Build a Hangman game with Java, Ajax, and Cloudant

Leopoldo Moreno Mata (https://www.ibm.com/developerworks/community/profiles/html/profileView.do?key=2c32dbec-3615-4fc1-90b5-9637acab8219&lang=en&tabid=dwAboutMe)
IT Specialist
IBM

04 September 2014

Learn how to build an online Hangman game by using the Bluemix Liberty for Java runtime and Cloudant NoSQL database service.

Have you played Hangman, the paper-and-pencil guessing game? Now you'll learn how to program your own online Hangman game, in which the app simulates your opponent. I'll cover the steps for using two services available in IBM® Bluemix™— the Liberty for Java™ runtime and the Cloudant NoSQL database — to build the application.

READ: Creating apps with Liberty for Java

READ: Getting started with Cloudant NoSQL Database

What you'll need for your application

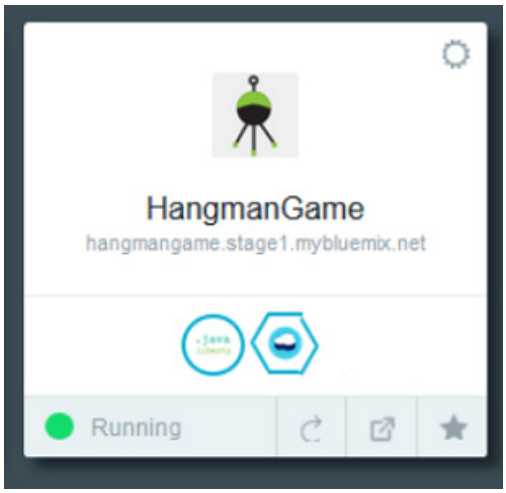
- Bluemix and IBM DevOps Services accounts
- Basic familiarity with JavaServer Pages (JSP) technology, servlets, JavaScript, and Dojo
- Basic familiarity with JSON, Apache CouchDB, and Cloudant
- A Java development environment such as Eclipse
- The Cloud Foundry command-line tools
- Dojo Base

“ You'll create the back and front ends for the application and modify the web.xml file. ”

If you only want to know how easy it is to use Bluemix services, get the code and follow Steps 1, 2, 3, and 5. For more fun, follow Step 4 too and learn how the app uses JSP technology, servlets, Ajax, CSS, JavaScript, and a Cloudant connection.

Step 1: Create a Java web application in Bluemix

1. Log in to Bluemix.
2. In the catalog, click **Liberty for Java** under Runtimes.
3. Choose and enter your application name and host name, and select **Default** as your plan. Click **Create**.
4. In the catalog, click **Cloudant NoSQL DB** under Data Management. Select your app from the list, and select **Shared** as your plan. Click **Create**.
5. In the dashboard, click your application to go to its overview page:



Step 2: Populate the Cloudant NoSQL DB database

> Cloudant is compatible with the Apache CouchDB project, so you can use any driver that uses CouchDB as the store engine. For this tutorial, I use Ektorp.

1. In the application overview page, click the **Cloudant NoSQL DB** service, under Development Services.
2. Click **Launch** to launch the Cloudant console.
3. Click **Databases** in the menu and click **Add New Database** to add databases named `category` and `word`.
4. Using the **New > Document** menu item, add the following documents (one by one) into the `category` database:

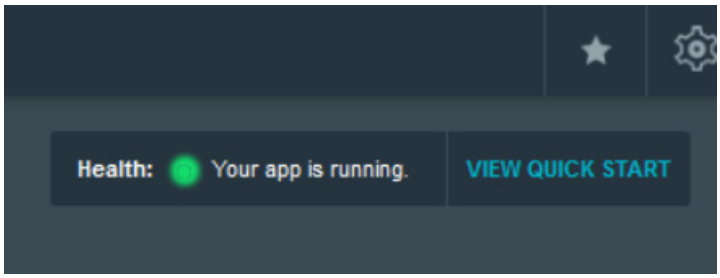
```
{"_id":"0","name":"Animals"}
{"_id":"1","name":"Food"}
{"_id":"2","name":"Music"}
{"_id":"3","name":"Movies"}
{"_id":"4","name":"Names"}
{"_id":"5","name":"Video Games"}
```

5. Add the following documents (one by one) into the `word` database:

```
{"name":"CAIMAN","category_id":"0"}
{"name":"BEAR","category_id":"0"}
{"name":"CARROTS","category_id":"1"}
{"name":"AVOCADO","category_id":"1"}
{"name":"LEOPOLDO","category_id":"4"}
{"name":"ISMAEL","category_id":"4"}
{"name":"LOREN","category_id":"4"}
{"name":"SPACE RUN","category_id":"5"}
{"name":"WATCH DOG","category_id":"5"}
```

Step 3: Download the starter application package

1. In your application overview page, click **VIEW QUICK START**:



2. Click the **Download the starter application package** link and save the file to your local PC.

Step 4: Build your application

In this step, you'll create the back and front ends for the application and modify the web.xml file. (Click the **Get the code** button at the beginning of this tutorial to download the complete CSS file, JavaScript file, Java classes, and all the files that the application needs.)

Prepare your environment

1. Create a new Java dynamic web project in your IDE.
2. Import the starter application package into your project.
3. Add the org.ektorp.jar file in the library folder (WEB-INF/lib) to your project. Also add the library dependencies. All of the libraries are available from DevOps Services via this tutorial's **Get the code** button.

Create your back end

1. Create the Category.java and Word.java classes, with the following attributes:
   - **`Category`**: `String id`, `String revision`, `String name`
   - **`Word`**: `String id`, `String revision`, `String name`, `String category_id`
2. Create the getters and setters. Use the `@JsonIgnoreProperties` and `@JsonProperty` annotations, which cause the processing of the JSON properties to be ignored:

```
@JsonIgnoreProperties({"id", "revision"})

public class Category {

   @JsonProperty("_id")
   private String id;

   @JsonProperty("_rev")
      private String revision;
```

3. Create the CategoryRepository.java and WordRepository.java classes. In the classes, extend the `CouchDbRepositorySupport<T>` class, which is a generic repository-support class that provides all create, read, update, and delete operations for a persistent class. In the classes, add the constructor:

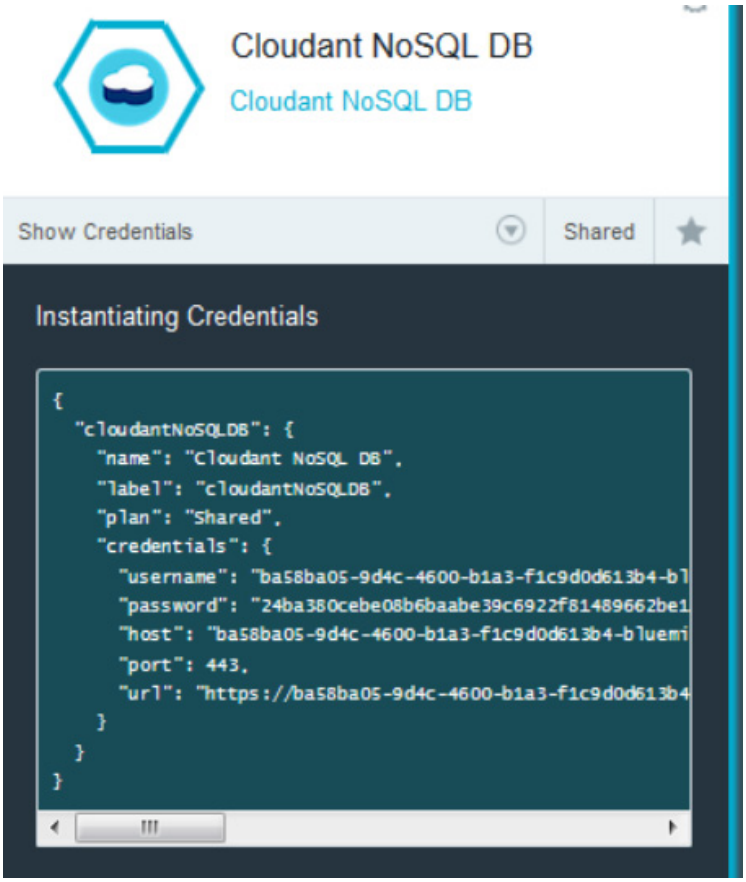
```
public CategoryRepository(CouchDbConnector db) {
     super(Category.class, db);
}
```

4. Create the CloudantConnection.java class. Set the connection in your constructor, using the `VCAP_SERVICES` environment variable from your runtime:

```
JSONObject obj = new JSONObject(System.getenv("VCAP_SERVICES"));
   String[] names = JSONObject.getNames(obj);

   if (names != null) {
   for (String name : names) {
      if (name.equals("cloudantNoSQLDB")) {
      JSONArray val = obj.getJSONArray(name);
      JSONObject serviceAttr = val.getJSONObject(0);
      JSONObject credentials = serviceAttr.getJSONObject("credentials");
      httpClient = new StdHttpClient.Builder()
         .url(credentials.getString("url"))
         .build();
      break;
      }
      }
   }
```

You can find your environment variables in your Liberty for Java runtime overview. The `VCAP_SERVICES` variable has all your database information (username, password, host, port, and URL); if you click **Show Credentials** under your Cloudant service, you'll see that information:



5. Create the `getCategories()` and `getWords()` methods to get data from the Cloudant database. You'll use the repository classes to read data:

```
public List<Word> getWords(){
   CouchDbInstance dbInstance = new StdCouchDbInstance(httpClient);
   CouchDbConnector db = new StdCouchDbConnector("word", dbInstance);
   WordRepository wordRepo = new WordRepository(db);
   return wordRepo.getAll();
}
```

6. Create the `getWordsByCategory()` and `getRandomWordByCategory()` methods to get a list of words by category and a random word for each category. In the `getWordsByCategory()` method you can call to the `getWords()` method to get the available words and then iterate through the list to find the words by category. In `getRandomWordByCategory()`, you create a random number to select the word that will be used in the game:

```
List<Word> words = this.getWordsByCategory(category_id);
Random generator = new Random();
if(words.size()>0){
   int random = generator.nextInt(words.size());
   word = words.get(random);
}
```

   I used the `Random` object (available in the `java.util package`) to create an integer value randomly.
7. Create the LoadIndex.java servlet, which is used to load categories information into the index web page. You need the `doGet()` method only where you use the `CloudantConnection` class:

```
CloudantConnection cloudantConnection = new CloudantConnection();
List<Category> categories = cloudantConnection.getCategories();
request.setAttribute("categories", categories);
 request.getRequestDispatcher("index.jsp").forward(request, response);
```

8. Create the LoadGame.java servlet. Use the `doPost()` method and set the `HttpServletResponse` with the random word. The random word is generated by the category that is sent in the `HttpServlerRequest`:

```
String action = request.getParameter("action");
String value = request.getParameter("value");

if ((action != null)&&(value != null)) {
 CloudantConnection cloudantConnection = new CloudantConnection();
 Word word = cloudantConnection.getRandomWordByCategory(value);
 if(word!=null){
    response.setContentType("text/html");
    response.getWriter().write(word.getName());
 }
}
```

   You'll use this method for the Ajax implementation.

Create your front end

1. Create the index.jsp file and import the necessary libraries:

```
<%@ page import="java.util.List" %>
<%@ page import="com.bluemix.hangman.model.Category" %>
```

2. Add references to the CSS and JavaScript files:

```
<link rel="stylesheet" href="style.css" />
<script src="index.js"></script>
<script src="dojo.js"></script>
```

3. In the body, add the `<div>` tags and the `<select>` drop-down list. The list will be filled with the list of categories that you get from the `LoadIndex` servlet. Also add the `<img>` and `<table>` tags (which are empty now and will be filled in dynamically):

```
<div id="menu">
      <select onChange="javascript:loadWord(this.value);">
      <option value="">Select category</option>
      <%  List<Category> categories = (List<Category>) request.getAttribute("categories");
      for(int index=0; index<categories.size(); index++){
       %>
      <option value="<%=categories.get(index).getId()%>"
      ><%=categories.get(index).getName() %></option>
      <%
      }
       %>
    </select>
</div>
<div id="content">
   <img id="hangmanImage" style="visibility:hidden"><br><br>
   <table id="wordTable"></table><br>
   <table id="lettersTable"></table>
   </div>
```

4. Create the style.css file to provide CSS styles for the `<table>`, `<div>`, and `<a>` tags. The style classes display the letters of the alphabet that are used in the game. (Remember that you can apply a style class to any HTML element by using `class="myclassname"`.)
5. Create the index.js JavaScript file. Define the global variables and copy the dojo.js file in the WebContent folder. Add the `loadWord` function, which is used to implement Ajax through Dojo functionality. This function is used to get a word from the `LoadGame` servlet:

```
function loadWord(category) {
   dojo.xhrPost({
      url: "game.do",
      postData: "action=loadWord&value="+category,
      handleAs: "text",
      load: function(text){
            updateWord(text);
      },
      error: function(error){
            alert(error);
      }
   });
}
```

In the `loadWord` function call to the `updateWord` function, the JavaScript variables are initialized and call other functions to fill in dynamically the HTML tags defined in the index.jsp file. Here, the `<img>` tag is filled by using the `document.getElementById()` JavaScript method, indicating that a new game has started:

```
document.getElementById("hangmanImage").style.visibility = "visible";
```

The `loadWordTable` function is used to add the word to guess in the UI. The `string.split(separator)` JavaScript method is used to count the characters and white spaces of the words. The `loadLettersTable` function is used to print the letters of the alphabet that will be used to play. The `updateImage` function is used to update an image in the UI. As you can see, `document.getElementById("elementId")` is used to access the HTML elements and update them:

```
var table = "<tr>";
for(var index=0; index<word.split('').length; index++){
   table += "<td ><a id='wordLetter"+index+"' class='wordLetter'>_</a></td>";
}
table += "</tr>";
document.getElementById("wordTable").innerHTML = table;\
```

6. Create functions to control and update the game status. The `verifyLetter` function, called when the user clicks a letter of the alphabet, checks how many chances remain. The `updateGame` function checks if the letter exists in the word and updates the image, letter of alphabet, and correct letters status. Again, the `split(separator)` JavaScript method is used to split a string into an array of substrings. In this case, the method has an empty string as a separator, so the original string is split after each character:

```
var wordSplit = globalWord.split('');
for(var index=0; index<wordSplit.length; index++){
       if(letter == wordSplit[index]){
   document.getElementById("wordLetter"+index).innerHTML = wordSplit[index];
   correctLetters+=1;
   find = true;
    }
}
```

7. Create the img folder in the WebContent folder and add the images.

Edit the web.xml file

1. Configure the `LoadIndex` and `LoadGame` servlets in the web.xml file:

```
<servlet>
    <servlet-name>LoadIndex</servlet-name>
    <servlet-class>com.bluemix.hangman.controller.LoadIndex</servlet-class>
</servlet>
<servlet>
    <servlet-name>LoadGame</servlet-name>
    <servlet-class>com.bluemix.hangman.controller.LoadGame</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>LoadIndex</servlet-name>
    <url-pattern>/play.do</url-pattern>
</servlet-mapping>
   <servlet-mapping>
    <servlet-name>LoadGame</servlet-name>
    <url-pattern>/game.do</url-pattern>
</servlet-mapping>
```

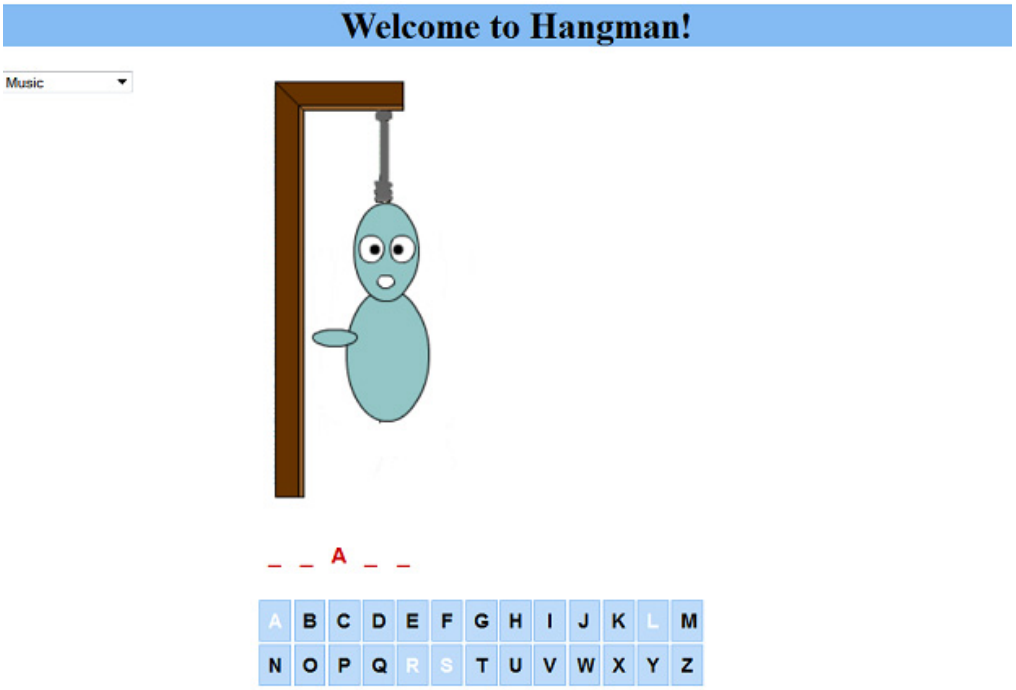
2. Specify the default page of the web application by calling to the `LoadIndex` servlet:

```
<welcome-file-list>
    <welcome-file>play.do</welcome-file>
</welcome-file-list>
```

Step 5: Run the application

1. Compile the Java web project and generate the WAR file.
2. From the command line, change to the directory where you saved your WAR file.
3. Connect to Bluemix by running `cf api` *`bluemix_domain`*.
4. Log in to Bluemix with `cf login -u` *`username`* and target your environment (Bluemix space) by running `cf target -o` *`username`* `-s` *`space`*.
5. Deploy the application by running the `cf push` *`appname`* `-p` *`appname`*`.war` command.

Now you can access the application in your Bluemix domain (for example, http://hangmangame.mybluemix.net) and play the game:



Conclusion

Using Bluemix, you can rapidly deploy and manage your cloud applications. I used only one runtime and one service to deploy the Hangman application in the cloud, but an extensive software portfolio is available in Bluemix. Get started today!

RELATED TOPICS: Cloudant Dojo

About the author

Leopoldo Moreno Mata

Follow me on G+