display this div for users with an administrator role. The popular mobile framework jQuery utilizes custom `data-` attributes extensively to render mobile touch-sensitive user interfaces from standard HTML tags. As an example, jQuery mobile list interface and its HTML are shown in figure 13.

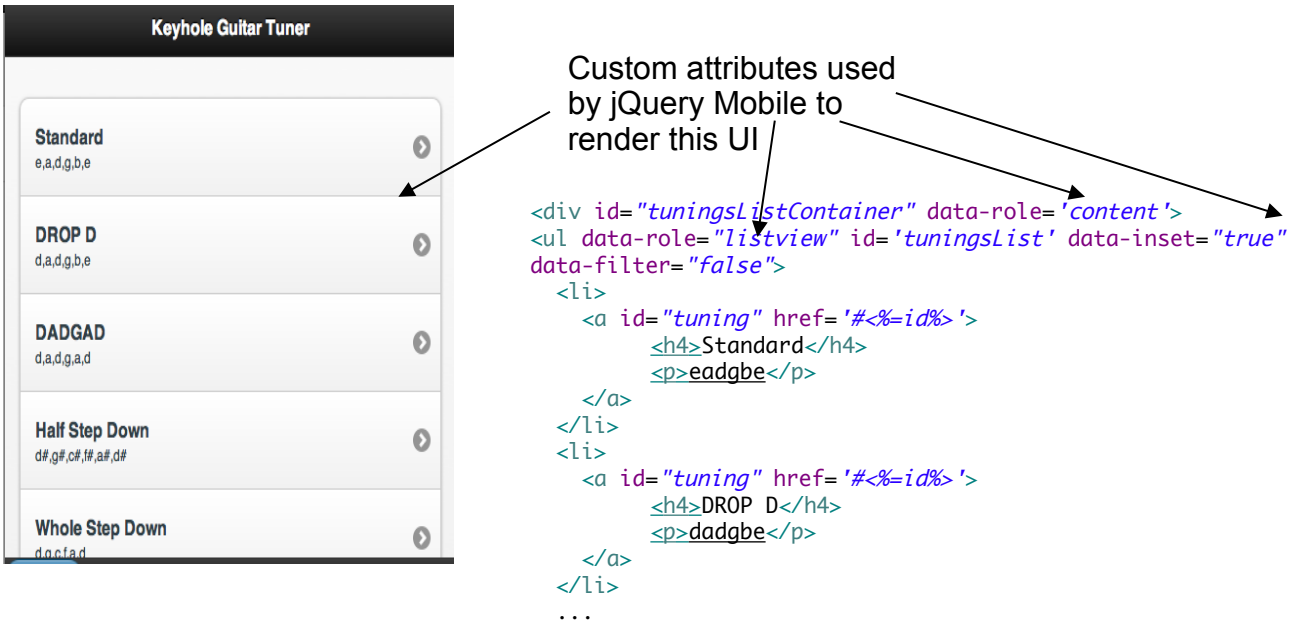


Figure 13 – Custom attributes applied to a jQuery mobile UI

The data-role attribute of the `<ul>` tag indicates to the jQuery mobile framework that this will be a list. You can also see the data-filter attribute set to false. If set to true, a search entry field and the ability to search would be rendered. The `listView` role indicates that the `<ul>`, `<li>`, and `<a>` tags are rendered and styled as a touch-friendly mobile interface.

The custom `data-*` attributes allow metadata to be attached to standard HTML elements so that they can be accessed and manipulated for specific, customized needs.

Local Storage

Until HTML5 came on the scene, the only way to store information locally in the browser, and between user sessions, was to use cookies. But user cookies are only accessible from server-side logic and storage size is limited to 4096 bytes per user cookie. This does not support an SPA-based architecture, as its user interface logic is located client-side.

HTML5's local-storage feature provides a simple way to store and access data locally from JavaScript. Local storage is essentially a key/value data-store for strings. A local-storage object is accessible from the `localStorage` global variable. Methods are defined that allow key/values to be stored and retrieved. Here is a usage example:

```
localStorage.userid = 'dpitt';          ← Set value in local storage
localStorage['userid'] = 'dpitt';

var id = localStorage.userid;           ←  Get value from local-storage object
var id = localStorage['dpitt'];

localStorage.clear();                   ← Clear local storage
```

Only string values can be stored in `localStorage`, but JavaScript's seamless ability to convert strings to and from JSON allows objects to be stored in the local cache. Here is an example that uses local storage to cache a list of state objects in local storage:

```
var states =
 { ks:'Kansas',mo:'Missouri',co:'Colorado',ok:'Oklahoma',ne:'Nebraska'};
 localStorage.states = JSON.stringify(states);  ← Turn JSON into string, save in local storage

 states = JSON.parse(localStorage.states);      ← Obtain state JSON from local storage, parse to JSON
 console.log(states.co);                        ← Outputs Colorado to console
```

Using local storage as a cache can improve performance by cutting down on remote-server access calls. It can also be used to store session-specific information in order to improve the user experience. It is also a required element to help support an application to function in a disconnected state. This might not be a requirement for desktop enterprise applications, but may be a consideration for mobility support.

`LocalStorage` objects are tied to an originating URL from which the application is served. Its contents are permanent between sessions, with no expiration date. It is cleared and managed with JavaScript code.

Session Storage

Session storage provides local storage but it is only visible during the browser session. When this ends, `sessionStorage` is cleared. A global `sessionStorage` variable is available and has the same access methods and API as `localStorage`. Saving the current-user object in `sessionStorage` is shown:

```
var user = {userid: 'dpitt', first: 'David', last: 'Pitt' };
sessionStorage.user = JSON.stringify(user);
```

Like `localstorage`, `sessionstorage` can only save string values.

Inspecting

Browser debugging tools provide a way to inspect local-storage values. The screenshot in figure 14 displays Safari's local-storage inspecting capabilities.

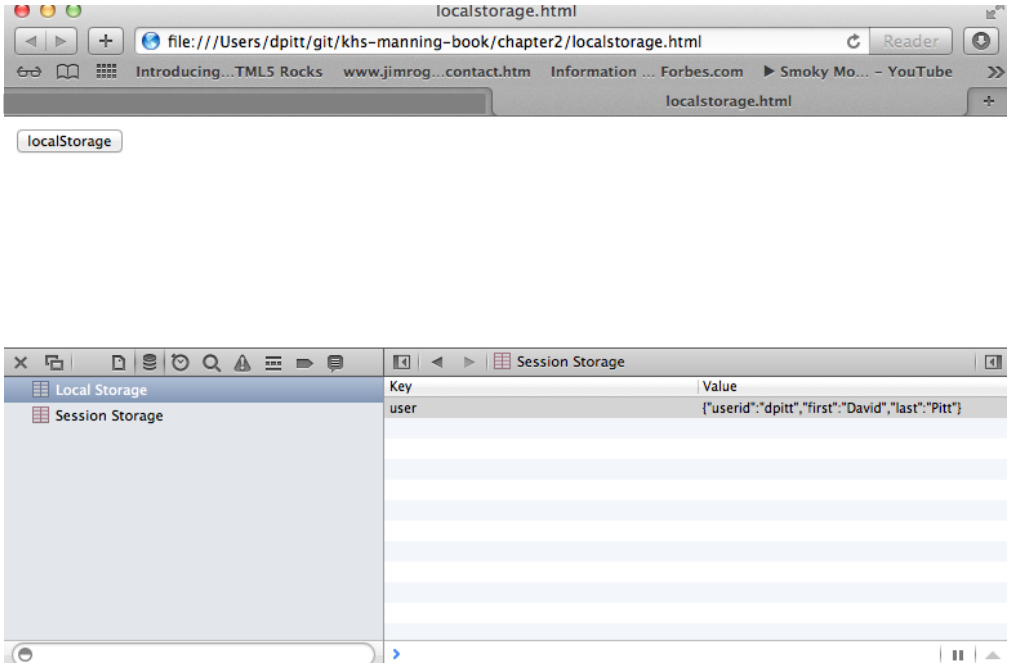


Figure 14 – Safari's local-storage and session-storage inspector

Local storage is limited to 50 megabytes of data, which should be more than enough to support SPA caching requirements.

Fallback

Local storage is a feature well-supported by almost all browsers. IE versions prior to 9 don't have this support, but there are many ways to implement `Fallback` or substitute mechanisms to compensate. JavaScript can check for local-storage and session-storage support with the following code.

```
if(typeof(Storage)!=="undefined")
  {
    console.log("Local Storage not supported...");
  }
else
  {
      console.log("Local Storage is supported...");
  }
```

You can also make local storage transparent by implementing your own storage object that encapsulates local-storage behavior; if it's not present, a cookie or global object can be used.

WebSockets

WebSocket technology has a profound impact on building highly interactive, low-latency, rich user interfaces for the browser. Before WebSockets, browsers accessed the server using a request/response-based HTTP protocol. Traditionally, the client would initiate requests and a connection with the server. AJAX technology allowed the client to emulate communication with the server using a polling technique. The client would periodically check and "poll" the server for information. AJAX allowed server-polling to happen without a page refresh. This works, but HTTP has overhead. As the server does not have a true connection to the client but is just emulated, latency occurs.

WebSockets allow for a true two-way client-server connection over sockets without the overhead of HTTP. A comparison of HTTP and WebSockets client/server connectivity is shown in figure 15.

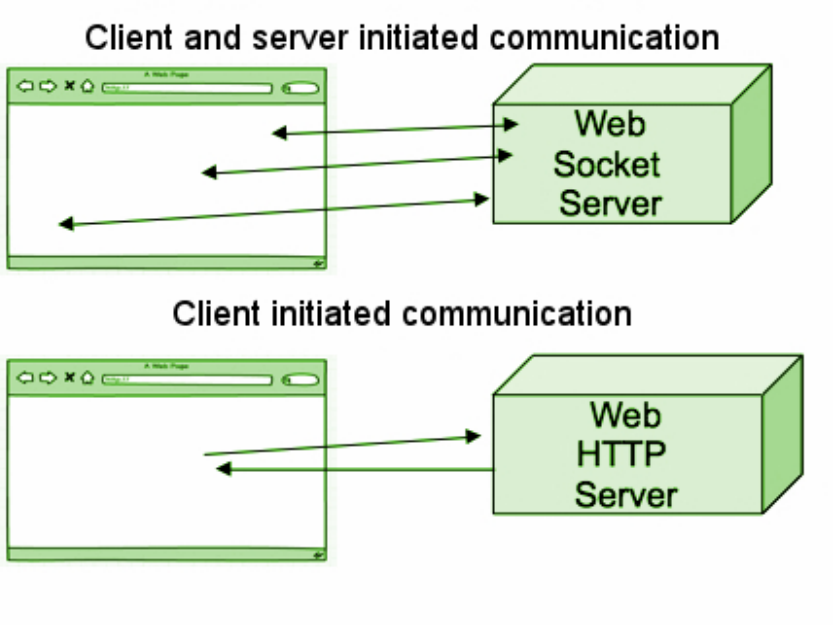


Figure 15 – HTTP versus WebSocket client/server connectivity

The type of applications that require the low-latency, realtime feedback that WebSockets support are:

- Realtime dashboards
- Game development
- Chat and messaging

WebSockets are used in a client browser with JavaScript. The best way to see how they work is to see some code in action, so let's walk through some usage examples. Clients open a connection to a WebSockets server with the following JavaScript expression:

```
var connection = new WebSocket('ws://manning.websocket.org/ticker', ['soap',
    'xmpp']);
```

The connection object's first argument is a WebSocket-specific URL, and the second argument is a supported subprotocol. With a URL specific to the WebSocket specification, notice the `ws:<servername>` format. The second argument is a supported sub-protocol.

Connections raise events for errors, messages from the server, and when the connection is successfully opened. Here is how these events are handled:

```
connection.onopen = function () {                    ← Send message to server on connection open
  connection.send('hello world');
};
connection.onerror = function (error) {              ← Handle error message
  console.log('WebSocket Error ' + error);
};
connection.onmessage = function (e) {                ← Handle message sent from server
  console.log('Data from Server: ' + e.data);
};
```

String and binary data are the only types supported by the WebSocket specification. Receiving binary data can be accomplished as either a blob or array buffer. Binary data can be sent and received as a blob or binary array buffer. Strings need to be converted to other types. Here is an example of how a number is converted:

```
connection.onmessage = function (e) {
  console.log('Number sent back from server ' + new Number(e.data));
};
```

With the client-side API introduced, the next section will identify server software that supports WebSockets.

WebSockets Server

Many server options support HTTP server/request client-server architecture through the web. Since WebSockets is new, fewer options exist, but the compelling possibilities of this technology is sure to change this. Even though WebSockets do not implement the HTTP protocol, the specification requires an initial handshake to occur in HTTP before the WebSocket connection is opened. Once opened, it communicates using the lightweight WebSocket protocol. It makes sense that existing HTTP-based servers could be augmented or enhanced to honor WebSockets.

Here is a list of some currently available WebSocket server implementations:

Node.js

- Socket.io
- WebSocket.node
- ws

Java

- Jetty

.Net

- SuperWebSocket

Fallback

WebSockets are both new and compelling, but this is still one of the features slow in adoption. So, how can the enterprise start taking advantage of the technology now, especially considering the fact that enterprises are typically not quick to upgrade or grab the latest browser version?

There are some JavaScript libraries that implement client and server implementations of the WebSocket API. This tactic allows any browser to use WebSockets, as the server side uses HTTP connections to emulate the WebSocket protocol. It won't perform as pure WebSockets, but you'll still be able to apply WebSockets.

ADDITIONAL HTML5 FEATURES

Before HTML5 came available, 3D graphics, animations, and videos required plug-in technologies to be delivered to browsers. A good example of this is Flash, which enabled game development and highly produced graphical content. This worked well enough, but users frequently experienced compatibility problems, lengthy loading times, and performance issues.

Considering the downtrend on laptops and desktops and the uptrend of mobile devices for games, videos, advertising, and graphic content, this is a perfect time to shift to SPAs and abandon plug-in technology in favor of native browser support. Here are these graphic, multimedia, and offline capabilities.

Canvas

Canvas is one of the most compelling new features of HTML5. A canvas is a region set within a browser that allows 2D and 3D graphics to be defined with a JavaScript API. Here's a sample canvas element:

```
<canvas id="guitar">
</canvas>
```

Assuming the canvas element shown, images can be loaded and drawn upon using the canvas JavaScript API. The figure and code snippet that follow show how JavaScript can be used to display a guitar image and music notes on the canvas.

Figure 16 – Guitar fretboard with notes drawn on the canvas

The following is the JavaScript code that draws the fretboard image and music notes onto the canvas:

```
var note = ['D','A','D','G','A','D'];          ← Notes to draw
var stringsX = [26,63,102,138,174,212];            ← String x coordinate
for (i = 0;i < strings.length;i++) {           ← Loop over strings
   var x = strings[i]-9;
   var y = 150 - 32;
   ctx.fillStyle = "white";                ← Draw note on canvas context
   ctx.fillText(note[i], x,150);
}
```

HTML has been able to display images since its beginning, but being able to directly draw in the browser is compelling. Developers can use the canvas in the same manner as for a graphic context in an operating-system display. Shapes, lines, and polygons can be drawn and click events can be captured. With 2D and 3D capabilities, what can be done with the canvas API is limited only by one's imagination.

Audio

Prior to HTML5, playing audio files (MP3, WAV, and ogg, for example) would require a browser plug-in. Now, simply defining the `<audio controls>` element tag will load and display an audio-control widget that will play the file. See figure 17.

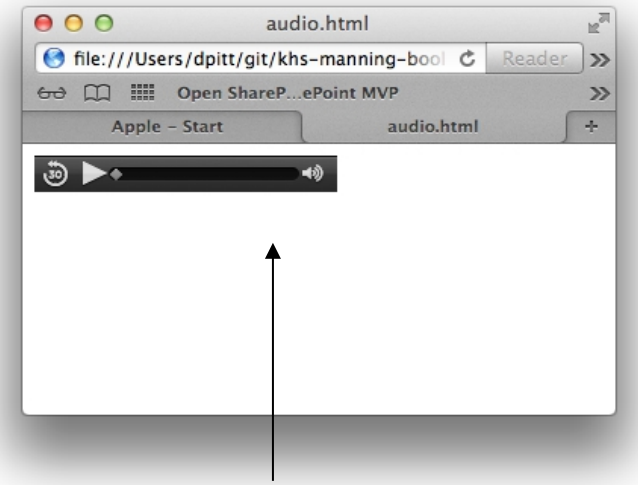


Figure 17 – An example of the <audio controls> element tag with a screenshot of the control displayed in the browser

```
<audio controls>
  <source src="audio/a-1.mp3" type="audio/mp3">
</audio controls>
```

You can also use an <audio> tag by itself to play sounds without displaying the control widget, or even use the JavaScript API to apply more control to the file. Here is a snippet of JavaScript code that uses a jQuery selector to obtain reference to an <audio> object in the DOM, and then issues the play() method:

```
var a = $('#audio1')[0];                ← Get audio-element object using jQuery selector
a.play();                               ← Play sound
```

Scalable Vector Graphics (SVG)

Displaying vector graphics has commonly been done using PNG or JPEG images. Unfortunately, JPEG image quality suffers as the image is displayed on different screen resolutions. PNG images have good quality, but don't scale well due to their size. By contrast, SVG images are smaller and scale well with varying screen resolutions. SVG images can be defined using the <img> tag just like any other image. Here is an example:

```
<img src=”logo.svg‟ alt=”logo”/>
```

Gradients, animations, image manipulation, and other effects can be applied using the <svg> tag.

Listing 23 – An example of gradient definition with the SVG tag

```
<svg xmlns="http://www.w3.org/2000/svg">
  <defs>
    <linearGradient id="blueshiny">
      <stop stop-color="#a0caf6" offset="0"/>
      <stop stop-color="#1579df" offset="0.5" />
      <stop stop-color="#1675d6" offset="0.5"/>
      <stop stop-color="#115ca9" offset="1"/>
    </linearGradient>
  </defs>
  <g id="button" onclick="alert('ouch!');">
    <rect fill="url(#blueshiny)" width="198" height="83" x="3" y="4"
     rx="15" />
    <text x="100" y="55" fill="white" font-size="18pt" text-
     anchor="middle">Press me</text>
  </g>
</svg>
```

SVG images are much smaller in size than any other format and look great even when resized. With so many available devices of varying screen sizes and processing power, SVG images are a good solution.

Video

HTML5 browsers provide an embedded video viewer. Like the other features mentioned, this eliminates the need for plug-ins.

CSS3

CSS3 is a separate specification that the W3C also governs. It applies styling and layout elements to HTML elements. It's rare to find an enterprise developer skilled in applying CSS, but they do exist. If you find one, they can be worth their weight in gold. The right CSS touches can make an application's user interface beautiful. We say that an enterprise application should be useable, not beautiful, but we all know that a graphically appealing user interface goes a long way.

CSS style sheets are usually used and not created, and maybe the organization has produced or used a standard style sheet. Developers will sometimes modify and tweak styles for padding and alignment by overriding CSS elements.

CSS3 has introduced new elements that allow for animations, effects, fonts, and colors to be applied with pure CSS as opposed to using a plug-in or JavaScript code. Here's a list of CSS3's new UI features:

- Borders – New properties allow for round and shadowed borders
- Backgrounds – New properties for greater control over sizing and appearance
- Text effects – Shadow effects and word-wrapping properties for text
- Fonts – Non "web safe" fonts can be used
- 2D and 3D transforms – Properties and methods to rotate, scale, spin, turn, and move images
- Transitions – This style allows time-sensitive effects to be added to elements without JavaScript or plug-ins
- Multiple columns – Allows divisions to be divided into columns

Application Cache

Caching application resources from the server improves performance and allows applications to execute without connectivity. This is one reason why this section is located in the section of additional/non-relevant SPA features. Arguably, most enterprises have a robust networking infrastructure and require access to enterprise data sources. Having to build applications that operate without connectivity is not usually a requirement, at least for now. This could change going forward as the "bring your own device" movement surges in the enterprise.

The application cache will locally store resources as they are encountered. If connectivity is lost, the application or website uses the resources in the cache. Large resource files like images have always been cached by browsers, but HTML5's application cache standardizes caching to all resources: JavaScript files; JSON; and CSS, for example.

The cache is enabled by simply defining the manifest attribute in a document's `<HTML>` tag. This will cause all resources loaded by the browser to be stored locally in the application cache.

```
<html lang="en-us" manifest="example.appcache" >
   ….
</html>
```

The manifest attribute defines a manifest file. If this definition is not present, all resources will be cached as they are encountered.

Explicit Caching

A developer can also specify in advance what resources to pull from the server and cache by defining a `cache.manifest` file. This text file specifies where files for caching can be listed.

Listing 24 – Example of cache-manifest definition

```
CACHE MANIFEST # 2012-9-1:24
# Explicitly cached 'master entries
CACHE:
app.js
css/jquery.mobile-1.1.0-rc.2.min.css
css/jquery.mobile.structure-1.1.0-rc.2.min.css
css/jquery.mobile.theme-1.1.0-rc.2.min.css
css/styles.css
index.html
libs/AMDbackbone-0.5.3.js
libs/backbone-0.9.2.js
libs/css/normalize.css
libs/jqm-config.js
libs/jquery-1.7.2.js
libs/jquery.mobile-1.1.0-rc.2.
```

Files specified under the `CACHE:` directive in this file will be loaded asynchronously. Wildcard entries are not supported, so all files to cache have to specify full file paths.

Some resources require a connected state and cannot be cached. You can also specify files not to cache using the `NETWORK:` directive. Files and resources under this section will bypass the application cache. The following is an example URL endpoint that requires network connectivity:

```
NETWORK: http://localhost:8080/khs-backbone-example/sherpa
```

Another nice capability is the ability to communicate when a resource is not available. The `FALLBACK:` directive can be associated with resources. If connectivity is lost then the specified HTML file is displayed to inform the user that connectivity has been lost. Here is an example:

```
FALLBACK: /*.html  /offline.html
```

Interestingly, the `FALLBACK:` directive allows wildcards to be specified.

JavaScript API

JavaScript can also be used to check on the status of the application cache. The cache object is attached to the global-window object. See listing 25 for an example.

Listing 25 – Example of an expression that returns application cache's current status

```
var appCache = window.applicationCache;
switch (appCache.status) {
  case appCache.UNCACHED:                          ← Uncached == 0
    return 'UNCACHED';
    break;
  case appCache.IDLE:                              ← Idle == 1
    return 'IDLE';
    break;
  case appCache.CHECKING:                          ← Checking == 2
    return 'CHECKING';
    break;
  case appCache.DOWNLOADING:                       ← Downloading == 3
    return 'DOWNLOADING';
    break;
  case appCache.UPDATEREADY:                       ← Updateready == 4
    return 'UPDATEREADY';
    break;
  case appCache.OBSOLETE:                          ← Obsolete == 5
    return 'OBSOLETE';
    break;
  default:
    return 'UKNOWN CACHE STATUS';
    break;
};
```

Also, the JavaScript application-cache object has a method to request an update of the cache. Here is the code snippet:

```
var appCache = window.applicationCache;
appCache.update();
```

You can also control when a cache update is applied. The following snippet shows how the updated cache can be swapped in:

```
if (appCache.status == window.applicationCache.UPDATEREADY) {
  appCache.swapCache();

}
```

Making an application work in a non-connected state is not a major enterprise requirement today, but as organizations embrace mobile devices and the work force becomes more distributed, this could change. It is good to know that the application-cache mechanism can be applied on an "at will" basis using a cache manifest.

MOBILITY AND HTML5

Mobility is something that enterprises have no choice but to address. As more and more users bring tablets and mobile devices into the enterprise, they will expect to access applications on them. Enterprises ignoring mobile devices will be missing a user-productivity and efficiency opportunity. At some point it could even be a recruiting opportunity – people love their mobile devices.

There are three approaches that can be taken for mobility. They are depicted in figure 18.

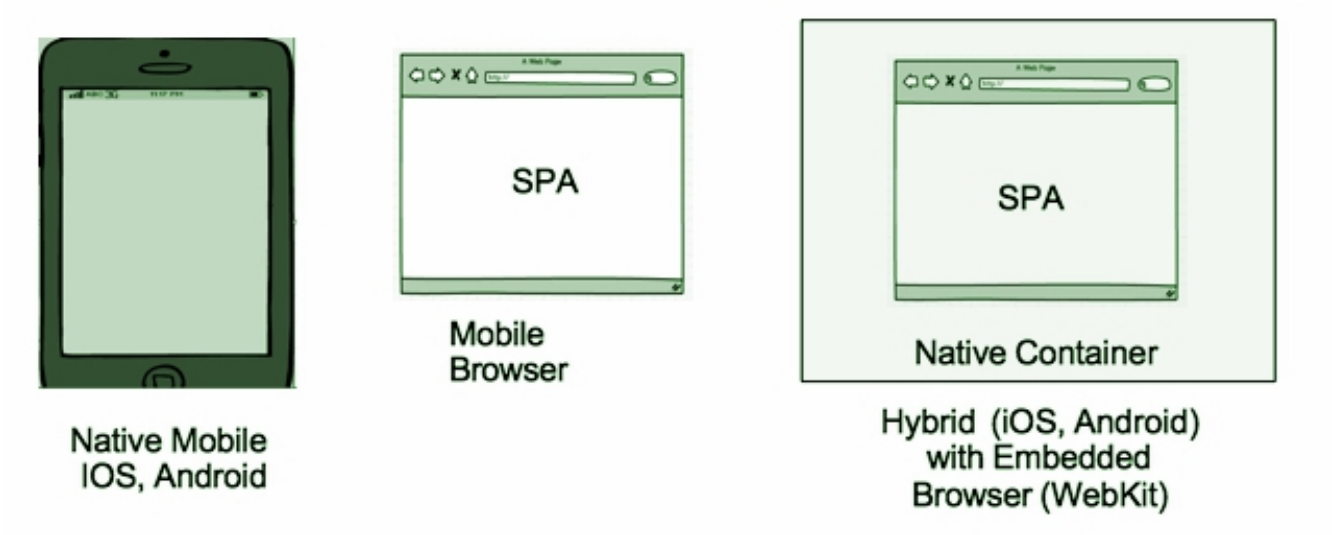


Figure 18 – Architecture options for mobile application development

SPA and HTML5 are components involved in browser and hybrid-based applications. They both utilize HTML5 and JavaScript to develop the application. Browser-based applications allow applications to run on any device with a supporting browser. Hybrid applications typically allow cross-device user-interface development using HTML5 and embedding a browser in a native container. The native container allows access to device peripherals and the embedded browser allows one user interface to be developed for all devices. Native speaks for itself and offers the best user experience, but at the cost of having to develop applications for every supported device.

HTML5 has some specific mobility features. Local storage and the cache manifest, which we've already covered, compensate for the limited connectivity issues that can arise with mobile devices. Also, SVG images are more efficient and scale better on mobile devices. Additionally, CSS features support a responsive UI design. Other features include geolocation. Here's an example of how to obtain the current longitude and latitude of a device.

```
var show = function(position) {                          ← Closure to display position coordinates
     console.log(position.coords.longitude);
     console.log(position.coords.lattitude)
};

if (navigator.geolocation) {                             ← Check for browser support
     navigator.geolocation.getCurrentPosition(show);      ← Geolocate coordinates
} else {
     console.log("Browse does not support geolocation");

}
```

Also, a device's camera can be accessed to capture pictures and video with HTML. Here's an example of how a picture can be captured:

```
<input type="file" accept="image/*" capture="camera">
```

There's also a JavaScript API that can be used for more flexibility. The video and camera features of HTML5 are currently supported by the mobile versions of browsers.

Hybrid Mobile Approach

As the name implies, hybrid mobile applications are a cross between a native application and a HTML5/JavaScript-based application accessed from the browser. Native mobile applications rely upon "native" UI components and this provides a responsive, clean-looking application. These applications are specific to mobile operating systems.

Hybrid mobile development frameworks implement a one-size-fits-all solution by providing native containers for each operating system that interacts with an HTML5-based browser component. APIs from the native browser component provide access to native operating system APIs. User interface look and feel is developed using HTML5/JavaScript, which is then packaged and bundled locally into a native application binary. The advantage is that a single UI can support multiple types of mobile devices.

Here are a few popular hybrid-based mobile development platforms, but there are more:

- **PhoneGap** – phonegap.com Also, here's link to an introductory PhoneGap blog by one of our Keyhole developers – Introduction to PhoneGap.
- **Sencha Touch** – sencha.com
- **Appcelerator Titanium** – appcelerator.com
- **Apache Cordova** – cordova.apache.org
- **Appspresso** – appspresso.com

There is another approach that more ambitious enterprises can take for hybrid development. It takes more engineering, but if the population of supported mobile devices is known, custom native containers can be developed that consume HTML5/JavaScript UI elements. This provides additional control and more flexibility in application deployment and provisioning. However, it does come with additional support requirements.

SUMMARY

HTML5 features provide native support for application capabilities that were solved with JavaScript libraries and frameworks. Developers have to introduce and manage these libraries, leading to a larger code base that can affect performance and maintainability, especially for enterprise SPA applications, which can have robust functionality. But not all browsers consistently implement HTML5 features.

I believe that in time, especially since the HTML5 specification has been finalized and the "bring your own device" (BYOD) approach has gained ground, enterprises will be driven to implement mobile-based applications with HTML5 rather than writing native applications.

So, do enterprise-application architects write SPAs that support specific HTML5 features? Or should they just continue to use frameworks that provide HTML5 features? These are not yes-or-no questions; the answer primarily depends upon the sensitivity to browsers used and availability in the enterprise.

Usually, organizations control the users' workstations and push an authorized browser and version. This gives IT a target browser for which to specifically write software. The enterprise may allow users to install a later version or a browser from another manufacturer, but enterprise applications are not guaranteed to function. For more liberal enterprises that do not lock down desktops and allow multiple browsers to be installed, applications can be written to specific browser versions with certain HTML5 features.

Enterprise developers can protect themselves by abstracting away specific HTML5 features from direct access. The SPA-architecture-supporting frameworks described in upcoming tutorials can provide this direction. Also, there are cases where the developer can apply a simple wrapper API to HTML5 features. This lets fallback capabilities handle incompatibilities.

References

- HTML5 Rocks. http://www.html5rocks.com/en/tutorials/websockets/basics/
- W3 Schools. http://www.w3schools.com/

Section Three: Responsive Design

This section covers:

- ✔ Responsive design in the enterprise
- ✔ Mobile first or one-size-fits-all
- ✔ How responsive design works
- ✔ Responsive-design frameworks
- ✔ Bootstrap, a responsive-design framework
- ✔ Responsive UI layout ideas

Enterprises are feeling the pressure to develop applications that allow users to use their own devices to access enterprise applications. Most devices will have a browser application, just like a desktop device, so current web applications are accessible without doing anything but providing connectivity to the corporate network. However, odds are that these "built for desktop" browser applications will not be fully useable, especially if any kind of data entry is required. Why are they unusable? Because devices' screen sizes differ so much, and even though the application will run in a mobile browser, many users will constantly have to scroll and expand the display to comprehend, navigate, and interact with the application. And a browser application assumes a standard keyboard and is not designed for mobile touch screens.

What options do enterprises have to solve this problem? Of course, native, non-browser applications can provide an excellent user experience. However, going down the native mobile-application path also introduces issues of deployment and security, and a big learning curve. It requires commitment to application development. If an organization relies only on one homogenous device, this could be an option. But most enterprises would have to develop capabilities in multiple platforms (for example, Android, iPhone, and Blackberry), which is both expensive and time-consuming.

Enterprise SPA offers a solution by employing responsive design for user-interface implementation. This tutorial will describe what it is and how a responsive user interface can be implemented with HTML5 and CSS3.

WHAT IS RESPONSIVE DESIGN?

Simply, the term "responsive design" can mean a lot. But for this tutorial, we will take it from the perspective of enterprise application development. Generally speaking, the term means that an application's user interface is "responsive" to the accessing device.

If done correctly, the user experience when accessing an enterprise application is as clean and consistent as if the interface was built for the specific accessing device. This is accomplished not only by CSS3 browser features, but also by making appropriate user interface design decisions.

Screen resolutions vary significantly by device. Here in figure 19 are some common screen sizes for mobile and desktop devices.

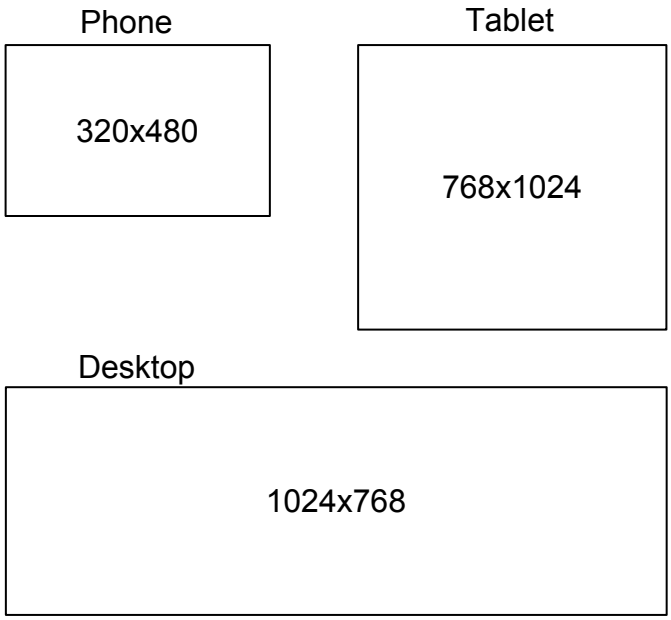


Figure 19 – Common device screen resolutions

A responsive design will change the user interface layout and elements based upon the specific user interface size. As an example, consider the elements shown below in the standard browser view of the SPA called Command Grok:

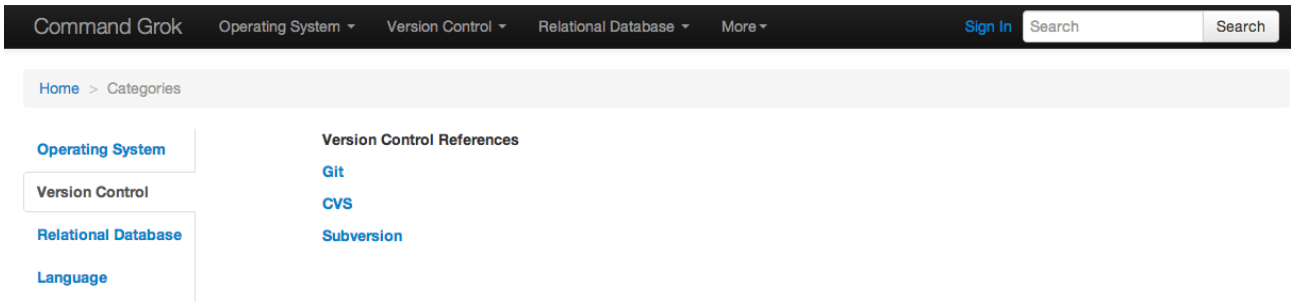


Figure 20 – UI with navigation elements across the top

Notice how in figure 20, the navigation options are displayed vertically across the top of the screen. When the screen dimensions change, the navigation collapses. By contrast, see how the application is viewed on a tablet device as shown in figure 21:

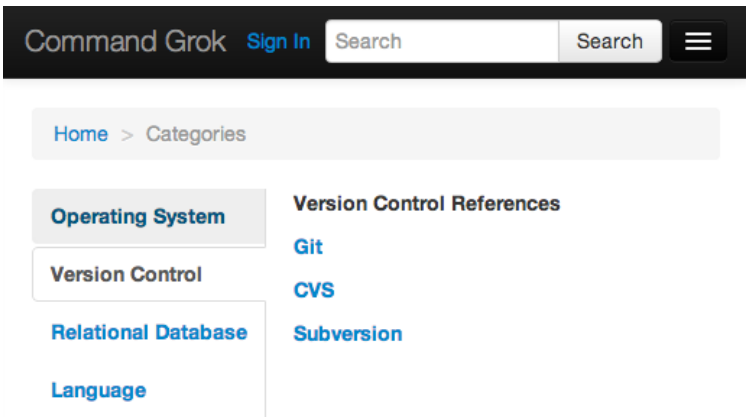


Figure 21 – Navigation options collapse when a smaller resolution is detected.

This specific responsive feature is accomplished by using a responsive layout. However, more than just using a responsive layout is required in order to make a web-based application responsive to other devices.

IMPLEMENTING RESPONSIVE DESIGN

Responsive design is applied with a combination of HTML/CSS magic and user-interface design that uses interface layouts, widgets, and input patterns that are device-neutral and take advantage of touch capabilities. HTML and CSS provide a nice platform for laying out and partitioning elements to allow a limitless, creative user-interface experience. Experienced HTML/CSS user-interface designers currently apply their craft more to marketing-focused web sites or applications. They don't usually find themselves in the ranks of corporate IT projects. That will likely change.

Mobile First

One approach to pervasive design is to take a "mobile first" design approach. This means that the user interface is designed for a specific, targeted mobile-device screen resolution and mobile-type widgets first, as then a desktop browser will present the UI without resolution issues. But let us be realistic. While a mobile web interface will display and run on a tablet or desktop, it will look odd and not take advantage of the extra real estate. Since we are speaking about enterprise applications, it is safe to assume that targeted devices could include tablets, desktops, laptops, and some kind of soon-to-be-invented device.

Starting with a mobile-phone user interface is not realistic, but starting with a tablet-type device is. Look around your office and take an inventory of who is using some kind of tablet and who takes notes on a tablet in your next meeting. I believe there is going to be some kind of convergence on surface-based, touch-gesture-based devices. Look no further than Windows 8 tiles. So why not leverage your application user interface now?

CSS Media Queries

Version three of CSS added media queries. This allows CSS designers to "query" a device's width, height, and orientation parameters. CSS can be applied using conditional media-query expressions that will allow CSS properties to be set based upon the device's browser display properties.

Listing 26 shows CSS expressions that use the media-query capabilities to change the background color depending upon the device screen size.

Listing 26 – CSS expressions with media query

```
@media screen and (max-width: 600px) {                          ← Background to purple if width < 600px
     .one {
            background: #F9C;
     }
     span.lt600 {
            display: inline-block;
     }
}

@media screen and (min-width: 900px) {                          ← Background to orange if >900px
     .two {
            background: #F90;
     }
     span.gt900 {
            display: inline-block;
     }
}

@media screen and (min-width: 600px) and (max-width:900px) {    ← Background to blue if >600px and less
     .three {                                                     900px
            background: #9CF;
     }
     span.bt600-900 {
            display: inline-block;
     }
}
```

```
@media screen and (max-device-width: 480px) {                    ← Background to xx if <480px
     .iphone {
            background: #ccc;
     }
}
```

As you can see, media-query expressions provide a way for CSS designers to apply different CSS styling based upon device size with a single CSS implementation. Most current browsers support the media query. Older browsers that do not support media query have to implement and maintain separate CSS files for each device type. Also, some frameworks will read and parse CSS on the fly and transform the CSS attribute based upon a device size. The device size can be determined by the HTTP user-agent property.

Responsive Layout

The media-query CSS element itself does not give you a responsive user interface, as it's just a mechanism used to implement CSS that applies a responsive design framework. Most enterprises don't have the wherewithal to write custom responsive CSS for their enterprise applications. Since SPA-based enterprise applications are constructed with JavaScript, HTML, and CSS, some of the responsive design is going to depend upon the UI framework you decide to use.

Use a CSS Framework

Writing your own responsive UI framework would be a time-consuming endeavor, so one decision in selecting your SPA UI framework needs to be how responsive it is. CSS separates both the look and feel and responsive logic from HTML, and therefore markup. However, it is useful to have an introduction to some of the responsive-layout frameworks that are available. Some common frameworks will be introduced in a later section.

<DIV> not <TABLE>

User interfaces can be laid out using HTML `<table>` and `<div>` elements, with `<div>` being the preferred mechanism for performance and flexibility. The reasoning behind that is that more HTML has to be processed for `<table>` and `<div>` has more layout-control properties that CSS can manipulate.

Avoid HTML Pos, Width, Height, and Type Attributes

No matter what UI framework you select, or how you define your SPA's HTML user interface elements, you should avoid defining position, width, height, and type attributes – unless you are using a UI framework with responsive-design APIs that require usage of these attributes.

Responsive UI Layout Frameworks

A common approach for CSS frameworks is to provide a responsive-layout framework that implements a grid-based CSS theme. A set number of columns are defined for a grid, and then UI HTML elements are placed within column grids. The media query and some math determine the number of columns that can appear. They are displayed horizontally, and remaining columns wrap horizontally, as figure 22 displays:

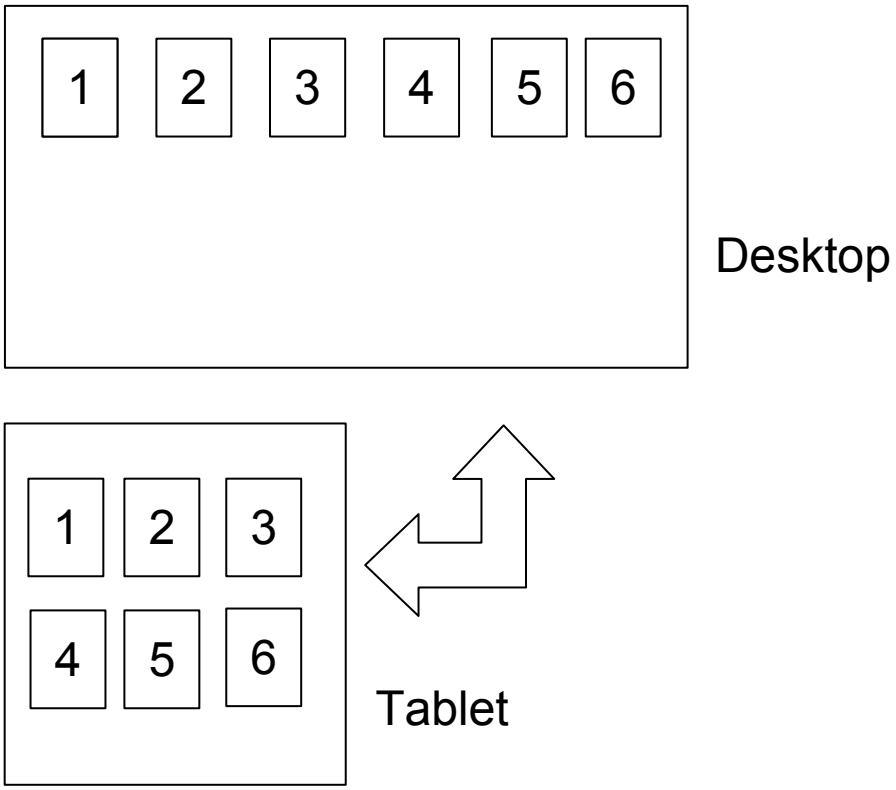


Figure 22 – Grid-layout columns wrap with device width

There a number of frameworks and approaches that will generate grid-based CSS. You can specify the number of columns you want and offsets between columns, and the framework will do the math and generate responsive CSS for your specified grid for you to apply. Some even allow you to specify the number of columns.

Listed below are a some popular grid-based frameworks, as published by Mashable.com:

- **Gridset** – gridsetapp.com
- **Frameless** – framelessgrid.com
- **Tiny Fluid Grid** – tinyfluidgrid.com
- **Gridpak** – gridpak.com
- **SimpleGrid** – simplegrid.info
- **Responsify** – responsify.it
- **Golden Grid System** – goldengridsystem.com

These grid frameworks are more useful for web-page designers. We are focusing on enterprise web applications for which we will be using UI widgets rendered with CSS, HTML, and JavaScript to handle user interaction and events. Just having a CSS grid mechanism by itself is not enough. Other elements like forms, tables, and navigation need to work in concert with the responsive grid layout.

Bootstrap - Turning Mere Developers into UI Geniuses

One popular open-source framework, Bootstrap.js, provides a responsive grid based upon Less Framework. It also provides many other features such as typography, buttons, and user-interface themes such as pills, navigations, alerts, dialog components, and forms, among others. All of these elements are styled

consistently, under the same responsive framework. This framework can make a developer look like a UI design genius.

Enough of the plug.... The power of this framework, besides a clean look and feel and a responsive, fluid layout, is that is provides forms, buttons, modal dialogs, navigation, and other UI elements and components that enterprise applications can use to implement a responsive user interface. Developers can put together a clean user interface without knowing CSS magic, even though most of the framework is implemented with CSS and behind-the-scenes JavaScript. Developers apply simple HTML with CSS class attributes.

Bootstrap employs a 12-column, fluid grid layout. Using `<div>` tags, class attributes are used to define a containment area that has rows and columns, or `spans` in Bootstrap lingo.

A common UI layout with a heading area for a navigation element, then a side bar and content area, is shown in figure 23:

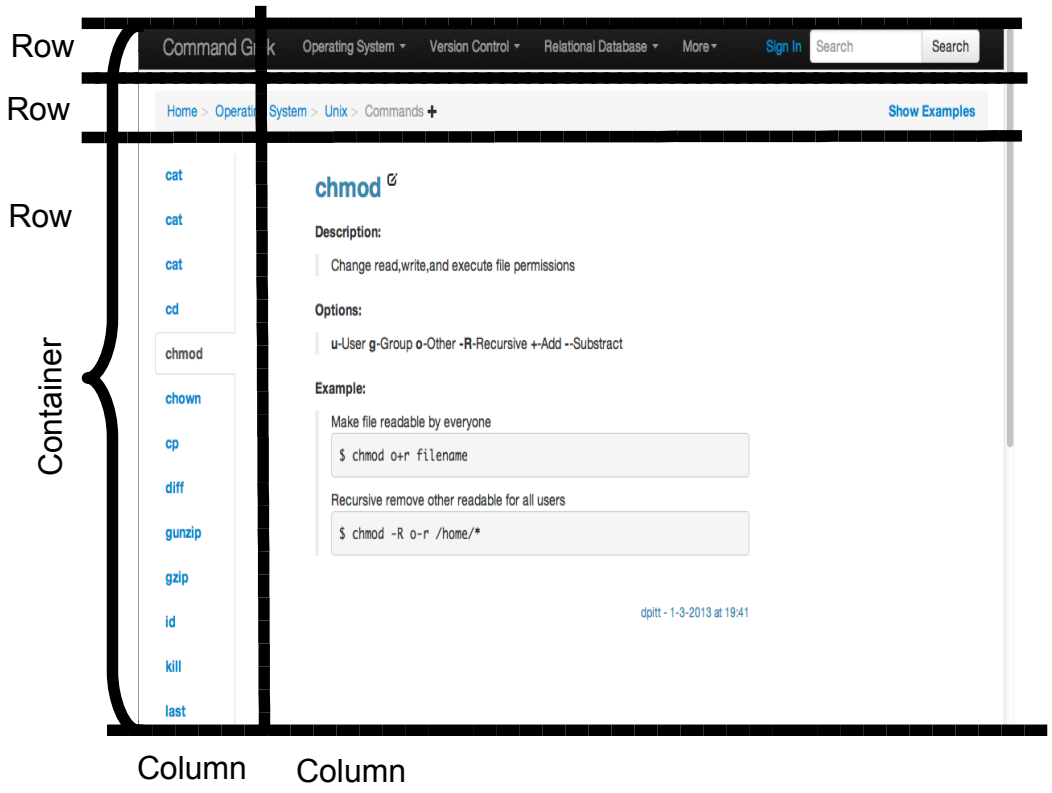


Figure 23 – A Bootstrap container's row/column layout

Another valuable feature worth pointing out, especially for enterprise applications, is the form support. If you ever tried to implement an input form using plain HTML `<forms>` then you know that trying to get fields to line up and look cohesive can be frustrating. But as you can see in the previous form example, the input form is lined up and looks cohesive. Bootstrap allows this to be defined using plain old HTML tags. The framework uses CSS attributes to communicate styling and layout.

Bootstrap's novel approach is that it is completely class-based. It also has many other components and features. Plugins are also available to let other GUI components be added. The bottom line is Bootstrap provides most of the elements you need for a clean, responsive user interface. One possible downside is the lack of table, tree, and grid components available out of the box. But there are many Bootstrap plug-in components available. With the popularity of Bootstrap, many more are sure to follow.

GUI Libraries

There are numerous libraries available that emulate user-interface controls beyond the HTML's standard form controls. This includes trees, controls, grids, tables, sliders, robust drop-down entry fields, date pickers, and more. The primary theme is to emulate widgets that are common to windowing-based operation systems. In

the enterprise, they are most likely emulating Microsoft Windows-type widget controls. Popular widget libraries include jQuery UI, DOJO, YUI, and there are many more available in the open-source world. These libraries provide a rich user experience but many of them are not responsive. To make them responsive, you will have to adapt them to a responsive CSS framework.

RESPONSIVE UI DESIGN DECISIONS

More than just CSS magic is required for applications to be responsive. Developers and user-interface designers need to implement user interfaces that depart from the traditional "windows" GUI metaphors we are used to. Why? These user interface designs attempt to emulate native operating systems for desktops. The desktop, or laptop for that matter, is no longer the only device on which your enterprise application will be viewed. So let us discuss some design decisions you can make that will help make your applications more responsive across many devices.

LIMIT DATA-ENTRY DIALOGS

The "windowing" metaphor that we are used to across major operating-system environments allows users to have many windows open at the same time. A common way to accept input from users has been to open a modal or dialog window, which essentially locked the input window until the user entered and applied data input or canceled the operation. One way to support a responsive layout and small screen sizes is to eliminate dialogs and use a direct-edit approach. Consider an edit option, as seen in figure 24, that shows how a user interface allows a timesheet to be edited.

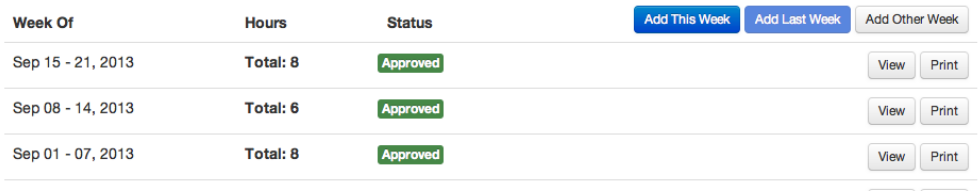


Figure 24 – Timesheet form

When the view/edit button is selected, instead of opening a dialog to view and edit a timesheet record, an edit form is exposed inline using a UI transition. When done, the edit form transitions away. This is shown in figure 25.

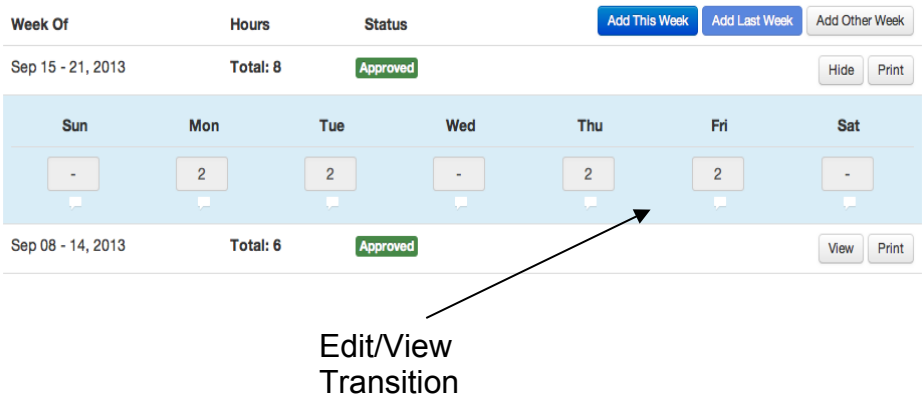


Figure 25 – Hitting the "View" button on a row, in this example of the view/edit transition, exposes a form inline that provides more details and editing capabilities

Dialogs, alerts, and short messages serve a purpose in situations that require user attention. Otherwise, try to minimize their usage as they can act wonky on different device screen sizes. If they are too big, they might not even work on some devices.

Other ways you can eliminate dialogs is to use UI form controls that integrate validation and selection mechanisms in the form-control entry field. Examples would be date/time pickers and range-based entry fields. Don't use or implement a dialog to assist the user in selecting a date. Make the date appear as an inline transition within an input form.

Apply Touchable Controls

There are many mobile touch-specific UI libraries. Many of these controls work equally well in desktop applications and work just fine with a keyboard and mouse. Feel free to apply them in enterprise desktop applications. Examples of using a touch control instead of a traditional input form are shown in figure 26.

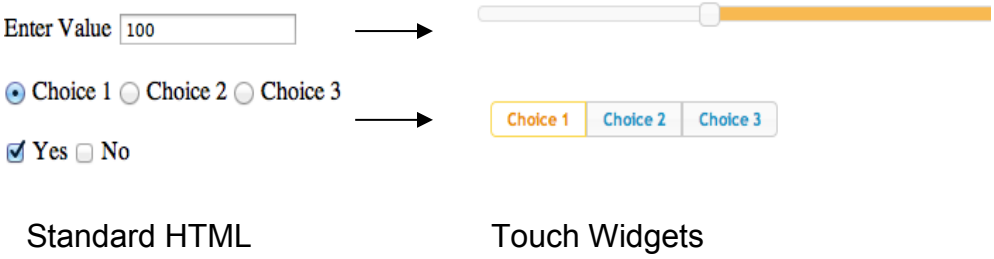


Figure 26 – Use touch widgets instead of standard HTML widgets

Slider controls can be used for numeric input. Toggle buttons can be used in the place of radio buttons and check boxes. Touch controls additionally provide a modern UI look to your application, which can help with user satisfaction.

Design Vertically

Moving or scrolling up and down, i.e. vertically, is more natural to the eye than horizontally, or left and right. Avoid designing UI screens with a lot of information going across the screen. If you do it, however, partition the information into panels, so they move naturally when the responsive layout stacks them vertically as device screen size changes.

Think about Navigation

Enterprise applications can offer much functionality with a variety of user interface screens that have lots of information to display and maintain. Implementing a simple way to access and navigate features or functions of the application makes the application easy to use. The right decisions on navigation can help with responsive design. For instance, a common navigation scheme is to put tabs across the top of the user interface. They can be thought of as menus in traditional windows applications, as shown in figure 27.

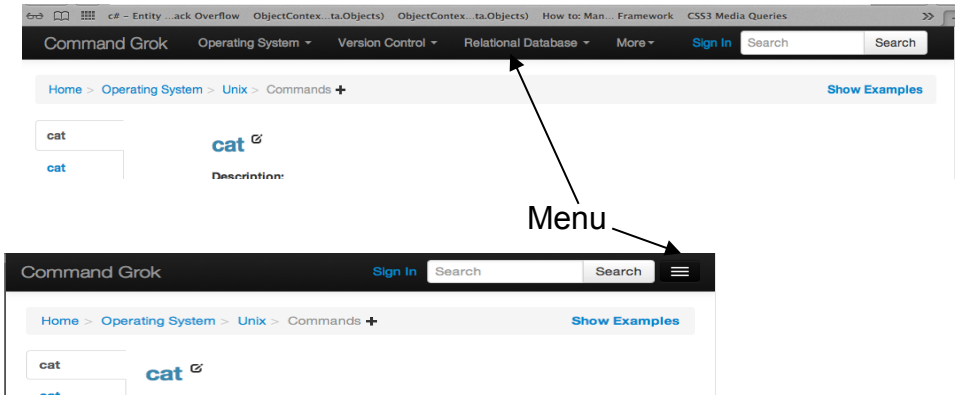


Figure 27 – An example of collapsing menu options

When the screen is resized, menu options collapse to an icon. Options and sub options are displayed horizontally when selected.

Using small, meaningful icons are a great way to conserve real estate and keep your interfaces clean. Use icons that commonly infer the operation that you are trying to communicate, as you can see in figure 28.

Figure 28 – An example of meaningful icons that replace menu options

This may be obvious, but don't underestimate the power of icons that can be used for commands and points that allows the use to transition between application screens. You don't need a graphic artist to obtain icons; many libraries provide them out of the box.

Also, tabs are your friend. Use tabs instead of popping up windows upon new windows to make it an easier experience for your user.

SUMMARY

This section introduced the concepts of responsive design for enterprise web applications. The goal is to make applications function well on devices other than desktop browsers. Here is a recap of what we covered in this chapter:

- Responsive design allows a single application to work across many devices.
- A common responsive-design technique is to divide UI elements into a grid.
- CSS3's media query is a key technology element in implementing a responsive design.
- Frameworks are available with ready-to-use responsive layouts.
- Bootstrap.js is a library that provides a responsive-layout mechanism, as well as UI controls and a standard look and feel that can make mere developers look like UI geniuses.
- Touch controls can be applied not only to make the an application touch-compatible, but to conserve UI screen space and add some pizazz to applications.
- Responsive design is not just applying a responsive CSS, but making smart UI-element choices.

References

- Mashable. http://mashable.com/2013/03/18/web-design-tools/
- Web Designer Wall. http://webdesignerwall.com/tutorials/css3-media-queries
- Smashing Magazine. http://www.smashingmagazine.com/responsive-web-design-guidelines-tutorials/#a2

Wrap Up

By this point, you should have an understanding of the JavaScript and HTML5 language features that are compelling to enterprise single page application development. You should also understand the general design decisions that you'll need to consider to ensure application responsiveness with various device resolutions.

Single page applications give enterprise the ability to provide its users with rich, responsive applications through the browser. In addition to the cost savings that the enterprise will celebrate by this choice, user experience can improve via the lack of reliance on plug-in technologies, improved speed, and implementation of responsive design.

But, as always, change is complicated – especially in the enterprise. Software architects and developers within these organizations must have the right toolset to fall back on, as well as the correct combinations of experience and knowledge to successfully guide the enterprise through this paradigm shift.

A thorough understanding of the topics discussed in this book is necessary to build solid web SPA applications. If some concepts are still fuzzy, I recommend that you play around with and modify some of the samples to see if that helps your understanding. I hope that this mini-book will help you to avoid the pitfalls in Single Page Application development.

ABOUT THE AUTHOR

David Pitt is a Sr. Solutions Architect and Managing Partner of Keyhole Software with nearly 25 years IT experience. For the last fifteen years, David has helped enterprise IT departments adopt object technology, leading and mentoring development teams focused on Java (JEE) and .NET (C#) technologies. Most recently, David has been helping organizations to make the architecture shift to JavaScript/HTML5 and use best practices to create rich client and single page applications. Document editing by Lauren Fournier.

Keyhole Software is a Midwest-based software development and consulting firm founded on the principle of delivering quality solutions through a talented technical team. Core capabilities include custom application development with Java, JavaScript/HTML5, and .NET technologies, technical education and mentoring, project recovery, and enterprise application enhancement. The Keyhole team loves the technologies behind the solutions - Kansas City, St. Louis, Chicago.