The app is a node app calling Watson Question and Answer REST service. its on github. Make sure you have git installed

```
git clone https://github.com/iwinoto/qa-sample-node.git
```

When run locally, its hard-wired to my instance of the Watson QA service so you can do:

```
npm install
npm start
```

Then go to http://localhost:3000/

All the logic is in app.js. The view stuff is in views/ and public/. The app uses express for the HTTP serving & request routing and Jade templating engine to generate the HTML. Jade makes human readable files into HTML. The .jade template files are in views/.

Walking through the code:

To do the REST requests, I need some modules

```
9  var https = require('https');
10 var url = require('url');
```

Set up default service credentials for running locally

```
29 // defaults for dev outside bluemix
30 var service_url = 'https://gateway.watsonplatform.net/qagw/service';
31 var service_username = '7b0710cb-8c80-433a-b780-d1042ba8abfc';
32 var service_password = 'Z7BPYJwDoBdV';
```

If we are running on Bluemix, then the service credentials will be in VCAP_SERVICES environment. This block tests for the variable and extracts the service credentials.
If it exists, get VCAP_SERVICES json object into a variable.

```
35 // VCAP_SERVICES contains all the credentials of services bound to
36 // this application. For details of its content, please refer to
37 // the document or sample of each service.
38 if (process.env.VCAP_SERVICES) {
39   console.log('Parsing VCAP_SERVICES');
40   var services = JSON.parse(process.env.VCAP_SERVICES);
```

The service type is known as “question_and_answer”. We can find this from Bluemix. Look for a service with this name

```
41   //service name, check the VCAP_SERVICES in bluemix to get the nar
42   var service_name = 'question_and_answer';
43
44   if (services[service_name]) {
```

If its found, then get the credentials and extract the relevant information. For the Watson Q&A, we want the URL, username and password. This is common, but for some services, like APIs from APImanagement, we want the client ID and key as well as the URL.

```
45      var svc = services[service_name][0].credentials;
46      service_url = svc.url;
47      service_username = svc.username;
48      service_password = svc.password;
```

Log any problems

```
49    } else {
50      console.log('The service '+service_name+' is not in the VCAP_SE
51    }
52
53  } else {
54    console.log('No VCAP_SERVICES found in ENV, using defaults for lo
55  }
56
57  console.log('service_url = ' + service_url);
58  console.log('service_username = ' + service_username);
59  console.log('service_password = ' + new Array(service_password.leng
```

The Watson REST service wants to base64 encode the credentials. Store the encoded string for later use.

```
61  var auth = "Basic " + new Buffer(service_username + ":" + service_p
```

This next bit is an actual REST call to POST a question. It happens in the function to handle a POST request from the application web page.

```
68  // Handle the form POST containing the question to ask Watson and r
69  app.post('/', function(req, res){
```

First we get the URL (string) and parse it into its parts. This one is specific to Watson Q&A

```
76    var parts = url.parse(service_url +'/v1/question/' + watsonServic
```

Create the json object to hold the HTTP request options

```
78    // create the request options to POST our question to Watson
79    var options = { host: parts.hostname,
80      port: parts.port,
81      path: parts.pathname,
82      method: 'POST',
83      headers: {
84        'Content-Type'  :'application/json',
85        'Accept':'application/json',
86        'X-synctimeout' : '30',
87        'Authorization' :  auth }
88    };
```

Now we get into call-back heaven. Node.js is non-blocking, so you have to code as asynchronous. This means, almost all methods will send a call back that you need to handle. In this case we first create the HTTPS request object that will handle the response from sending Watson a question and store this request object in a variable called "watson_req".
The call back handler for this request object start with the function declaration **`function(result)`** where **`result`** is the function parameter that will hold the object to deal with HTTPS results.

```
90    // Create a request to POST to Watson
91    var watson_req = https.request(options, function(result) {
```

We expect the result to be UTF-8 encoded, so we make sure the result object knows this.

```
92        result.setEncoding('utf-8');
```

We also create and empty variable to hold the results when it arrives

```
93        var response_string = '';
```

The **`result`** object will emit various events some of which we are interested in. Using the **`on`** method, we add some event handler methods to the **`result`** object for the events we are interested in
'**`data`**': Data comes in chunks. Each time a chunk of data is received, the **`data`** event is emitted. We need to get the data and add it to our result string:

```
95        result.on('data', function(chunk) {
96          response_string += chunk;
97        });
```

'**`end`**': At the end of the data stream, the **`end`** event is emitted. Now we know we've got the complete result, so we can parse it and display it. How you handle this depends on how your API responds. In most cases its JSON, so you can parse the string into a JSON object as we do here. This one deals with a JSON array of answers to the question that we're going to send.

```
 99      result.on('end', function() {
100        var answers_pipeline = JSON.parse(response_string);
101        answers = answers_pipeline[0];
102        /*
103        console.log("[INF]", "Watson answers = " + util.inspect(answ
104        console.log("[INF]", "Answers: " + util.inspect(answers.ques
105        console.log("[INF]", "EvidenceList: " + util.inspect(answers
106        */
107        feedback = new Array(answers.length);
108        answers.question.answers.forEach(function(answer, index){
109          feedback[index] = {
110              questionId: answers.question.id,
111              answerid : answer.id,
112              feedback : "0",
113              comment : ""
114          };
115        });
116        return res.render('index',
117          {
118            'questionText': req.body.questionText,
119            'answers': answers,
120            'feedback': feedback,
121            'service': watsonService})
122      })
```

'**error**': we always need to handle the error

```
126    watson_req.on('error', function(e) {
127      return res.render('index', {'error': e.message})
128    });
```

Now that we've told **result** how to handle events, we can send the actual request with the payload. In this case, the payload is a JSON object containing the question and some meta information.

```
130    // create the question to Watson
131    var questionData = {
132      'question': {
133        'evidenceRequest': {
134          'items': 5 // the number of anwers
135        },
136        'questionText': req.body.questionText // the question
137      }
138    };
```

Now we can write the payload to the prepared HTTPS request object and end the request. The call to **end()** is important, otherwise the request object will just wait and not close the request.

```
140   // Set the POST body and send to Watson
141   watson_req.write(JSON.stringify(questionData));
142   watson_req.end();
143
144 });
```

The rest of the code is more of the same.