

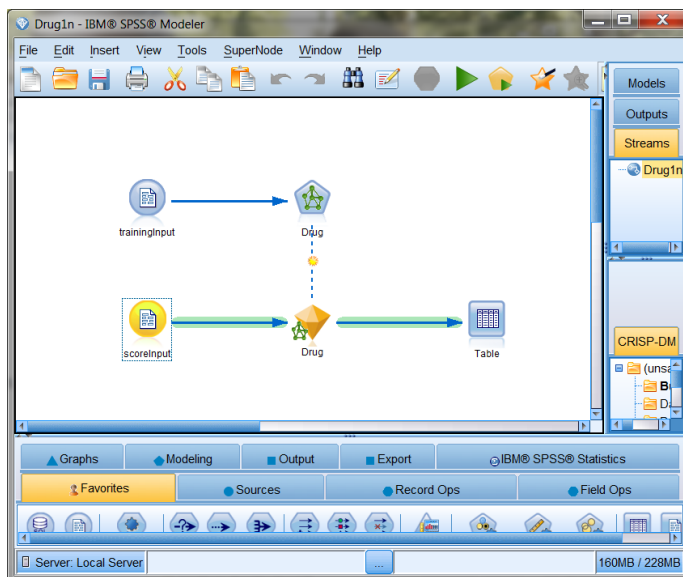
IBM Watson Machine Learning

Example Application #1

Preparation:

In this example, we'll use the IBM SPSS Modeler stream file *Drug1N.str* from the SPSS Modeler *Demos* set shipped with the product.

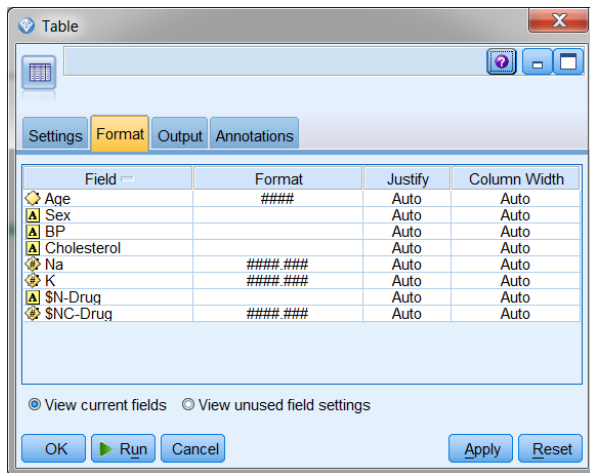
The scoring branch (highlighted in green) is the trained predictive model; a Neural Net algorithm in this case:



Our application must supply the required inputs on the source node when making a score request, as shown here:

Field	Measurement	Values	Missing	Check	Role
Age	Continuous	[15,74]	None	None	Input
Sex	Flag	M/F	None	None	Input
BP	Nominal	HIGH,LOW...	None	None	Input
Cholesterol	Flag	NORMAL/H...	None	None	Input
Na	Continuous	[0.500169,...	None	None	Input
K	Continuous	[0.020022,...	None	None	Input

The results of the scoring request will resemble the fields as summarized in the terminal node shown here, where the predicted drug is communicated in the \$N-Drug result field and the confidence of this prediction is communicated in the \$NC-Drug field for our application to use.



Field	Format	Justify	Column Width
Age	####	Auto	Auto
Sex		Auto	Auto
BP		Auto	Auto
Cholesterol		Auto	Auto
Na	#### #	Auto	Auto
K	#### #	Auto	Auto
\$N-Drug	#### #	Auto	Auto
\$NC-Drug	#### #	Auto	Auto

Preparing to develop the application

See the *IBM Watson Machine Learning service for Bluemix - General* document for an introduction to Bluemix application development.

Developing the NodeJS portion of the Single Page Application for this sample

We will be making some REST service calls, passing data in for scoring in our predictive model, and handling the score results, so we'll need some additional packages in our NodeJS application.

Adjusting the NodeJS `packages.json` and `app.js` files.

First let's open the `package.json` file and update the name and description, as well as add the **request** and **body-parser** packages. We'll use AngularJS in this example.

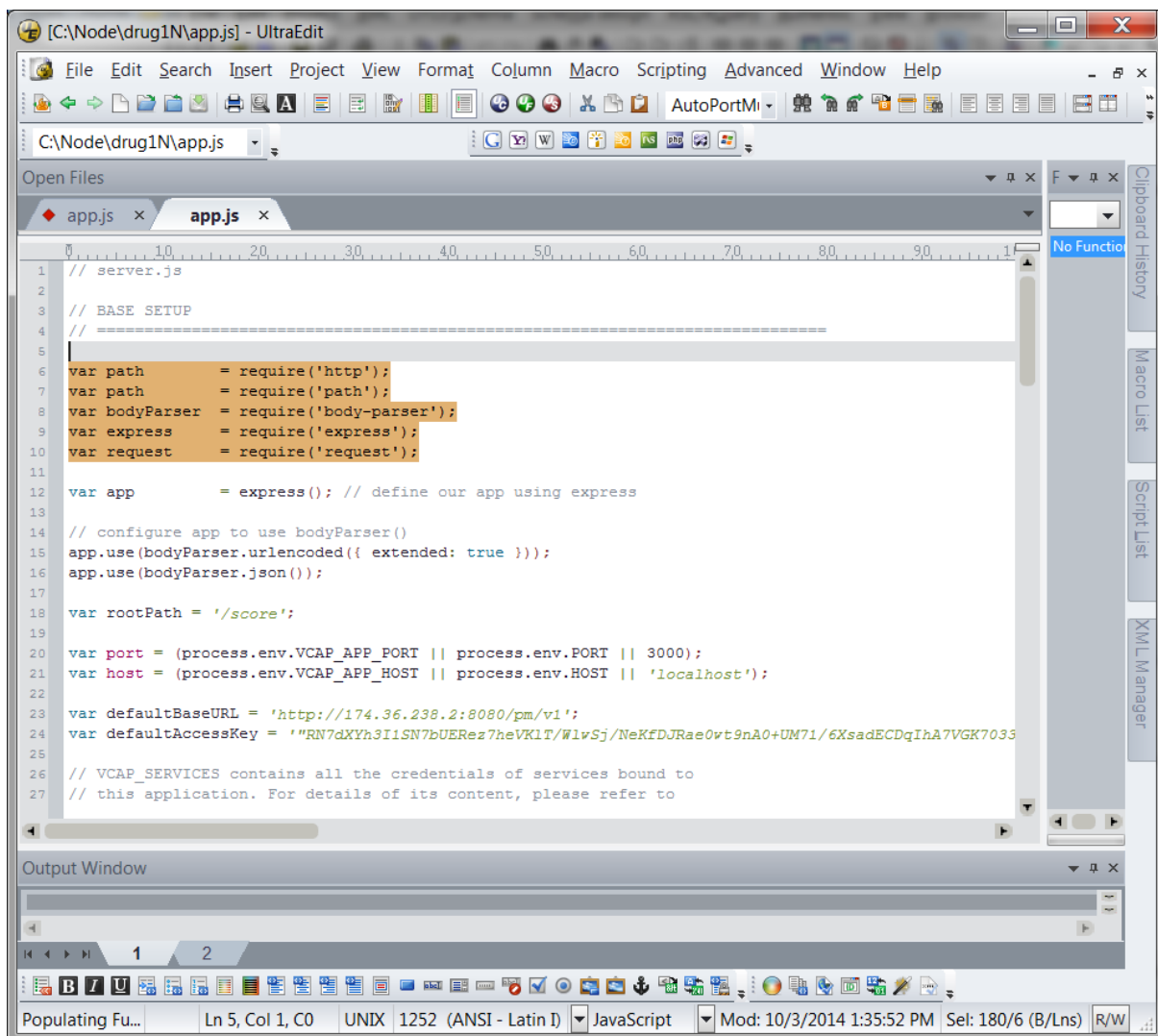
Original:

```
{
  "name": "NodejsStarterApp",
  "version": "0.0.1",
  "description": "A sample nodejs app for Bluemix",
  "dependencies": {
    "express": "3.4.7",
    "jade": "1.1.4"
  },
  "engines": {
    "node": "0.10.26"
  },
  "repository": {}
}
```

Modified:

```
{
  "name": "SPSS-PM-sample-1",
  "version": "0.1.0",
  "description": "A Node.js app using Express delivering SPSS PM sample application 1",
  "dependencies": {
    "express": "~4.0.0",
    "request": "2.36.x",
    "body-parser": "~1.0.1"
  },
  "engines": {
    "node": "0.10.26"
  },
  "repository": {}
}
```

We've added **http**, **path**, **body-parser**, and **request** to the **express** in this sample application. If we ask the Node Package Manager to pull down these packages by issuing the **npm install** command, we can watch it pull these help packages down to the desktop.



Defining the routing entries for our services and the entry point for our Single Page Application

We'll be using some helper services in this NodeJS application, as well as serving up our Single Page Application (SPA), so we need to set up some routing information. Note that we'll serve up the SPA from **public** instead of **views**, so we'll drop the *views* directory and its Hello World content.

```
// ROUTES
// ===== var router =
express.Router(); // get an instance of the express Router

// middleware to use for all requests
router.use(function(req, res, next) {
    next(); // make sure we go to the next routes and don't stop here
});

// TBD services and their routing...

// Register Service routes and SPA route -----
var rootPath = '/score';

// all of our service routes will be prefixed with rootPath
app.use(rootPath, router);

// SPA AngularJS application served from the root
app.use(express.static(path.join(__dirname, 'public')));
```

Machine Learning service connectivity, the Scoring Helper Service, and NodeJS startup

Let's set things up so we can test and debug our NodeJS application from our desktop.

To do this, we'll either use the port and host we get from Bluemix in the VCAP_APP_PORT and VCAP_APP_HOST environment variables, or the environment variable set on our local system, or some defaults.

```
var port = (process.env.VCAP_APP_PORT || process.env.PORT || 3000);
var host = (process.env.VCAP_APP_HOST || process.env.HOST || 'localhost');
```

We'll also want to be able to call our provisioned instance of the Machine Learning services for Bluemix, so we'll define some defaults for the service instance URL and access_key and initialize a data structure we'll modify using the VCAP_SERVICES information if we are deployed in Bluemix.

```
var defaultBaseURL = 'http://174.36.238.2:8080/pm/v1';
var defaultAccessKey =
'"RN7dXYh3I1SN7bUERez7heVK1T/Wlwsj/NeKfDJRae0wt9nA0+UM71/6XsadECDqIhA7VGK7033Xo
CVgABt84wo7Io6/ltsqOs0i7k0j8lNI9jBxf8YqDOGT0+qpTLwzRqXP+5vfe4jhLDBIIf4BdQ=="';

var env = { baseURL: defaultBaseURL, accesskey: defaultAccessKey };
```

Now let's look at the VCAP_SERVICES information Bluemix sets in our environment. The Machine Learning service will be identified by a **pm-20** label, and from that we are interested in the credentials where the URL of our service instance and the **access_key** to be used as set in the **bind** event are communicated to us. We'll store this information in a simple data structure for later use.

```
// VCAP_SERVICES contains all the credentials of services bound to
// this application. For details of its content, please refer to
// the document or sample of each service.
var services = JSON.parse(process.env.VCAP_SERVICES || "{}");
var service = (services['pm-20'] || "{}");
var credentials = service.credentials;
if (credentials != null) {
    env.baseUrl = credentials.url;
    env.accesskey = credentials.access_key;
}
```

Next we'll define a helper service for the thin-client UI to use for making score requests:

```
// score request
router.post('/', function(req, res) {
    var scoreURI = env.baseUrl + '/score/' + req.body.context + '?accesskey='
+ env.accesskey;
    try {
        var r = request({ uri: scoreURI, method: "POST", json:
req.body.input });
        req.pipe(r);
        r.pipe(res);
    } catch (e) {
        console.log('Score exception ' + JSON.stringify(e));
        var msg = '';
        if (e instanceof String) {
            msg = e;
        } else if (e instanceof Object) {
            msg = JSON.stringify(e);
        }
        res.status(200);
        return res.send(JSON.stringify({
            flag: false,
            message: msg
        }));
    }

    process.on('uncaughtException', function (err) {
        console.log(err);
    });
});
```

And the last step in the NodeJS work is to start things up:

```
// START THE SERVER with a port reminder when run on the desktop
// =====
app.listen(port, host);
console.log('App started on port ' + port);
```

Developing the HTML, CSS, and AngularJS portion of the Single Page Application for this sample

HTML used

Our SPA for this example will be defined in *index.html* to make things simple. We won't go into the CSS used here. You can read through it in the sample bundle if you are interested.

First we define our application in AngularJS terms and set the main controller:

```
<body ng-app="drugInSample" ng-controller="AppCtrl" >
```

Then we define the main form that dominates this simple UI. The main thing to note here are the ng-model definitions binding these HTML elements to a data model managed by the controller in the Angular MVC pattern:

```
<div class="section">
  Age: <input type="text" ng-model="p.Age" size="4" required>

  Sex: <input type="radio" ng-model="p.Sex" value="M">Male</input>
      <input type="radio" ng-model="p.Sex" value="F">Female</input>

  Blood Test Results:
    Blood Pressure:
      <input type="radio" ng-model="p.BP" value="HIGH">HIGH</input>
      <input type="radio" ng-model="p.BP" value="NORMAL">NORMAL</input>
      <input type="radio" ng-model="p.BP" value="LOW">LOW</input>

    LDL (bad) Cholesterol:
      <input type="radio" ng-model="p.Cholesterol" value="HIGH">HIGH</input>
      <input type="radio" ng-model="p.Cholesterol" value="NORMAL">NORMAL</input>
      <input type="radio" ng-model="p.Cholesterol" value="LOW">LOW</input>

    Sodium Level:
      <input type="text" ng-model="p.Na" size="4" required>

    Potassium Level:
      <input type="text" ng-model="p.K" size="4" required>
</div>
<div class="section">
  <button type="button" class="btn.lg" ng-click="score()" >
    <i class="glyphicon glyphicon-cloud-upload"></i> Score Now </button>
</div>
```

Certain countries block Google, in which case you'll have to deploy AngularJS yourself. Our example here will just pull it in dynamically via reference and then we pull in the controllers and services we define for this application:

```
<!-- load angular via CDN -->
<script
src="//ajax.googleapis.com/ajax/libs/angularjs/1.2.10/angular.js"></script>
<script src="//angular-ui.github.io/bootstrap/ui-bootstrap-tpls-
0.11.0.js"></script>

<!-- our scripts -->
<script src="js/app.js" type="text/javascript" ></script>
<script src="js/srv.js" type="text/javascript" ></script>
```

The controller

This module uses some dialog and data services from another module, so we'll note these dependencies now:

```
Var AppCtrl = ['$scope', 'dialogServices', 'dataServices',  
function AppCtrl($scope, dialogServices, dataServices) {
```

When we deploy an SPSS Modeler file in Bluemix, we give it an alias called its *context ID* that's used when submitting a score request against that model:

```
// context ID is a configuration constant in this example  
$scope.context = 'drug1N';
```

We initialize the data model used by the UI with some reasonable values:

```
// init UI data model  
$scope.p =  
  { Age:'35', Sex:'M', BP:'NORMAL', Cholesterol:'NORMAL', Na:'0.697',  
    K:'0.056' };
```

There is one button on this UI telling us to score with the data set on our form. This request will be performed in an asynchronous fashion, and we'll react to the results when they come in by displaying the information returned in a model dialog OR by showing whatever error messages we get back in an error dialog.

```
$scope.score = function() {  
  dataServices.getScore($scope.context, $scope.p)  
    .then(  
      function(rtn) {  
        if (rtn.status == 200){  
          // success  
          $scope.showResults(rtn.data);  
        } else {  
          //failure  
          $scope.showError(rtn.data.message);  
        }  
      },  
      function(reason) {  
        $scope.showError(reason);  
      }  
    );  
}  
  
$scope.showResults = function(rspHeader, rspData) {  
  dialogServices.resultsDlg(rspHeader, rspData).result.then(); }  
  
$scope.showError = function(msgText) {  
  dialogServices.errorDlg("Error", msgText).result.then(); }  
}]
```

The dialog and service call helpers

The data services are our **getScore** helper that builds the input data structure that will be passed. This data model is defined by the scoring branch of the SPSS Modeler file we looked at earlier. Note that the **tabular input data source name** and all other names must match the scoring branch as defined in the deployed predictive model.

```
sampleSrv.factory("dataServices", ['$http',
function($http) {
    this.getScore= function(context, p) {
        /* create the scoring input object */
        var input = {
            tablename: 'scoreInput',
            header: [ 'Age', 'Sex', 'BP', 'Cholesterol', 'Na', 'K'
                ],
            data: [[ p.Age, p.Sex, p.BP, p.Cholesterol, p.Na, p.K ]]
        };

        /* call      scoring service      to generate results */
        return $http({      method: "post",
                                url: "score",
                                data: { context: context, input: input }
                            })
            .success(function(data, status, headers, config) {
                return data;
            })
            .error(function(data, status, headers, config) {
                return status;
            });
    }

    return this;
}]);
```

We already discussed the dialog services, but we should look at the data model bindings and the partial HTML template used here to help see how we handle single row and multiple row returns. The basic work done here is to take the **score results** object we get back from a successful score request and make its contents easy for the HTML template to process. The two **resolve** items make the field names available by a **rspHeader** reference and the **0..N** result rows available by a **rspData** reference.

```
sampleSrv.factory("dialogServices", ['$modal',
function($modal) {

    this.resultsDlg = function (r) {
        return $modal.open({
            templateUrl: 'partials/scoreResults.html',
            controller: 'ResultsCtrl',
            size: 'lg',
            resolve: {
                rspHeader: function () {
                    return r[0].header;
                }
            }
        });
    }
}]);
```



```

        },
        rspData: function () {
            return r[0].data;
        }
    });
}

return this;
}));

```

The HTML template used for score results is interesting. We use an **ng-repeat** on the number of fields in the **rspHeader** to insert column headers in the table:

```

<thead>
<tr>
<th ng-repeat="fname in rspHeader">{{fname}}</th>
</tr>
</thead>

```

Then we use an **ng-repeat** on the rows we got back and a lower level **ng-repeat** on the number of columns in each row to display all values for all rows in the response:

```

<tbody>
<tr ng-repeat="row in rspData">
<td ng-repeat="fval in row track by $index">{{fval}}</td>
</tr>
</tbody>

```

That covers the AngularJS implementation of the UI for our Single Page Application.

Testing the Single Page Application for this sample

See the *IBM Watson Machine Learning service for Bluemix - General* document for an introduction to Bluemix application testing.