A Schema-Agnostic Approach to Web Application development

----James Xie (jimx83@yahoo.com)

# Summary:

This framework is aimed to design an architecture/solution that enables Create, Retrieve, Update and Delete (a.k.a. CRUD) operations upon heterogeneous tables/schema whereas client-side JavaScript files have no prior knowledge of the underlying database structure, through integration of various technologies that include open-source Spring MVC, JPA, Hibernate, RESTful, AngularJS, Ajax, JSON, Bootstraps, JavaScript, etc.

***Simply put, one single JavaScript file (SearchEditForAll.js) will take care of all CRUD operations for all use cases and JSP/HTML pages, regardless of underlying schema.***

**The benefits**

* Greatly reduce client-side development efforts, as we don’t need to create multiple use-case specific JavaScript files. Developers can now focus more on the coding of business logic.
* Reduce project maintenance cost, as the number of client-side JS files for web pages have been significantly cut down.
* SPA-style. For every use case, with respective set of entities/tables, CRUD operations will be performed in one single page, handled by a shared JS file.
* Autonomous client-side data validation. This JavaScript file (SearchEditForAll.js) will also handle all client-side two-way data binding and input validations.
* Dynamic search: by adding entity fields to search view, without changing model, a flexible and dynamic criteria search can be accomplished in no time, for retrieval of data.

**How is this achieved?**

The overall methodology is simple. For each use case, upon loading the CRUD page, it performs a one-time-only retrieval of page-specific metadata of the underlying dataset, which comprised of parent-child entity relationships, all column information and data validation rules, etc.

Based on the retrieved metadata, the page will carry out CRUD operations by interacting with server-side RESTful controller in JSON. These are all handled by one and only JavaScript file (SearchEditForAll.js) across all the client pages, therefore it is schema-agnostic, or more accurately, table-agnostic.

**The Server-Side Implementation**

1. For retrieving Entity metadata, a RESTful controller (AdminController) with a fixed URL has been implemented. Client page, upon loading, will pass a parameter string which is the entity object’s full class name, such as “com.jamesx.domain.employees”.
2. AdminController then uses this full class name to retrieve the following relevant entity information.
   1. “EmptyInstance”: Java reflection is applied to instantiate an empty instance of the entity, including related entities (such as child entity) on its entity graph.
   2. “EntityRestUrl”: This is the URL that entity will conduct CRUD operations upon. This is done by fetching entity object’s customized annotation (@EntityRestUrl).
   3. “MetaInfo": Metadata of the entity is obtained by traversing through the tree of entity graph, retrieving each entity’s metadata, such as property/column name, column label, data type, length, validation RegEx etc., by iterating through entity property’s annotations.
   4. “SmartTableInfo”: This is another customized annotation that is used to configure the display of search result on the web page.
3. The repository tier and service tier for CRUD are rather simple. A generic service class (GenericServiceImp) is implemented to respond to all CRUD requests, such as search, save, update, insert, delete, etc. Since it is of generic type, we need to add one line of configuration in our Java configuration file “WebAppConfig.java”, for the respective entity.
4. The implementation of entity’s RESTful controllers. An abstract controller (AbstractRestController) is implemented to be generic to interface with web page requests, such as search, save, insert, delete, etc., through exchange of JSON. As such, by extending this class, the implementation of each entity’s corresponding RESTful controller is very concise, only 3 lines of codes. The entity-specific RESTful controller needs to annotate the URL mapping, e.g. @RequestMapping(value = "/api/category/"), so that the client page would be able to perform CRUD through proper RESTful URL.
5. Finally, a router controller (IndexController) is created to map each webpage to corresponding URL. This web URL is client side browser URL, not related to RESTful URL.

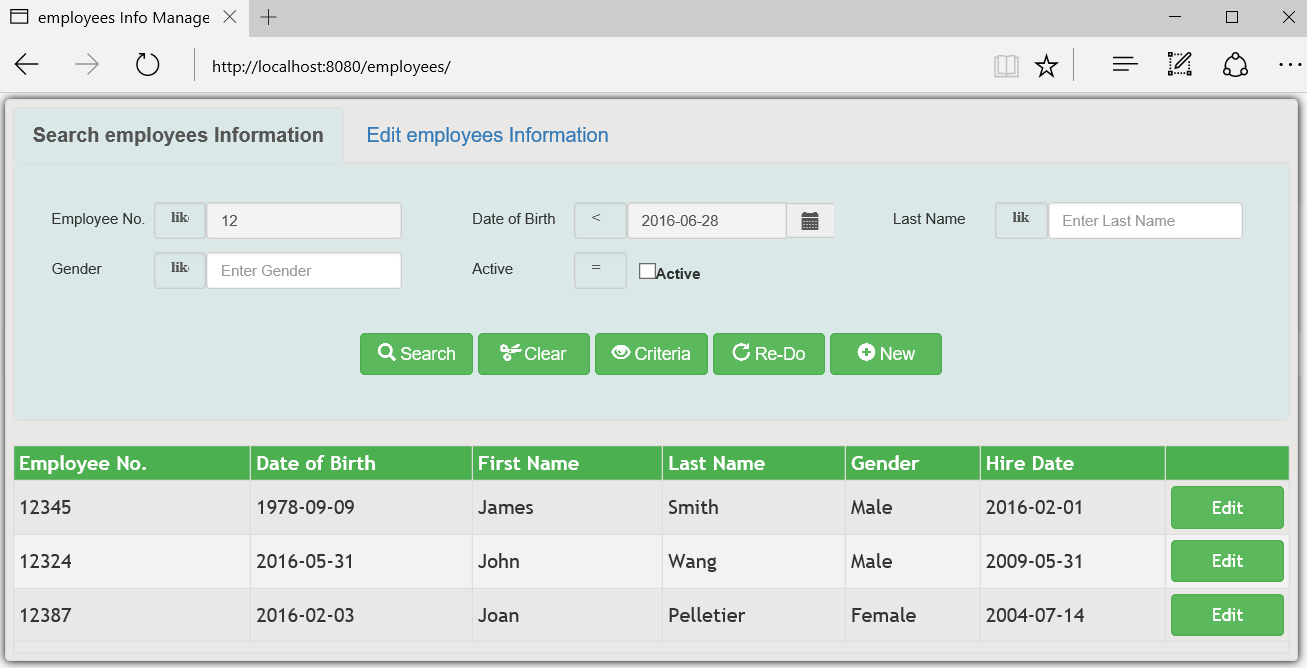
**The Client-Side Implementation**

In the context of Schema-Agnostic implementation, a user page needs to communicate with its corresponding RESTful URL. This is achieved by specifying underlying entity object’s full class name in each page’s header. One line of code such as the following will suffice.

<script type="text/javascript"> var MODEL\_CLASS\_NAME = "com.jamesx.domain.Employees"; </script>

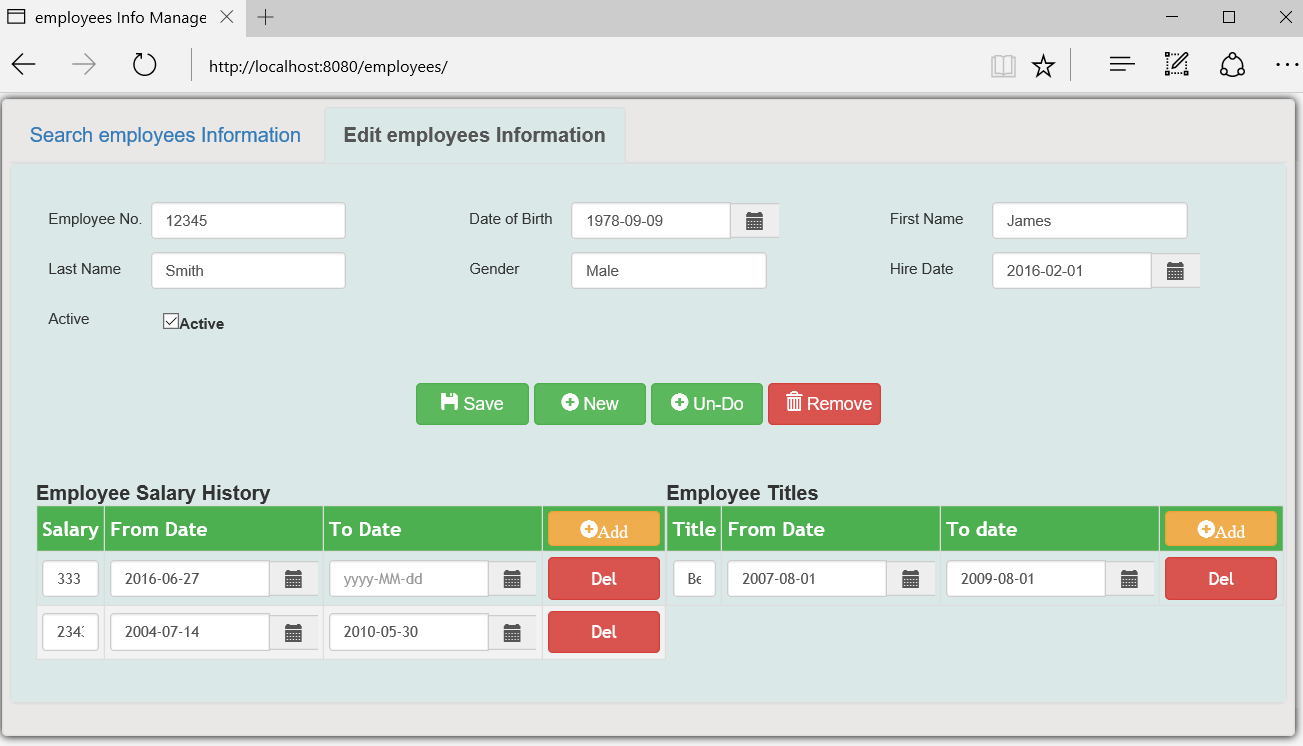
SearchEditForAll.js file should be included for each page. This JavaScript file is shared by all web pages, therefore it is **Schema-Agnostic**.

* **Search View implementation**

****

1. For each use case, search, edit, insert, delete operations are all performed within one page. The page consists of 2 tabs. On the search tab, user enters multiple criteria to search, the search result will be displayed in a format of paged table (AngularJS smart-table).
2. User is allowed to specify multiple criteria, all of which joined as “and” condition for search. Per each search criterion, user can specify conditions such as "equals to", "is greater than", "is less than", "NOT equals to", "is greater than or equals to", "is less than or equals to", "begins with", "contains", depending on the underlying data type. Those fields that are left blank will not be part of search criteria.
3. For performance reason, by default, only a minimum set of paged records be returned from server.
4. When the result is displayed, user can select any entry for Edit / Delete and navigate to Edit View. Clicking on the search view’s “New” button will enable user to insert a new record.

* **Edit View implementation**



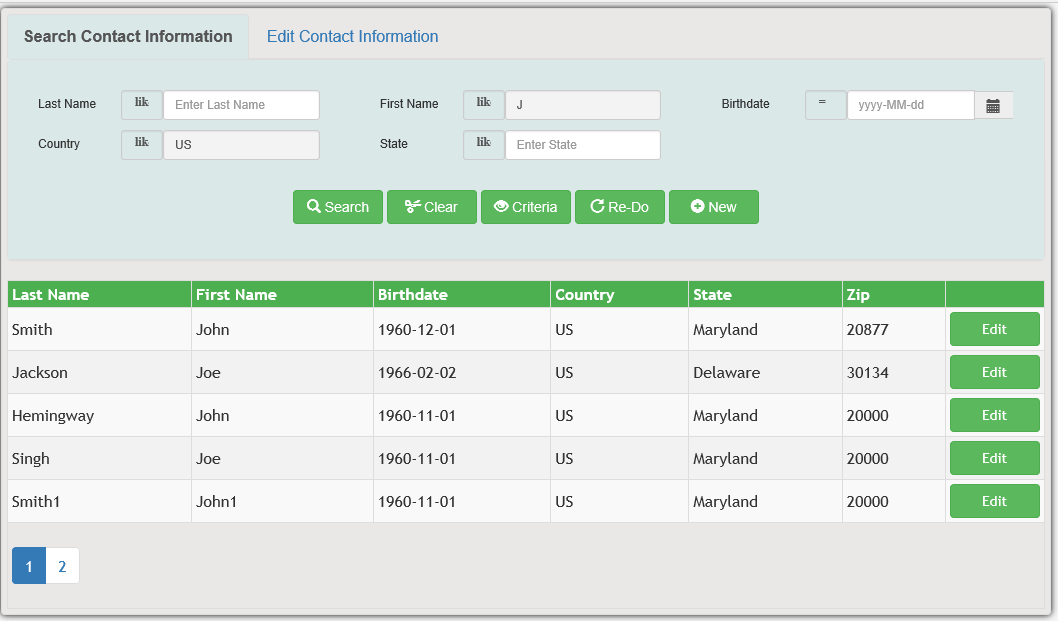
1. An entity ID is used to retrieve selected record along with its child entities. This is achieved by utilizing an annotation “@NamedEntityGraph” on entity class. This entity graph should be named in format of "graph.EntityClassName.eagerLoad", where EntityClassName is the page’s underlying entity class, so that our generic EntityManager defined in GenericServiceImp will load the entire entity graph properly.
2. The Edit View also implements input validation. SearchEditForAll.js file will use metadata to evaluate each input field’s value including those of child entities if any, when a save operation is performed. Validation information will be displayed if any invalid input is present. There is no need to implement data validations in web page’s HTML codes.

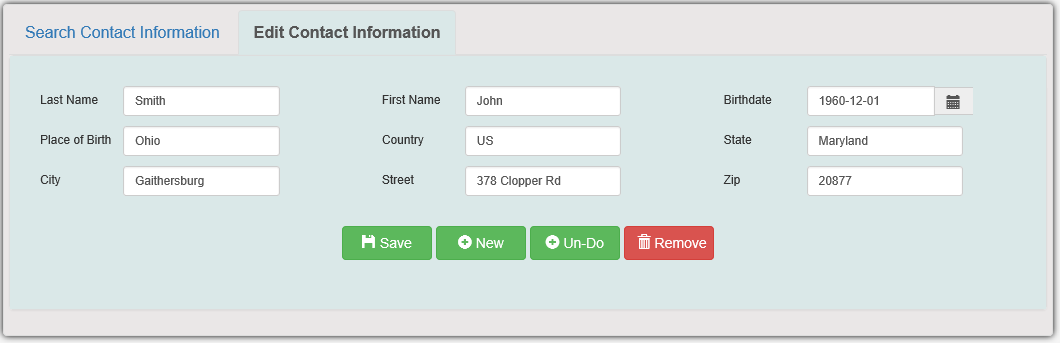
# Conclusion:

By implementing this framework, the effort of web application development can be significantly reduced on both server-side and client-side. The actual codes on both server and client sides only consists of a few hundred lines of codes, it is designed with performance consideration, and easily maintained.

**P.S.** Based on this framework, the author has also developed a “Spring MVC Automation Tool” that can generate codes for Model / View / Controller directly from any database schema. The following use cases are the results that are generated by the tool.

**Some other use cases (Generated by Spring MVC Automation Tool):**

Contacts Information



Order Information