Custom types and Data manipulation API for Kaeru

Introduction

The purpose of Kaeru is to enable people with little programming experience in building web applications. The users need to be kept away from the complications of normal programming language and database configurations. Having a sophisticated API to abstract database manipulation becomes a key to achieve the above goal. Providing the user with the power of custom types that are persistent, would complete the design of making database operations abstract from the user.

The rest of the documents will talk about how the custom types are built, how could it be used by the user and what data manipulation APIs are exposed to the user.

Design

The custom types would be defined by the user through the Kaeru UI. Upon submitting the type specifications from the UI, the custom type generation API kicks in. The user input would be fed into the API in a JSON format coming from the UI.

The API then parses the JSON and generates custom code to mimic the new data type on the client. This code is dynamically generated and resides only on the client machine.

The Custom type generated is a full-fledged type, encapsulated with data and methods to operate on data.

One more type would be created at runtime named <typename>Query which would focus on advanced DB manipulations.

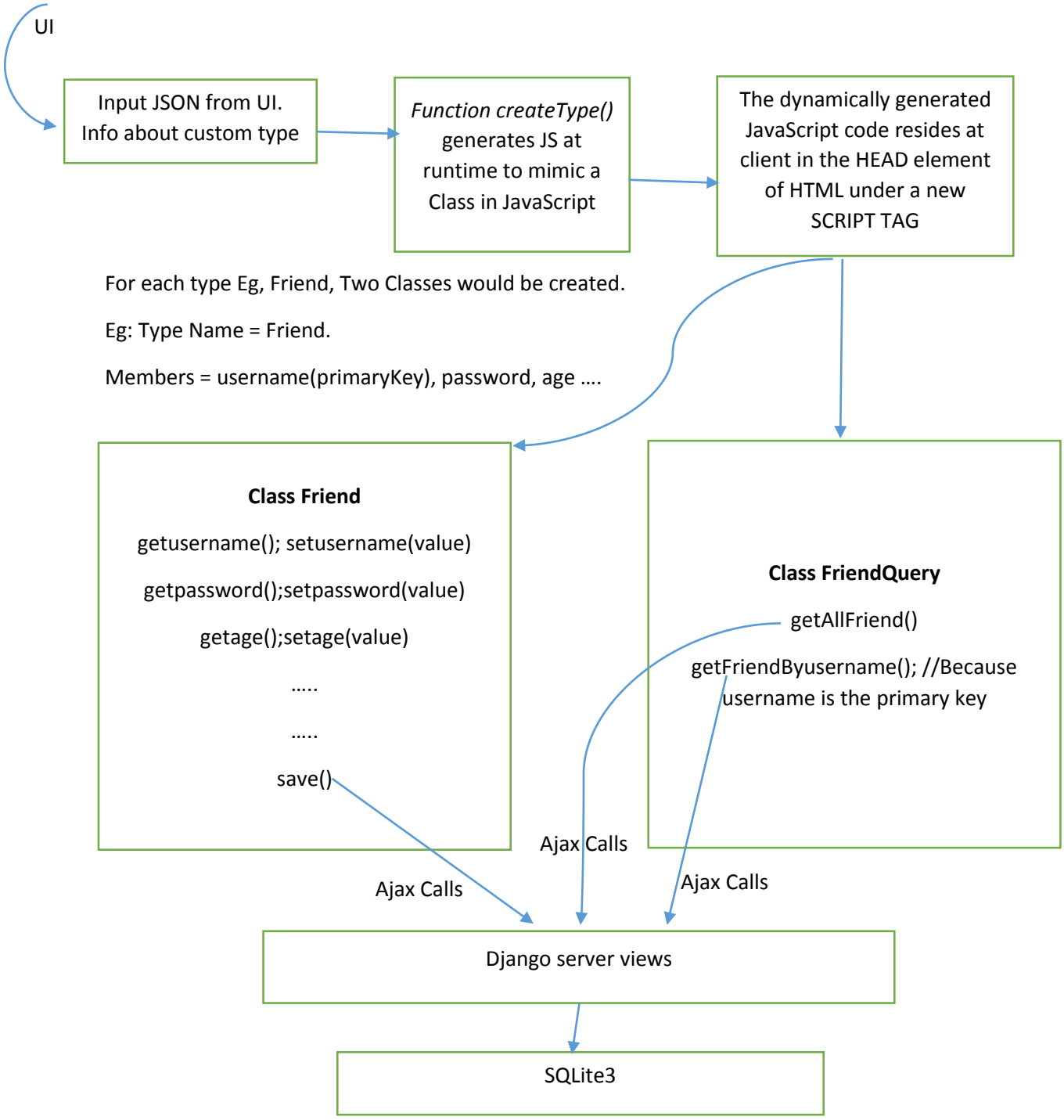
One of the key aspects of dynamically generating types JavaScript is the pain that comes with the loose typing of JavaScript. To minimize the downsides of the same, we have used “use strict” directive in JavaScript.

To make the custom types persistent, there is a *save()* method provided in every type created. The user can simple call the save method and the API shall take care of saving the data back to database. This way the user is kept completely away from the hassle of manually saving the data and configuring the database servers.

To achieve all the above features, the user page needs to make several calls to the server. Such calls would result in a page refresh and the entire dynamically generated type would be lost.

To avoid this problem, the API behaves like any other Single Page applications, making all server calls through AJAX. This is essentially difficult since there needs to be server calls made from within the dynamically generated API. Even there, we generate AJAX code to completely avoid full page refresh.

Flow


UI
Input JSON from UI.
Info about custom type
Function createType() generates JS at runtime to mimic a Class in JavaScript
The dynamically generated JavaScript code resides at client in the HEAD element of HTML under a new SCRIPT TAG
For each type Eg, Friend, Two Classes would be created.
Eg: Type Name = Friend.
Members = username(primaryKey), password, age ….
Class Friend
getusername(); setusername(value)
getpassword();setpassword(value)
getage();setage(value)
…..
…..
save()
Class FriendQuery
getAllFriend()
getFriendByusername(); //Because username is the primary key
Ajax Calls
Ajax Calls
Ajax Calls
Django server views
SQLite3

Functions/Components

To facilitate the above mentioned functionality, two other data structures were also implemented in this API which could also be used directly by the user. The List and the Map.

The List has the following methods.

1) *add(value)* – returns void, adds the element 'value' to the instance List
2) *remove(index)* – returns void, removed the element index = 'index' from the instance List
3) *get(index)* – returns value at the index.
4) *Size()* - returns the size of the list

The Map has the following methods.

1) *Put(key, value)* – adds the element with key in the map
2) *Get(key)* – returns the value of the item key
3) *isEmpty()* – returns Boolean. True if map empty, false otherwise
4) *size()* – returns the no. of keys in the Map

Backend methods

The following methods have been implemented in Django.

1) *def create_test_tables:* A test method to create tables at the time of creation of the custom type. This method could be removed later on as it is not directly linked to any of the key functionality related to custom types.
2) *save_user_data:* This method takes in table name and the query string from the request object and then inserts values from the query string into the SQLite3 table. A dummy select query is also run towards the end of this method to ensure if the data is saved properly or not. The same result is send back to the UI where it is printed using Ajax.
3) *get_all_type_data:* This method is called when the user called the generated API method getAll<type> of the <type>Query class. This method gets all the rows from the table and returns in a CSV format which is then read by the API at the client and converted in to a List of Custom type objects.
4) *get_type_data_by_key:* This method reads three values from the request. I)table name ii) primary key name iii) primary key value. Then the method queries the SQLite3 tables to fetch the data and send back as httpresponse in csv formatho. This response is then read by the API at the client and is converted in the custom type before returning.

   It is worth noting that methods 3) and 4) when called from the API, the user expects a return object. If the calling method from the API is Ajax, by the time the ajax call returns, the method exits and the user is sent back wrong data(which is mostly a null). To avoid that, calls to the methods 3) and 4) from the client have been made synchronous. This solves the problem.

Sample Code:

```
/* Sample code that could be used to test the APi
var friend = new Friend();
friend.setusername('MyFirstFriendName');
friend.setpassword('abc@123')
friend.save() // This will save the friend object back to the databse.

var friendQuery = var FriendQuery();
var friend_retrieve = friendQuery.getFriendByusername('MyFirstFriendName')
alert(friend_retrieve.getpassword()) // should return abc@123

var session = Session.createSession(10,'/Login'); // set the timeout to 10 mins
session.setAttribute('loginTime','5/6/2015 00:00:00')

alert(session.getAttribute('loginTime')) //Should return 5/6/2015 00:00:00

var friendList = new List();
friendList = friendQuery.getAllFriend();

alert(friendList.get(0).getusername()); // should return the username of the first friend created and saved in the database
*/
```

Test Cases

No	Test Case Details	Expected Result	Actual Result
1	Does the system read correctly from the JSON and create a type corresponding to the name of the type.	User able to create a new object of the type described. Eg. var fr = new Friend()	As expected

2	The custom type should have all the parameters and the getter-setter methods for the fields in the JSON string.	Parameters and methods created within the type. Use should be able to do the following. var fr = new Friend() fr.setusername('Test') fr.getpassword() …..	As expected.
3	NEGATIVE: How does the system behave if the JSON string has errors	No type should be created	As expected.
4	A save() method should be provided to the user for each custom type created. On calling the save method, all the attributes that have been 'SET' for the object should be saved back to the database.	The call to save method should save all the info back to the database. Assuming that the corresponding database table has already been created	As expected.
5	Use should be able to use the custom types even after call to the save function. Preserve the custom types even after a server call.	the state of the object should not get changed on save(). User should still be able to operate on the same object.	As expected. Using Ajax calls.
6	With every type, a new class/type with the name <typename>Query should be created.	Use should be able to create an object of the type <typename>Query. Eg. var fr = new Friend() var fr_query = new FriendQuery()	As expected
7	The query class should have two methods. 1)getAll<typename> 2)get<typename>By<key>	Both the methods should be created and should not return an error when the user tries to call them	As expected.
8	getAll<typename> should return a List of <types> if the results is more than 1 rows.	getAll<types> returns a List which contains items of type <Type> Eg. getAllFriend() returns a List of type Friend.	As expected
9	The method get<type>By<key> should return an object of type <type> since we are querying using the key	Get<type>by<key> should return an object of type <type> from the database.	As expected

10	User should be able to use the types even after using functions of the Query class.	Use to be able to create object of the types dynamically generated after calls to methods of the query class.	As expected. Using Ajax calls only
11	Check is the return values from the query class functions is correct	If a getter method is called on any parameter on an object returned by either getAll<typname> or get<typename>By<key>, a valid value should be returned	As expected
Test cases for backend python functions			
12	Call save_user_data, get_all_type_data, get_type_data_by_key without the tablename.	Return response should say table not found. Null check should kick in.	As expected
13	Call save_user_data, get_all_type_data, get_type_data_by_key with a tablename that does not exist	Return response should return an empty string.	As expected
14	Call get_all_type_data with a tablename with zero rows.	Returns a List object with length zero.	As expected
15	Call get_type_data_by_key with a key name that does not exist.	Returns an empty string if there is no such a column. If there is such a column(even if non key), should return the first value. This is kept for scalability purpose. A function like get<Typename>by<anycolumnName>() could be developed using get_type_data_by_key as base.	As expected
Test cases for List and Map types			
16	Test adding different types of elements in a list. And check if they are the same when get(index) is called	The get(index) returns the right value irrespective of the type of the object set for that index	As expected
17	Test if the size() function returned the correct length. Zero if the list was empty. Try calling size() right after creating an object of type List	Call to size() should return Zero if called right after object creation, the right value of the size of the list otherwise.	As expected
18	Test the get(index) with non-existent indexes. Check index overflow and underflow.	Call to get(index) with unknown index should return null.	As expected. Bug was identified initially, was fixed

			later by defining boundry checks on list indexes.
19	Test if the map get() and put() functions are working with normal values	Put(key,value) should save an item into the map with specified key. If a get(key) is called with the same key used, should return the correct 'value'	As expected
20	Test if the map put() does not allow values with duplicate keys.	Two calls to put('k1','test') and put('k1','test2') should not be allowed	As expected
21	Size() function on a Map should return the size of the Map. Check for the size right after instantiating the Map object.	Call to size() should return Zero if called right after object creation, the right value of the size of the map otherwise.	As expected.

Known Issues

1) The test routine *create_test_tables* doesn't work if the .db file is not created. The reason for this bug could be the way the .db file is attempted to be created in the code. However, if we run the same piece of code directly on the python interpretor through the command line, it works fine. There is a test python script included in the project called createTestTables.py. The limitation of this script is that it creates test tables of only one hardcoded type 'Friend'(since it does not communicate with the kaeru webpages).

   So if the data type creation needs to be tested without being dependent on the table creation module which is handled by the backend team, the above mentioned script should be run before testing the API. Also only use the type name 'Friend' in the type. If one needs to test with a different type name, please change the name of the type in the above mentioned test script and run the script before testing the API.