Intro To Web Services

JavaScript Object Notation (JSON)

JSON is a lightweight data-interchange format, based upon the JavaScript language.

We are interested in JSON because it is the format we are going to use to communicate through our Web Service API.

Something that makes JSON appealing to the JavaScript developer is that JSON evaluates directly to JavaScript objects!

It is built upon two structures:

- Collection of name/value pairs; basically a JavaScript object.
- An ordered list of values; basically an array

Here is some example JSON...

```
{
      “players” : [
            {
                  “name” : “John Doe”,
                  “nickname” : “Johnny  Boy”
            },
            {
                  “name” : “Jane Doe”,
                  “nickname” : “Plain Jane”
            }
      ]
}
```

Here is how you go about transforming between JSON and JavaScript object...

```
var jsonTxt = '{ "name" : "John Doe" }';

var jsonObj = eval('(' + jsonTxt + ')');  // NEVER, EVER DO THIS!!
var jsonObj2 = JSON.parse(jsonTxt);

console.log(jsonObj);
console.log(jsonObj2);

var jsonTxt2 = JSON.stringify(jsonObj2);
console.log(jsonTxt2);
```

HTTP Protocol/Verbs

When a request is made via HTTP, a verb is associated with it, and these verbs should be used! The verbs are (the ones we care about):

- GET : Request to retrieve information
- PUT : Request to store or replace information at a specific URI
- POST : Request to add or update information
- DELETE : Request to remove information

Safe Methods

A method that doesn't change state. Information retrieval is considered a "safe" method. PUT, POST, and DELETE are not safe!

Idempotent Methods

Idempotent means that doing something 1 or N times has the safe effect. For example, if I flip a light switch on, further attempts to turn it on don't do anything new, the light stays on. In other words, something that is idempotent is considered to have no side effects.

GET, PUT, and DELETE are considered as idempotent, they should not have any side effects.

Why is POST not considered idempotent?

HTTP Response Codes

Whenever an HTTP request is made, the server responds with a code. These definitions of these codes are located at: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html A few key codes to know are:

- 200 : OK – The request has succeeded.
- 400 : Bad Request – The request was not understood by the server.
- 401 : Unauthorized – The request requires authentication.
- 404 : Not Found – The server could not find anything matching the request.
- 501 : Not Implemented – The server does not support the functionality required to fulfill the request.

Designing a Web Service API

Explain what in the world a web service is, give a little history of how we got here.

Use the following best practices:

- Correctly use the HTTP verb (GET, POST, PUT, DELETE) appropriate for the request.
- Version your API; it is going to change in the future, be prepared!
- Use routes to identify resources, use parameters only when necessary.

Create a table with the API and verbs, like this:

API	GET	POST	PUT	DELETE
/v1/high-scores	Return all high scores			
/v1/high-scores/{id}	Return the high score for the specified {id}			
/v1/high-scores/{name}	Return all high scores for the specified {name}.			
/v1/high-scores?name={name}&score={score}		Add a new high score.		
/v1/high-scores/{id}?score={score}			Replace the high score associated with {id}.	
/v1/high-scores/{id}				Delete the high score associated with {id}.
/v1/high-scores/{name}				Delete all the high scores associated with {name}.

Implementing a Web Service API with Node.js and Express

Use app.get to define the routes and what is performed when those routes are requested at the server.

```
app.get('/v1/high-scores', function(request, response) {
      response.writeHead(200, {'content-type' : 'application/json'} );
      response.end(JSON.stringify(scores));
});

app.get('/v1/high-scores/:id', function(request, response) {
      console.log(request.query.id);      // .query contains the params (as 
strings)
      ...
});

app.post('/v1/high-scores', function(request, response) {
      console.log(request.query.name);
      console.log(request.query.score);
      ...
});
```

Consuming a Web Service with jQuery

First thing is to include jQuery as part of your client side html, like the following:

```
<script type="text/javascript" src = "http://code.jquery.com/jquery-
latest.min.js"></script>
```

Consuming web services from a browser is known as AJAX (Asynchronous JavaScript And XML...even though we aren't going to worry about XML, we are using JSON). jQuery provides a convenient syntax for consuming a web service, it looks like:

```
$.ajax({
      url: 'http://localhost:3000/v1/high-scores',
      cache: false,
      error: function() { console.log('failed to access the web service'); },
      success: function(data) {
            console.log('received a response!');
            console.log('here is the data: ' + data);
      }
});
```

If you have provided the response in JSON format and indicated it in the content type, then 'data' is now a JavaScript object and can be used as such.