Sofware Development with Scripting Languages: Browser Side Scripting

Onur Tolga Şehitoğlu

Computer Engineering,METU

2 January 2015

Javascript Syntax

- Standard name is ECMAscript
- C alike
- Semicolon ; is optional for multiple lines
- Interpreter is embedded on browser
- Code can be embedded in HTML with `<script>` tag.
- Browsers provide a Javascript console

```
<head>
<!-- include a script file from URL -->
<script src="/libs/jquery/1.7.1/jquery.min.js">
</script>
</head>
<body>
<script>
var msg = 'hello'; var target=' world'
alert(msg + target) {
}
</script>
```

Values and Types

- Very minimal set of types: `string`, `number`, `object`
- Literals `'Hello','world'`, `3.454`, `7`, `{name:'value',id:5}`
- Type conversion is done as long as possible

```
a= "1"+4 ; a++ ; // a is 15
```

- Objects are created on the fly, in a way similar to Python dictionaries. Object method can access instance with `this`.

```
var counter = { val: 0 ,
                get: function () {return this.val},
                incr: function() { this.val++}}

counter.get()
counter.incr()
```

- No type enforcement on objects.

Objects and Arrays

- Object members `student = { name: 'ali' }` can be accessed in two ways:
  - `student.name`
  - `student['name']`
- `Array` is a builtin object type that implements integer indexed elements.

  ```
  a = [7,18,23] ; a[0] = a[1] + a[2]
  ```
- Some array methods work in place, modifies array:

  ```
  a.sort() ; a.reverse()
  a.push('hello') ; a.pop() ; a.unshift('hello') ; a.shift()
  ```
- Some returns new values:

  ```
  a.indexOf(3) ; a.join(":") ; a.slice(2,4); a.length
  a.filter(boolfunc) ; a.map(func); a.concat([3,4,5]) ;
  ```

Manipulating Objects

- Arbitrary members/methods can be added by assignments:

```
student = {} ; student.name='Ali'
student['no'] = 444 // { name: 'Ali', no: 444 }
```

- Members can be deleted:

```
delete student.name // { no: 444}
```

- Array elements can be deleted using `splice(start,num)`:

```
a.splice(5,2)  // delete a[5], a[6]
```

- Assignment is by reference. Copy can be done member by member.
- Testing for a member:

```
if ( student.hasOwnProperty('name')) or simply:
if (student.name)
```

Operators

- Very similar to C. Most operators including increments, arithmetic assignments and conditional expression are works similar to C.
- '+' is concatenation unless both operands are numbers
- `/` is real division. `Math.floor/celing/round()`
- Numbers and strings are also objects. They have some conversion operators
- `Math` object contains more arithmetic functions.
- `/regexp/` creates a regular expression object. `/^[0-9]+$/.test(mystring)` returns `true` or `false`, match result of regular expression

Conditionals and Loops

Similar to C:

- `if (cond) else if (cond2) else (cond3);`
- `switch (val) { case num1:...; break; case "str1":...;...}`
- `while (cond) { ... }`
- `do { ... } while (cond)`
- `for (i=0; i < n; i++) { ... }`
- Definite iteration on objects (`a=[1,4,6,10]; b={x:3,y:4,z:5}`)

```
for (i in a) { ...  a[i] ...} //i=0,1,2,3
for (i in b) { ...  a[i] ...} //i='x','y','z'
```

Exceptions

```
try {
    ...
} catch (excp) {
    ...
} finally {
    ... // optional, executed always
}
```

- `throw` statement can be used to raise an exception
- If not handled exceptions stop current script execution

Defining Functions

```
function f(param1, param2, param3)  {
    var x = ...

    if (param3 == undefined) param3 = 'default value'
    ...
   return val
}
// lambda like definition possible
a = [1,2,3,4]
b = a.map(function (x) { return x*x }    // b= [1,4,9,16]
// following two are equivalent:
function add(x,y) { return x+y }
var add = function (x,y) { return x+y }
```

Variable Scope and Lifetime

- Variables are global by default, regardless of their position of initialization
- `var` is used to create a variable in local scope.

```
var x; var y = 5 .
```

```
for (var i = 0; i < n ; i++) ...
```

- Using always `var` is a good practice. Otherwise using global scope causes confusions, especially in recursion.
- Variables used by an object can be hidden using scope:

```
function counter() {
    var val = 0; // hidden inside the function
    return { incr: function () { val++},
             get: function () { return val} }
}
a=counter()
a.incr() ;     console.log(a.get())
```

- Similar closures can be defined.

```
function multiplyby(x) {
    return function (y) {
        return x*y;
    }
}
twice = multiplyby(2);
twice(4); // will return 8
function sortby(field) {
        function compare(a,b) {
                return (a[field] < b[field])? -1 : 1
        }
        return function (arr) {
                arr.sort(compare)
        }
}
rlist = [{name:'z',no:43},{name:'a',no:31},{name:'c',no:11}]
namesort = sortby('name')  // a new function sorts by 'name'
nosort = sortby('no')      // a new function sorts by 'no'
namesort(rlist)
nosort(rlist)
```

Assignment Semantics

- Share semantics
- Assignment copies reference, not data
- Object assignment creates two variables denoting same object
- Primitive values copied, objects shared (like Java)
- Parameters pass by value for primitives, reference for objects
- Copying requires special copy functions

- Objects can be created on the fly by `{...}`
- `this` denotes the current object inside the function members.

```
{c:0, incr: function() { this.c++}}
```

- Classes can be created by class functions updating members of `this` and returning `this`.
- Updates on `this` updates a function prototype and when `return this` is executed this prototype instance is returned.
- New instances can be created as `new classfunction()`.
- Define a binary search tree class `BSTree` such that:

```
bst = new BSTree()
f=[["orange","orange"],["apple","red"],["banana","yellow"],
   ["watermelon","green"],["strawberry","pink"]])
for (i in f)    bst.insert(f[i][0], f[i][1])
color = bst.get("apple")
bst.forEach(function (n) { console.log(n.key+":"+n.value) })
```

```
function BSTree() {
    var root = undefined;   // local scope, private
    this.insert = function (key, value) {
        if (root == undefined) {
            root = { node: {key: key , value: value},
                     left: new BSTree(), right: new BSTree() }
        } else if ( key < root.node.key )
            root.left.insert(key, value)
        else if ( key > root.node.key )
            root.right.insert(key, value)
        else root.node.value = value;                        }
    this.get = function (key) {
        if (root == undefined) return undefined
        else if (key < root.node.key) return root.left.get(key)
        else if (key > root.node.key) return root.right.get(key)
        else return root.node.value                          }
    this.forEach = function (func) {
        if (root == undefined)   return
        root.left.forEach(func)
        func(root.node)      // apply function to node content
        root.right.forEach(func)                             }
}
```

Function Prototype

- Repeating function definitions in each instance is inefficient.
- Javascript binding searches `functionname.prototype` object for missing members for the object. So that a single copy per function/class maintained

```
function Counter() { this.value = 0 }

Counter.prototype.get = function () { return this.value }
Counter.prototype.incr = function () { this.value++ }

c = new Counter() ; d = new Counter() ; e = new Counter()
c.incr()
console.log(c.get())
```

- `c`,`d`,`e` shares same prototype object of `Counter`.
- prototype modifications after construction of an object affects the object

BSTree class rewritten with prototype

```
function BSTree() {
    this.node = undefined;
}
BSTree.prototype.insert = function (key, value) {
    if (this.node == undefined) {
        this.node = {key: key , value: value}
        this.left = new BSTree(); this.right= new BSTree()
    } else if ( key < this.node.key )
        this.left.insert(key, value)
    else if ( key > this.node.key )
        this.right.insert(key, value)
    else this.node.value = value;                    }
BSTree.prototype.get = function (key) {
    if (this.node == undefined) return undefined
    else if (key < this.node.key) return this.left.get(key)
    else if (key > this.node.key) return this.right.get(key)
    else return this.node.value                      }
BSTree.prototype.forEach = function (func) {
    if (this.node == undefined)   return
    this.left.forEach(func)
    func(this.node)      // apply function to node content
    this.right.forEach(func)                         }
```

Inheritance

- Prototypes can be chained for inheritance.
  `Object.create(protobject)` Creates an object with prototype `protobject`. If you set prototype of `B` as an object with same prototype with `A`, bindings of `B` will be chained to `A`, results in inheritance.

```
function Shape(x, y) { this.x = x ; this.y = y}
Shape.prototype.move = function (x,y) {
    this.draw('background'); this.x = x; this.y = y;
    this.draw('foreground') }

function Circle(x, y, r) {
    Shape.apply(this,[x,y]); this.r = r   // Apply Shape constructor to this
}
Circle.prototype = Object.create(Shape.prototype)   // chain prototypes
Circle.prototype.constructor = Circle
Circle.prototype.draw = function (color) {
  console.log('drawing␣circle␣at', this.x, this.y, this.r, 'in', color)}

c = new Circle(10,10,5); c.move(30,20)
```

- Since EcmaScript 6 (2016) `class` definitions are implemented as syntactic sugar (maps code to old style prototype definitions)

```
class Shape {
    constructor(x,y) { this.x = x; this.y = y}
    move(x,y) {  this.draw('background'); this.x = x; this.y =y ;
                 this.draw('foreground')
    }
}
class Circle extends Shape {
    constructor(x,y,r) { super(x,y); this.r = r}
    draw(color) {
      console.log('drawing␣circle␣at', this.x, this.y, this.r, 'in', color)
    }
}
```

BSTree class with new syntax

```
class BSTree {
    constructor() { this.node = undefined}
    insert(key, value) {
        if (this.node == undefined) {
            this.node = {key: key , value: value}
            this.left = new BSTree(); this.right= new BSTree()
        } else if ( key < this.node.key ) this.left.insert(key, value)
        else if ( key > this.node.key ) this.right.insert(key, value)
        else this.node.value = value;
    }
    get (key) {
        if (this.node == undefined) return undefined
        else if (key < this.node.key) return this.left.get(key)
        else if (key > this.node.key) return this.right.get(key)
        else return this.node.value
    }
    forEach (func) {
        if (this.node == undefined)   return
        this.left.forEach(func)
        func(this.node)      // apply function to node content
        this.right.forEach(func)
    }
}
```

Browser Objects

- `window`: an interface to browser. Loading a new page, etc.
- `document`: an interface to current document. Document Object Model members.
- `console`: Browser's Javascript console. `console.log(message)` can be used to write debugging outputs on console.
- `document` can be used to access and update current web page content dynamically.

Most important members of `document` are:

- `document.head` : Header, `<head>` element of current page.
- `document.body` : Body, `<body>` element
- `document.forms` : list of `<form>` elements in this document
- `document.title` : Page title.
- `document.cookie`: Cookies of current page.
- `document.links` : List of links of the document (as `<a>` elements).
- `document.getElementById`: searches and returns the element with given `id=...` attribute in the document.
- `document.getElementsByClassName` `.getElementsByTagName`: searches and returns all elements with given `class=...` attribute and given HTML tag respectively.
- `document.querySelector` selects elements based on CSS selectors.
  `document.querySelector('div.student > ul')` selects `ul` elements that are immediate child of a `student` class `div`.

An HTML document contains nested elements. Top element is `<html>`. DOM element is a tree like representation of document structure. A DOM node named `el` has the following members:

- `el.children` : List of child HTML elements nested in this element. All `<...></...>` elements that are direct child of `el`.
- `el.childNodes`: List of child nodes including HTML elements and text.
- `el.nodeType`: Type of the node, 1 for element, 2 for attribute, 3 for text and 8 for comment node.
- `el.getAttribute(attr)`: returns attribute value for a given attribute. For example if an image element, `el.getAttribute('src')` will give image URL.
- `el.id`: Element id by `id="..."` attribute.
- `el.className`: element class by `class="..."` attribute

- `el.innerHTML`: HTML text content enclosed by element. Can be updated.
- `el.outerHTML`: HTML text content including the element. Can be updated. i.e. assigning it to `''` deletes the element.
- `el.attributes`: Object to attributes. Attributes can be modified, inserted, removed.
- `el.setAttribute('border','1')`: sets an attribute of the element
- `el.removeAttribute('border')`: remove an attribute of the element
- `el.insertBefore(newel, child)`: inserts an element as a child, before the child element.

  `el.inserBefore(newel, el.children[0])` inserts it as the first child.
- `el.appendChild(newel)`: add element as the last child of the element.

Example: Add a `<ul>` element to document at the end of the document body:

```
var li1 = document.createElement("LI")
var li2 = document.createElement("LI")
var li3 = document.createElement("LI")
li1.appendChild(document.createTextNode("apple"))
li2.appendChild(document.createTextNode("banana"))
li3.appendChild(document.createTextNode("orange"))
var ul = document.createElement("UL")
ul.appendChild(li1)
ul.appendChild(li2)
ul.appendChild(li2)
document.body.appendChild(ul)
```

Example: Recursively traverse all elements and log the tag name and attributes:

```
function traverse(el) {
    var str = el.tagName
    var attr = []
    if (el.attributes)
        for (var i=0 ; i < el.attributes.length; i++)
            attr.push(el.attributes[i].name + ":" +
                    el.attributes[i].value)
    console.log(str + "[" + attr.join(",") + "]")
    // recurse to children
    if (el.children)
        for (var i=0; i < el.children.length ; i++)
            traverse(el.children[i])
}
traverse(document.body)
```

DOM Actions

- HTML elements can have mouse, keyboard or other browsers events handler functions.
- Either defined as HTML attribute or set later on DOM:

```
<img .. onclick="JS code here">
```

```
el.addEventListener("click" , function (ev) { ... })
```

- Multiple event listeners can be added and removed on DOM:

```
el.removeEventListener("click" , functionname)
```

- for event type X, attribute name is onX
  - `click`: mouse click
  - `dblclick`: mouse double click
  - `mouseover`: mouse pointer is over
  - `keypress`: keyboard key pressed (event object paramater)
  - `beforeload`: when object starts to be loaded
  - `load`: when object is loaded

- Events for form elements:
  - `change`: when form element content changed
  - `select`: when form element is selected
  - `focus`: when form element gets focus
  - `blur`: when form element loses focus
  - `submit`: when form is submitted. Function should return `true` if default action (submit) should follow. Calling `el.submit()` will explicitly submit the form data.
- Form data can be validated through these events. Form elements give access to input elements with their names.
  `document.forms[0].surname` gives element:
  `<input ... name="surname"...>` of the first form.
- Keyboard events get an event object containing
  `keyCode`, `ctrlKey`, `shiftKey`, `AltKey` fields.
- Mouse events get an event object containing
  `button`, `clientX`, `clientY` fields.

Form data validation example:

```
<script>
function checkname(fname) {
        var n = document.getElementById('myform')
        var m = document.getElementById(fname + 'msg')
        if ( /^[a-z A-Z]+$/.test(n[fname].value)) {
                m.textContent = "[OK]"; return true
        } else {
                m.textContent = "[INVALID]";  return  false
        } }
function checkage(fname) {
        var n = document.getElementById('myform')
        var m = document.getElementById(fname + 'msg')
        if ( n[fname].value > 0 && n[fname].value < 200) {
                m.textContent = "[OK]"; return true
        } else {
                m.textContent = "[INVALID, 1-200]"; return  false
        } }
function checkform() {
        return (checkname('Name') && checkname('Surname') &&
                checkAge('Age')) }
</script>
```

Continued:

```
...
<form id="myform" action="..." method="post"
       onsubmit="return␣checkform()">
Name: <input type="text" name="Name"
             onblur="checkname('Name')"/>
      <span id="Namemsg"></span>
<br/>
Surname: <input type="text" name="Surname"
                onblur="checkname('Surname')"/>
         <span id="Surnamemsg"></span>
<br/>
Age: <input type="text" name="Age"
            onblur="checkage('Age')"/>
     <span id="Agemsg"></span>
<br/>
<input type="submit" name="submit" value="Send" >
</form>
...
```

Execution Model

- Javascript executes in a single thread/process (thread per browser tab/thread per nodejs connection)
- Main interpreter code is an Event Loop where a queue of messages are processed one message at a time.
- Each message in the queue is a Javascript function pointer. A new stack frame is created and function is called.
- Browser pushes scripts in HTML code into the message queue on page load.
- Each message is run to the completion. Then next message is processed.
- Browser events, timed events and callbacks push new messages.

- Execution is never preempted. Infinite loops and long running functions should be avoided.
- Most I/O tasks works with call backs
- Callback hell. Example: chain 3 functions with callbacks and possible error values:

```
getDocument(docA, vA => {
    if (vA) {
        getDocument(docB, vB => {
            if (vB) {
                getDocument(docC, vC => {
                    if (vC) {
                        // do something usefull with  vA + vB + vC
                    } else { // handle error
                    }
                })
            } else { // handle error
            }})
    } else { // handle error
    }})
```

- Run to completion is tricky:

```
function getsum(userinput) {
    var sum = 0
    for (c of userinput) { // userinput is a list of items user selected
        makedbquery(c, v => {
            sum += v
        })}
    return sum
}
console.log(getsum(['a','b','c']); // WON'T WORK!!!
```

- Solution: convert to callback style with recursion

```
function getsum2(userinput, cb) {
    if (userinput.length == 0) { cb(0); }
    else {
        makedbquery(userinput[0], v => {
            getsum2(userinput.slice(1), sum => {
                cb(v + sum)
            })
        })
    }
}
```

Promises

- Promise is a class defining an asynchronous event returning a value in future. Initially in pending state. When value is available, calls the resolving action.

```
function findPerson(name) {
    return new Promise(resolve => {
        getidfromname(name, v => resolve(v))
     })
}
function getList(id) {
    return new Promise(resolve => {
        getclistfromid(id, v => resolve(v))
    })
}
findPerson('onursehitoglu')
    .then(getList)
    .then(lst => { console.log(lst) })
    .catch(err => { console.log(err)})
```

- Promises give more clear code

- Independent promises

```
function getsum(userinput) {
    return new Promise(resolve => {
        // assume following give a list of promises
        var plist = userinput.map(makedbquery)
        Promise.all(plist).then( vlist => {
            // all promises resolved and gave a value list
            // resolve as the sum
            resolve(vlist.reduce( (v,r) => { return v+r}))
        })
    })
}
```

async/await

- async/await keywords make asynchronous programming much more easier.
- define all asynchronous functions with async prefix
- call all asynchrous functions with await prefix
- rest of the function is put into a message (callback) and inserted in queue when asynchronous function completes.
- Javascript make all necessary conversions.

```
async function wait(t) {
    return new Promise(resolve => {
        setTimeout(resolve, t) })
}
main = async () => {
    await wait(1000);
    console.log('hello')
    await wait(1000);
    console.log('world')
}
main()
```

- structureasync/await works as a syntactic sugar. Simplified description:

  `async function f(...) { return expr}` is equivalent to:

  ```
  function f(...) { return new Promise(res => res(expr)) }
  ```

  `{let v = await f(...); st1 ; st2}` is equivalent to:

  ```
  {f(...).then( ret => {let v = ret; st1; st2})}
  ```

- Previous slide without await

```
main = () => {
    wait(1000)
        .then(() => {
            console.log('hello')
            wait(1000)
                .then(() => {console.log('world')})
        })
}
```

- `getsum` async/await version

```
async function getsum(userinput) {
    var sum = 0
    for (c of userinput) { // userinput is a list of items user selected
        sum += await makedbquery(c)
    }
    return sum
}
```

AJAX

- Asynchronous Javascript And XML
- Browser script can make HTTP requests to web server to load/update data dynamically.
- `XMLHttpRequest` object does the job.
- Works in synchronous and asynchronous modes. Former should be avoided, blocks until request completed and result is taken. Asynchronous mode works with a callback.
- `open(method,path, async)` method sets up the connection object.
- `send()` will initiate the request
- On success `response` contains the server response.
- If server sends XML document `responseXML` contains DOM of it.

- ```
  function getFlights(depcity) {
      var req = new XMLHttpRequest()
      // assume getflights/city gives direct flights
      req.open('POST','/getflights/'+depcity, false) // syncronous
      req.send('id=all&nres=100')   // blocks until response ready
      if (req.status == 200) { // success
          return req.responseXML    // XML DOM of result
      } else
          return undefined
  }
  ```
- Server can return an XML DOM value of direct flights from city:

  ```
  <xml>
  <flights><flight no="LH1231" dep="ESB" dest="MUN">Munih</flight>
           <flight no="TK5434" dep="ESB" dest="ERC">Lefkosa</flight>
           <flight no="TK22" dep="ESB" dest="BRL">Berlin</flight>
  </flights>
  </xml>
  ```
- This DOM can be processed similar to HTML DOM.
- Alternatively XPath can be used

  ```
  doc.evaluate('//flight[@no="LH1231"]', doc)
  ```
  ```

Asynchronous operation handled by `onreadystatechange` event, called on different stages. `readyState == 4` when connection is ended.

```
function getFlights(depcity, updatefunc) {
    var req = new XMLHttpRequest()
    req.open('POST','/getflights/'+depcity, true) // asyncronous
    req.onreadystatechange = function () {
        if (req.readyState != 4) return; // intermediate states
        if (req.status == 200) { // success
            updatefunc(req.responsXML)
        } else alert('request failed:' + req.status)         }
    req.send('id=all&nres=10')   // if POST, send form data here
}// no value returned. function will be called when ready
function updflight(xmldom) {   // update <SELECT> with flights
    var flghts = xmldom.getElementsByTagName('flight')
    var selbox = document.getElementById('flightselect')
    selbox.innerHTML = ""
    for (var i in flghts) {
        var fno = flghts[i].getAttribute('no')
        var fdest = flghts[i].getAttribute('dest')
        var opt = document.createElement("OPTION")
        opt.setAttribute('fno', fno)
        opt.textContent = flghts[i].textContent
    }           }
getFligths('ankara',updflight)
```

JSON

- JavaScript Object Notation
- Shorter and easier then XML for exchanging data. Similar to JavaScript.

```
user = '{"name":"Onur",
         "courses": [{"code":5710498, "name":"ceng498"},
                     {"code":5710536, "name":"ceng536"}],
         "room":"B-209"}'
// eval('userobj = ' + user)   works but insecure!!!! Avoid it....
userobj = JSON(user)          // Javascript object
userstring = JSON(userobj)    // back to first string
```

- All languages have libraries for JSON generation and parsing.

```
import json
user = '{"name":"Onur",
         "courses": [{"code":5710498, "name":"ceng498"},
                     {"code":5710536, "name":"ceng536"}],
     "room":"B-209"}'
userobj = json.loads(user)   # python dictionary
userstring = json.dumps(userobj)   # back to string
```

- JSON is denoted by `application/json` in `Content-type` header.

```
Content-type: application/json

{"result":"success",
 "flights":[
    {"no":"LH1231","dep":"ESB","dest":"MUN","city":"Munih"},
    {"no":"TK5434","dep":"ESB","dest":"ERC","city":"Lefkosa"},
    {"no":"TK2222","dep":"ESB","dest":"BRL","city":"Berlin"}
  ]
}
```

- Server side can easily convert their objects into JSON strings.
- `XMLHttpRequest` object `response` member can be parsed directly into Javascript object.

```
result = JSON.parse(x.response)
```

- Much more easier than traversing XML DOM.

fetch

- Promise based `fetch` API is more convenient than callbacks. Supported after 2015.
- `fetch(url, options)`

```
async function getFlights(depcity, updatefunc) {
    var resp = await fetch('/getflights/'+depcity,
            { method: 'POST', body: 'id=all&nres=10'})
    if (resp.ok) {
            updatefunc(resp.responseXML)
    } else { alert('request failed:' + resp.status)      }
}
```

JQuery

- A Javascript library for easier Javascript development
  - Shorter and easy to use CSS selectors `$(".id")`
  - Browser version abstraction. Same code support multiple browsers
  - Animations with stylesheet
  - Easy document manipulation and AJAX
  - Ready to use interactive widgets through JQuery-UI
  - Many contributed applications and widgets sets.
- You need to download a version from `http://jquery.com` and include in web page for example inside of `<head></head>`:

```
<script src="/jquery/jquery.min.js"></script>"
```

- or hosted by content providers:

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js">

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js">
```

Selectors

- `$` is a shorthand for `jQuery`.
- Selector `$()` is entry point of JQuery.
- Selector select a set of DOM elements. Returns a result object.

  `$("tagname")`, `$("#id")`, `$(".class")`, `$("parent␣>␣immchild")`

  `$("parent␣descend")` `$("[attr]")`, `$("[attr=value]")`

  `$("[attr!=notvalue]")` `$("[attr^=prefix]")`, `$("[attr$=suffix]")`

  `$("p,span,div")` (or conn.),
- There are more in selector syntax and selectors can be combined:

  `$("table.plist␣img[src␣$=␣jpg]")` selects all `img` tags with `src` attribute ends with `jpg` and that are contained in tables with class `plist`.

Manipulation

- Matched elements inner content can be changed. Operation is applied on all elements:
  - `.html(content)` replaces inner HTML.
  - `.text(content)` replaces inner text, HTML escaped.
  - `.append(content)` add content as the last item inside
  - `.before(content)` add content as the first item inside
  - `.wrapInner(content)` enclose inner elements with content
- Matched elements surroundings can be changed:
  - `.wrap(content)` elements enclosed in content
  - `.after(content)` content inserted after elements
  - `.before(content)` content inserted before elements
- Elements can be removed or hidden
  - `.remove()` remove elements
  - `.hide()`, `show()`, `toggle()` hide, show, toggle visibility
- Attributes:
  - `.addClass(cls)`, `removeClass(cls)`
  - `.attr(attr,val)`, `.removeAttr(attr)` set/remove elements attribute

Traversal

- `.find(sel)` apply filter on currenet elements
- `.each(func)` apply function to all elements

  ```
  $("*").each(function (i,e){console.log(i+el.tagName}))
  ```

  will print all element tags in document.
- `.next()` give sibling following each element
- `.prev()` give sibling preceding each element
- `.children()` give children of each element
- `.parent()` give parent of each element

Events

- General form of setting an event handler:

  `.on("event", data, func (ev) {...})`

  `data` is passed as `ev.data` to handler.

  `.on("event", func (ev) {...})` is no data version

  `event` can be `"click"`, `"blur"`, `"focus"`,`"submit"`, `"select"`,...
- All events are also provided as functions:

  `.click(func)`, `.blur(funct)`, `.submit(funct)`,...
- Cancelling event handlers: `.off()`, all handlers.

  `.off("click")`, all click handlers.

  `.off("click",myfunc)` only `myfunc` handler.
- `.one()` binds a one time only event handler.

- `$.post()` and `$.get()` can be used to make AJAX requests directly.
- `$.post(url, data, succfunc, datatype)` is the full usage:
  - `data` is either a query string or an object. If object jQuery converts it into a query string using `$.param()`
  - `succfunc` is a function to be called on success. It gets server response as parameter.
  - `datatype` is the type of serer response, xml,json,...
  - `$.post(...).fail(func(){..})` defines failure functions
- Form data can be converted on POST query string:

  ```
  $('form').serialize()
  ```
- For detailed functionality, use `$.ajax()`.

Flights example with jQuery:

```
function getFlights(depcity, updatefunc) {
    $.get('/getflights/'+depcity, updflight)
     .fail(function() { alert('error')})
    // assume JSON response, jQuery automatically parse
    // and send object to success function
}

function updflight(resp) {   // assume json respons
    $('#flightselect').html('')          // clear
    for (var i in resp.flights) {
        $('#flightselect').append('<option></option')
        // :last() select last child. methods can be cascaded
        $('#flightselect>:last()').attr('fno',resp.flights[i].fno)
                                  .text(resp.flights[i].city)
    }
}
getFligths('ankara',updflight)
```

- Implement View on browser side
- Control on browser side call server side Control via AJAX
- Server side model updates are loaded in view on browser side.
- All server responses converted into XML or more practically JSON
- Javascript get responses and update web page.
- More dynamism achieved in HTML5 through web sockets.

Cross-Origin Resource Sharing (CORS)

- Web applications may need to connect to web services in other domains.
- Browsers default information flow allow HTML based links, embedded images and form posts direct to cross-origin (other than the current domain of HTML page) requests.
- CSRF protects POST requests from other domains.
- When there is a an AJAX request with custom headers or non standard (different content type) POST request, same origin policy forbids making a request to another domain.
- Browser declare their origin web domain with `Origin` header and web services can pick which domains to serve on `Access-Control-Allow-Origin` header.

- Browser side decision can be written as:

method ← HTTP method
targetorigin ← main part of target URL
origin ← main part of the URL of the running script
headers ← request headers
if origdomain=domain **then**
    Do actual request
else if *headers* $\not\subset$ standardheaders *or*
        (*method* =POST *and* contenttype is not standard) **then**
    make a GET or OPTIONS request to URL
    send `Origin:` *origin* header
    if Response is success with a matching `Access-Control-Allow-Origin` header **then**
        Do actual request
    else
        return Error
    end if
end if

- Web services expecting non standard requests (JSON,XML, etc) from other domains should implement `OPTIONS` method and send appropriate headers.
- Server sends either `Access-Control-Allow-Origin: *` or `Access-Control-Allow-Origin: OriginHeaderinRequest`
- This negotiation defines the authorization for the actual request. Server can also restrict:
  - Valid HTTP methods `Access-Control-Allow-Methods`
  - Valid HTTP headers `Access-Control-Allow-Headers`
  - Number of seconds the reply can be cached so that the headers will be valid `Access-Control-Max-Age`
  - If cookies and credentials should be sent with request `Access-Control-Allow-Credentials`
- `Access-Control-Allow-Origin: *` matches all origins.

Server Push

- One way to access newly available data on server is polling server continously.
- Polling interval determines how fresh data is. Small intervals end up in server load long intervals reduce responsiveness.
- Generally one of 3 methods are used to solve this problem
  - Long polling Server does not respond immediately and keep connection open until data is available. Client repeatedly polls on connection break and/or new data.
  - Server Sent Events `EventSource(URL)` request on client is responded by server with `Content-Type: text/event-stream`. Connection is kept open and each time server writes on connection, client gets a message event.
  - WebSocket A bidirectional tunnel between server and client.
- First two works on HTTP protocol. Websockets use HTTP to establish the connection, than switches to a special protocol.

Websockets

- Client sends:
  `Upgrade: websocket` and `Connection: Upgrade` headers to HTTP server to establish a websocket connection.
- Server responds:
  `HTTP/1.1 101 Switching Protocols`
  and repeats the two headers above in the respond.
- After this handshake, communication switches to a full duplex, binary frame based protocol. Each frame starts with the payload length in two or more bytes, followed by the payload. A message can be sent in a single frame or sent in multiple frames.
- Websockets are not subject to same origin policy so server check `Origin` header and authorize it when establishing the connection.

- Browser side works on a `WebSocket` object and it is event based:

```
const socket = new WebSocket('ws://example.com/ws');

socket.onopen = function () {....}
socket.onmessage = function (event) { var message =  event.data ; ...}
socket.onclose = function (event) {....}
socket.onerror = function (event) {....}
```

- Browser script can call `socket.send("....")` to send a message to established socket.
- URL protocol `'ws://'` is used for websocket connections. Secure version is `'wss://'`.

Websocket Authentication

- Websocket protocol does not have headers. The last time server gets an HTTP header is during the connection establishment (`Upgrade: websocket` header). The connection can be rejected based on session or CSRF cookies.
- Other means of authencitation should be handled in custom messages in websockets. i.e. Send authentication required as JSON message, client sends credentials in a JSON message etc.
- Similarly session token can be sent in each client message to get a stronger security.

Server Side

- Websocket is not HTTP so application firewalls and proxies can block websocket connections if not configured otherwise.
- For Python there are couple of alternatives:
  - `websockets` library. A Python library implementing websockets.
  - `Tornado`. A web application framework that serves websocket connections and works as a WSGI compatible server.
  - `Django Channels`. Some extensions and extra servers to add websocket support in Django.