HSE Offline App Parts

1. Templates:
   Templates are fragments of HTML for each 'page' in the application. They are associated with an id defined in the <script> tag, which allows the view for each page to find the correct template. They can contain embedded Javascript through Underscore.js. <%= %> indicates a variable binding, as in the example below. The view associated with the template is responsible for providing the context for all values, via the _.template function (as explained further below).

```
<input style="border-left: 1px solid black; padding:0px" type="datetime-local" name="Incident_Date_Time__c" value="<%= Incident_Date_Time__c %>" />
```

   <% %> is simply a script.

```
<% if (Form_Group__c == "Incident") { %>
<div class="input-group">
  ...
</div>
<% } else if (Form_Group__c == "Near Miss") { %>
<div class="input-group">
  ...
</div>
<% } %>
```

   Templates are all currently contained in index.html.
2. Models:
   Models represent objects, and extend the Force.SObject prototype (which itself extends from Backbone). They thus inherit save and fetch functionality automatically. You will need to define the sObject they represent, the list of fields on that object to fetch, the cache to store them in, and a cacheMode to define order of access for cache and server. Optionally, if you want to check for conflicts, define a cacheForOriginals.

```
app.models.ActivityFormAnswer = Force.SObject.extend({
    sobjectType: "Activity_Form_Answer__c",
    relationshipField: "Activity_Form__c",
    fieldlist: ["Id", "RecordTypeId", "Activity_Form__c", "Question_Order__c", "Answer__c"],
    cache: function() {return app.answerCache;},
    cacheMode: function (method) {
        if (!app.offlineTracker.get("isOnline")) {
            return Force.CACHE_MODE.CACHE_ONLY;
        } else {
            return (method == "read" ? Force.CACHE_MODE.CACHE_FIRST : Force.CACHE_MODE.SERVER_FIRST);
        }
    }
});
```

   Collections are also defined separately. These extend Force.SObjectCollection (which, again, extends from Backbone). They are used to hold results of a query, defined in the config property. Again, define the fields and cache for the object, and the model it will hold.

```
app.models.ActivityFormCollection = Force.SObjectCollection.extend({
    model: app.models.ActivityForm,
    fieldlist: ["Id", "RecordTypeId", "Form_Group__c", "Consequence__c", "Job__c", "Task__c", "Location__c", "Incident_Date_Time__c", "Inc__c", "Incident_Description__c", "Equipment_in_use__c", "Specif
    cache: function() { return app.cache},
    cacheForOriginals: function() { return app.cacheForOriginals;},

    config: function() {
        // Offline: do a cache query
        if (!app.offlineTracker.get("isOnline")) {
            return {type:"cache", cacheQuery:{queryType:"smart", smartSql:"SELECT {activity_form__c:_soup} FROM {activity_form__c} WHERE {activity_form__c:Form_Group__c} LIKE '" + (this.key == null ? "
        }
        // Online
        else {
            // First time: do a full query
            if (this.key == null) {
                return {type:"soql", query: "SELECT " + this.fieldlist.join(",") + " FROM Activity_Form__c ORDER BY Form_Group__c"};
            }
            // Other times: do a SOQL query
            else {
                return {type:"soql", query: "SELECT " + this.fieldlist.join(",") + " FROM Activity_Form__c WHERE Form_Group__c like '" + this.key + "%' ORDER BY Form_Group__c LIMIT 25"};
            }
        }
    }
});
```

3. Views:

These represent the page itself, and hold any logic needed for the interface. They link to a model to display data, and update automatically when their model changes. They define an initialize and render function which are called when the view is created and loaded, respectively, for any additional logic needed. They are responsible for passing in their model as context to a template, as below.

```
app.views.EditActivityFormPage = Backbone.View.extend({

    action: null,
    backAction: "#",

    template: _.template($("#edit-form-page").html()),          <- Gets the template with the id edit-form-page

            ...                                                                    passes the model in as
        $(this.el).html(this.template(_.extend({action: this.action}, this.model.toJSON())));  <-- a JSON string
```

They contain logic associated with buttons or change events.

```
app.views.OfflineToggler = Backbone.View.extend({

    template: _.template($("#offline-toggler").html()),

    events: {
        "click .toggleStatus": "toggle",
        "click .syncFiles": "syncFiles"
    },

    initialize: function() {
        this.model.on("change:isOnline", this.render, this);
    },

    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    },

    toggle: function(event) {
        event.preventDefault();
        this.model.set("isOnline", !this.model.get("isOnline"));
    },

    syncFiles: function() {
        app.router.navigate("#sync", {trigger:true});
    }


});
```

```
  change: function(evt) {
      // apply change to model
      var target = evt.target;
      var type = target.type;
      var value = target.value;
      if (type === "datetime-local") {
          value = formatDateTimeForSF(target.value);
      }
      this.model.set(target.name, value);
      $("#form" + target.name + "Error", this.el).hide();
  },
```

They can also be nested.

```
    initialize: function() {
        var that = this;
        _.each(["reset","add","remove"], function(eventName) { that.model.on(eventName, that.render, that); });
        this.listView = new app.views.ActivityFormListView({model: this.model});<- set up inner view for list on sync page
    },

    render: function(eventName) {
        $(this.el).html(this.template(_.extend({countLocallyModified: this.model.length}, this.model.toJSON())));
        this.listView.setElement($("ul", this.el)).render();          <- nested views must be rendered explicitly
        return this;
    },
```

4. Router:

   This is the equivalent of the MVC's controller, linking actions to navigation. Templates contain links within:

```
<script id="form-list-item" type="text/template">
    <a href="#edit/forms/<%= Id %>/false"> <- <%= Id %> is dynamic, set at runtime via Underscore
```

   Which are set in the router to point to a specific function:

```
routes: {
    "": "mainPage",
    "list": "list",
    "add/:type": "list",
    "add/form/:id/:fromServer": "addForm",
    "edit/forms/:id/:fromServer": "editActivityForm",
    "sync":"sync"
},
```

   Which then run and display views when clicked and set models as needed, enabling navigation:

```
editActivityForm: function(id) {
    var that = this;
    var form = new app.models.ActivityForm({Id: id});
    form.fetch({
        success: function(data) {
            app.editPage.model = form;
            app.editPage.model.set("Incident_Date_Time__c", formatDateTimeForJS(form.get("Incident_Date_Time__c")));
            that.slidePage(app.editPage);
        },
        error: function(model, error) {
            if (error) {
                console.log("error: " + JSON.stringify(error));
            }
            alert("Failed to get record for edit");
        },
        cacheMode: Force.CACHE_MODE.CACHE_ONLY
    });
},
```