**HNV DYNAMIC WEB PROJECT INSTRUCTION**

Table of Contents

[**HNV Project overview** 2](#_heading=h.30j0zll)

[**Terms and rules in client side development:** 6](#_heading=h.1fob9te)

[**1.**](#_heading=h.3znysh7) **Rules** 6

[**2.**](#_heading=h.2et92p0) **Template** 6

[**3.**](#_heading=h.tyjcwt) **Save/Share project with GIT** 8

[**Variable App** 8](#_heading=h.3dy6vkm)

[**Request and response** 9](#_heading=h.4d34og8)

[**1.**](#_heading=h.2s8eyo1) **Request (Client → Server)** 9

[**2.**](#_heading=h.1ksv4uv) **Response (Server → Client)** 11

[**Controller File** 15](#_heading=h.4i7ojhp)

[**Start a view** 16](#_heading=h.1ci93xb)

[**1.**](#_heading=h.2bn6wsx) **Setup in client side** 17

[**2.**](#_heading=h.3fwokq0) **Setup in Server side** 23

[**3.**](#_heading=h.1mrcu09) **Some important functions in client side** 25

[**4.**](#_heading=h.2dlolyb) **Some important functions in server side** 28

[**Get data from Dom** 31](#_heading=h.25b2l0r)

[**LockTool** 32](#_heading=h.1jlao46)

# **HNV Project overview**

A HNV Project is a dynamic web project, which mainly uses these languages development: HTML, JavaScript and Java, based on Eclipse working environment.

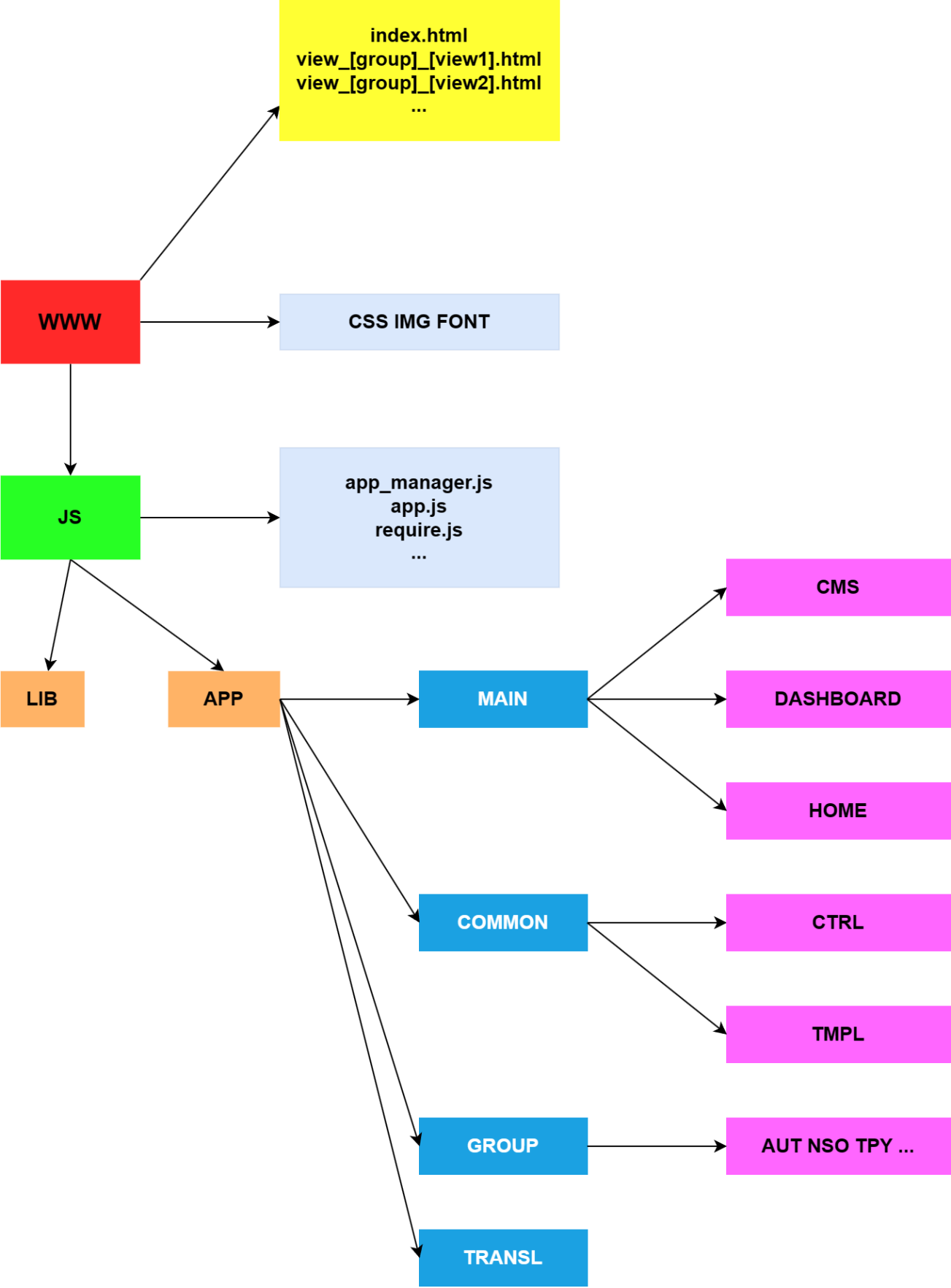
To approach a HNV Project, you need to be sure about your basic knowledge in HTML, JavaScript, JQuery and Java. HNV Project also uses Tomcat Server (current version is 8.5) and DBEAVER to manage databases. Otherwise, Handlebar, CSS, Bootstrap are frequently used also.

Some main parts of HNV Project are already configured (Database configuration, …) Hence, this document will show only important parts to help developers to create a view in a HNV Project configured.

What’s a view? A dynamic HNV Project web includes several different views. Every view is a child-page and has its own URL. By example:

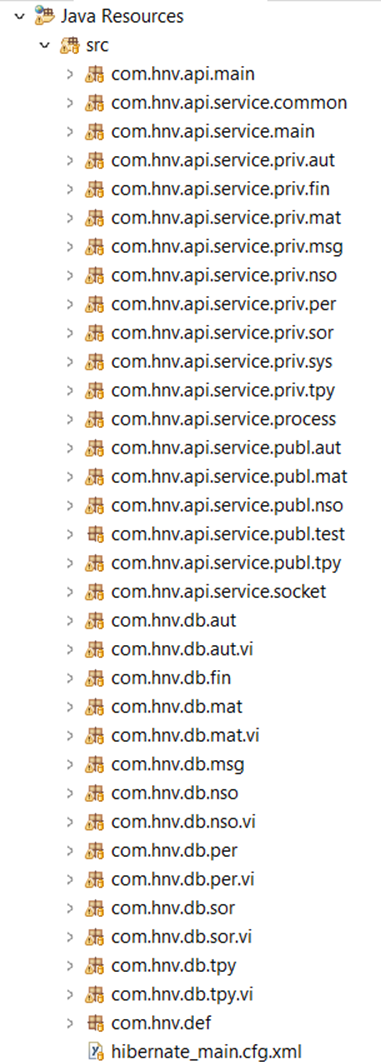


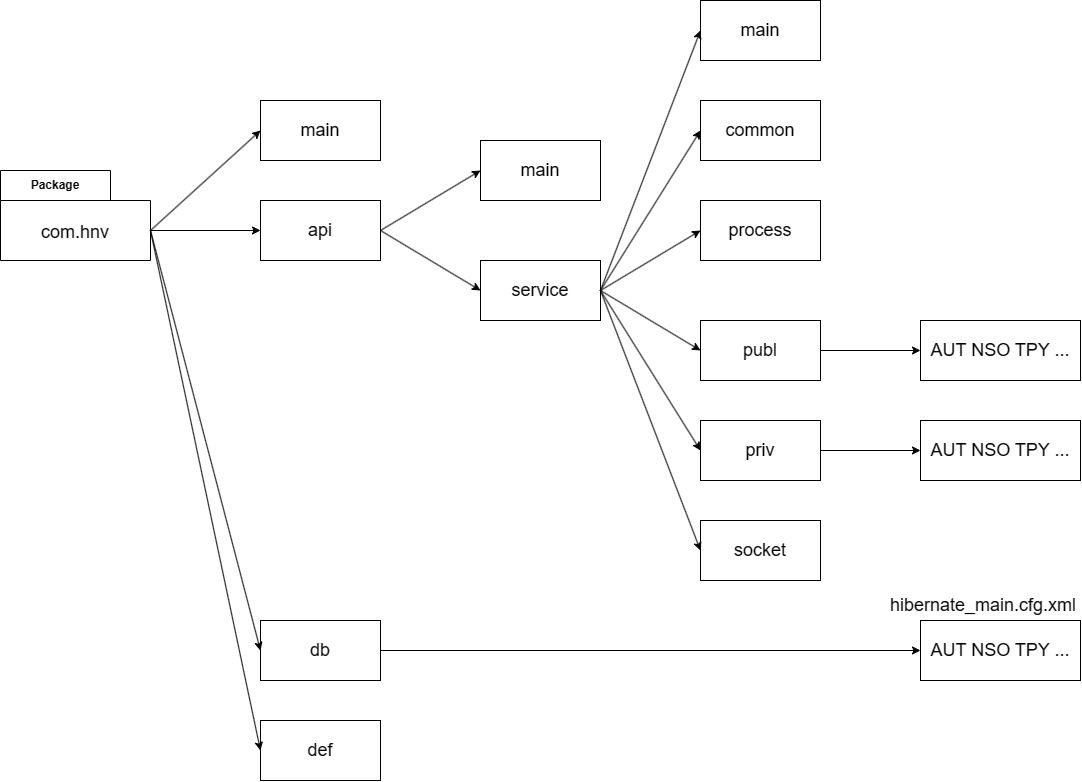
Before learning to create a view in HNV Project, take a look at HNV Project structure:



As a dynamic - web project, a HNV Project can be divided in two parts: **SERVER SIDE** and **CLIENT SIDE:**

* **SERVER SIDE**  includes all class Java, and locates in Java Resources folder:





* In the **SERVER SIDE**, there are three main parts:
* General configuration and define: (**main** and **def** package) the **def** package defines the url or name, the **main** package configures for the **SERVER SIDE**.
* Service: (**private**, **public** and **process** package) solves the main services of the model. Besides, we have the **main** package (input of the **SERVICE SIDE** and login service), the **common** package(some classes assist in executing services) and the **socket** package(service uses webSocket).
* Database: (**db** package) mapping with database through **hibernate\_main.cfg.xml** is automatically generated when creating databases
* **Client side** locates in WebContent folder and contains HTML, CSS, JavaScript and others resources files:

|  | In **client side**, there are some important files and folders that exist in every HNV Project:  **↦ css** - **font** - **img**: Resources folders, contains CSS Files, Images and Font  **↦ js/lib - js/app/common:** Library and Tools  **↦ app.js - app\_manager.js - require.js** : Initial, common definitions and config  In this side, developers will mainly work in these 3 folders, after configure a view (see 1.5 - Start a view)  **↦ Js > app > group > aut, nso, tpy,... > model > ctrl**   * All JavaScript files of this view   **↦ Js > app > group > aut, nso, tpy,... > model > tmpl**   * All HTML files (HTML elements) of this view   **↦ Js > app > group > aut, nso, tpy,... > model > transl**   * All JSon files of this view   Every view has 3 folders (one **controller**, one **template,** one **translate**) containing its files. It has also two other key files to function correctly:   * view\_<group>\_<name-view>.html (**view\_aut\_user.html** by ex.) * ent.js by ex |
| --- | --- |

# **Terms and rules in client side development:**

## **Rules**

* File Name rules

Every filename is preceded by the project initial.

With HTML File, use underscore (\_) to separate a word in file name.

With JavaScript File, uppercase of every word in name, file named without separate.

By example: **view\_aut\_user.html**, **EntTabDoc.js**

* Variable name rules

## **Template**

In HNV Project, templates are structured by empty div from parent to children. When controllers call to show the template content, they will call the id, and call the template HTML corresponding to this ID; then include this HTML content to the empty div.

A HTML content included may contain other empty div tags with specific ID and other template HTML files. These div tags will be called by other controllers, in a very similar way. Every part of a view will be established like this.



Here are some fixed div with ID (bellows are fixed, don’t change it because all defined id were called in others places on HNV Project)

| Index view (**view\_aut\_user.html** by ex.) | |-- Login\_Content  |----------------------Register\_Content.html  |----------------------CMS\_Select\_Lang.html  |-- Layout wrapper  |----------------------DBoard\_Main.html  |------------------------DBoard\_Sidebar.html  |------------------------DBoard\_Header.html  |------------------------DBoard\_Notification.html  |-- Right bar  |---------------------DBoard\_Rightbar.html  |-- Rightbar\_overplay |
| --- | --- |

## **Save/Share project with GIT**

* Before coding: Pull from Upstream.
* After code: Commit → Pull from Upstream → Push to Upstream
* Never delete the scope of others. Put it in comment if you don’t use this scope anymore or just change it.

# **Variable App**

* In the HNV Project, App is a very important **global variable** that appears all along the project building process. **App** is an object that contains almost definitions, data, functions and can be called anytime in any JavaScript file.
* Every view has its own app\_view.js file (**app\_com\_man.js** / **app\_com\_contact.js** by example). It inherits all definitions made in App.js, and extends this variable by its own definitions. (example: Templates, Router, … of App were declared in app\_view.js).
* With properties of a global variable, we can store data in the App and retrieve it all the time and call functions, data stored on it.
* An easy way to store data response from server to variable App, please check (4. Request & Response).

# **Request and response**

****

## **Request (Client → Server)**

* Every time your website loads, it will look for all these files to display them to the user. For each file, your website needs to make a request to the server the file is stored on, asking it for a copy to give to the person viewing your website. In this project, all request to server are formed in a same structure as below:



| **ref = {}** | **Ref** is an object that contains all necessary information to make a request. Some must-have element on ref:   * Directory of service that treat this request, identified by **req\_gl\_Request\_Content\_Send(pr\_SERVICE\_CLASS, pr\_SERVICE\_NAME)** * Type and Status: **ref.tpy01, ref.stat** |
| --- | --- |
| **fSuccess = [f01, f02, ...];** | → **fSuccess** contains one or more than one functions (**f01**, **f02**... - callback functions) to be executed when response data returned successfully. In this example, function **doShowContactListByGroupId** with none parameter will be called when data returned from server.  Notice: Callback function will be called when a connection client - server is established successfully, even if server return an error code. |
| **fError** | → **fError** contains one or more than one functions to be executed when an error occurred when establishing a connection to server. |
| **req\_gl\_Datatable\_Ajax\_Dyn (div, App.path.BASE\_URL\_API\_PRIV,App.data["HttpSecuHeader"], fileTransl, colConfig, ref, fError, undefined, null, undefined, do\_bind\_list\_event, dataTableOption) ;** | → This is a common ajax request to server. Only three variables (ref, fSuccess, fError) change. |

* Particularly case: Request a dynamic list: See (9. Dynamic List).
* Function to store response data from server to global variable App:  **App.funct.put, defined in App.js**

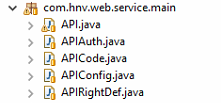


With callback function below in ajax request structure, data response from server will be stored in **App.data[“response”]:**



## **Response (Server → Client)**

* **API -** Common port for all request treatment



API is a main service, a port that check all request sent from client, check and redirect to related services. In this project, API is located in service.main, with 5 classes: **API**, **APIAuth**, **APICode**, **APIConfig**, **APIRightDef**. Two of these classes you may need to pay attention:

| //APICode File  **public** **class** APICode {  **public** **static** **int** *SV\_CODE\_NOTHING* = 0;  /\*--------------------Error code ------------------\*/  **public** **static** **int** *SV\_CODE\_ERR\_AUTHEN* = 10000;  **public** **static** **int** *SV\_CODE\_ERR\_RIGHT* = 10001;  **public** **static** **int** *SV\_CODE\_ERR\_USER\_DESACTIVATED* = 30000;  **public** **static** **int** *SV\_CODE\_ERR* = 10002;  **public** **static** **int** *SV\_CODE\_LOCK\_ERR* = 10003;  **public** **static** **int** *SV\_CODE\_INACTIVITY\_ERR* = 10004;  **public** **static** **int** *SV\_CODE\_USER\_ERR* = 10005;  **public** **static** **int** *SV\_CODE\_USER\_TP\_ERR* = 10006;    /\*-------------------Response code ----------------\*/  **public** **static** **int** *SV\_CODE\_API\_YES* = 20000;  **public** **static** **int** *SV\_CODE\_API\_NO* = 20001;  **public** **static** **int** *SV\_CODE\_API\_ERROR* = 20002;  } | This class defines all codes based on result of request treating process in services. Theses codes are the same as **sv\_codes** that defined in **APP[‘const’]**.  **SV\_CODE\_API\_YES, SV\_CODE\_API\_NO** and **SV\_CODE\_ERR** are usually used to check the response data or to debug. |
| --- | --- |
| //APIConfig File  **public** **static** **final** String[] ***PACK\_SERVICE*** = {"com.hnv.web.service.utilities"}; | This class defines the package that contains all classes related to requests from client side.  If you create a package or if you locate a service outside declared package in API Config, API could not find and redirect your AJAX request in a good destination. |



* Structure of a service (in **service.utilities)**

Service is executed when and only when user checking process ends without problem. There are two step to verify user and its right:

* Check if user exist, logged or session not null.
* Check if user has right to execute this request.

In every check-step there is a return statement (**API.doResponse**) with **sv\_code** corresponding. When **sv\_code** has been sent to client side, it could be found in **sharedJson.sv\_code**.



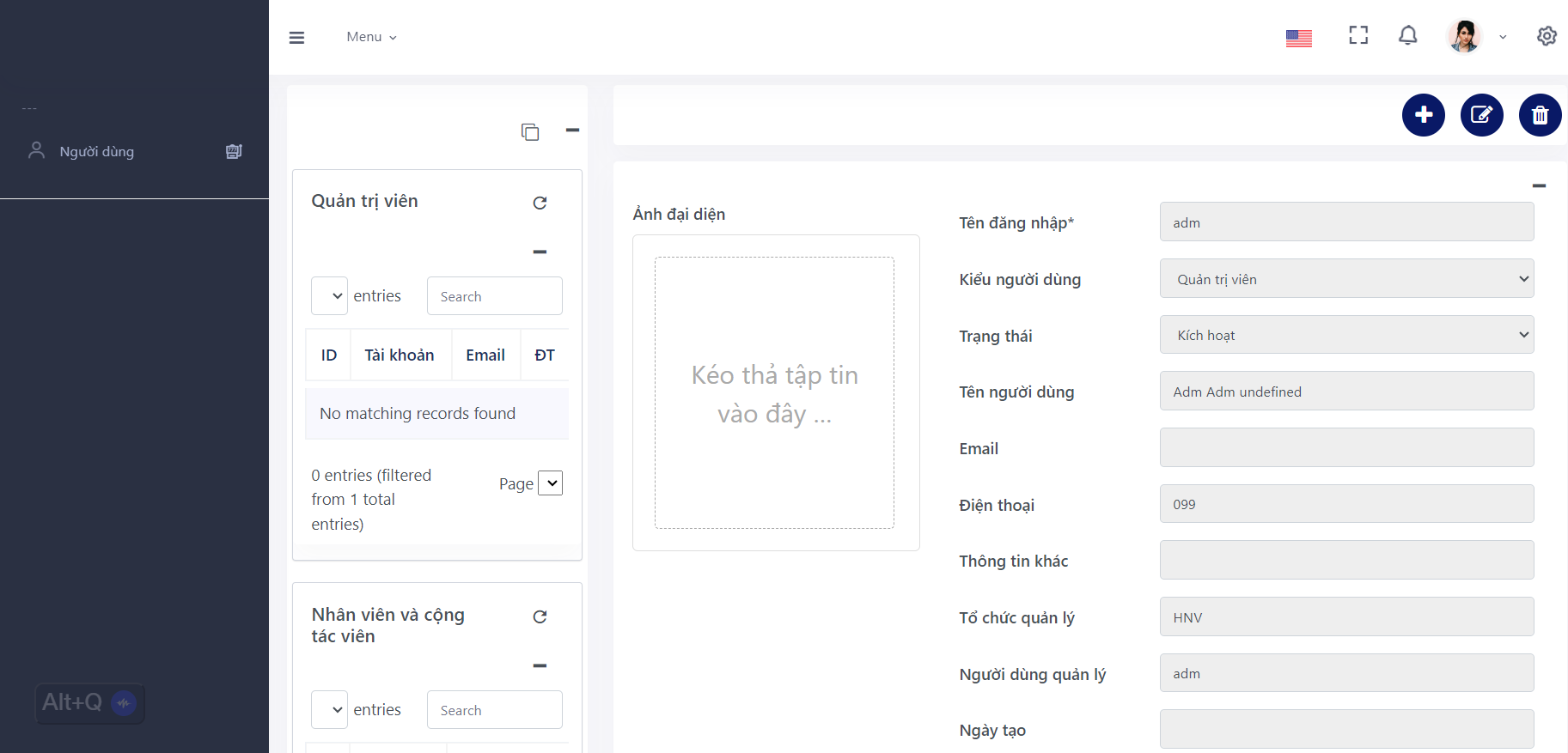
**return\_data** is data that service returns to client side after do service. This can be a list object, an item, a number or null (nothing). In client side and locates in **sharedJson.res\_data.**

# **Controller File**

Controller File is a JavaScript File, whose role is import template, send and receive data, show view and bind event handler for elements that it controls. Every controller file has a same structure:



# **Start a view**



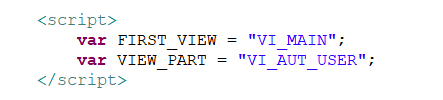
| **view\_aut\_user** is chosen for example in this section.  **Project Detail:**  **view\_aut\_user** is a view in HNVProject whose role is allow contact management. In this view, user will see the list of group contact in left panel.  Firstly, when the page finished loading, user will see only in left panel list of group contact (1). When user click on an item in (1),contact details appear in part (2) and some buttons to add,modify, delete contact.  There are Every part is a div with ID, and has its own template HTML. See table below: |
| --- |

| **Common** | **Main.html** | **Main.js** |
| --- | --- | --- |
| (1) | **List.html** | **List.js** |
|  | **List\_Typ\_Cand.html** |  |
|  | **List\_Typ\_Recr.html** |  |
|  | **List\_Typ\_Adm.html** |  |
|  | **List\_Typ.Agent.html** |  |
|  | **List\_Stat\_Deleted.html** |  |
|  | **List\_Stat\_New.html** |  |
|  | **List\_Stat\_Review.html** |  |
| (2) | **Ent.html** | **Ent.js** |
|  | **Ent\_Header.html** | **EntHeader.js** |
|  | **Ent\_Header\_Time\_Main.html** | **EntHeaderTimeMain.js** |
|  | **Ent\_Header\_Time\_View.html** | **EntHeaderTimeList.js** |
|  | **Ent\_Header\_Time\_Show.html** |  |
|  | **Ent\_Header\_Time\_New.html** |  |
| (2.1) | **Ent\_Btn.html** | **EntBtn.js** |
|  | **Ent\_Btn\_Sub.html** |  |
| (2.2) | **Ent\_Tabs.html** | **EntTabs.js** |
|  | **Ent\_Tab\_Doc.html** | **EntTabDoc.js** |
|  | **Ent\_Tab\_Role.html** | **EntTabRole.js** |
|  |  |  |

## **Setup in client side**

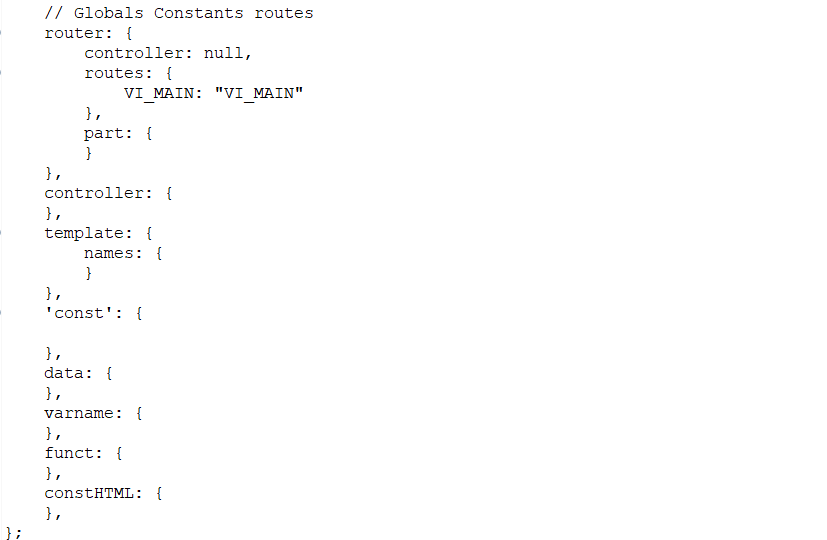
| **Files**  view\_aut\_user.html  app\_manager.js  Main.js  Main.html  **Folders**  WebContent/www/js/app/group/aut/user/ctrl  WebContent/www/js/app/group/aut/user/tmpl |
| --- |

* File view\_aut\_user.html (in WebContent folder): Declare FIRST\_VIEW , VIEW\_PART and script file of this view:





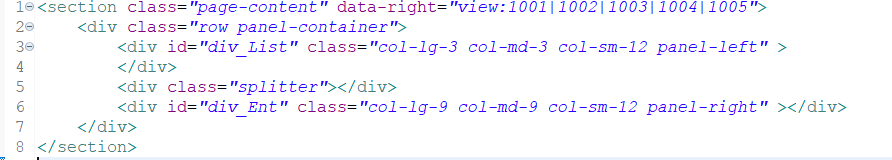
* File app\_manager.js (in WebContent/www/js folder):



* Declare a router (based on FIRST\_VIEW).
* Declare template names to store the HTML content of each part.
* Declare a const to store the application's data content.
* Declare a controller to manage application controls.
* Declare a funct to store the application's function.
* Declare a varname to store the application's variable or variable name.
* Declare a constHTML to store the application's code in static HTML.
* Create 2 folders to contain JS Files and HTLM Files. Create inside each folders necessary files

|  | Others HTML Files (templates) could be added in template/contact folder for later use. |
| --- | --- |

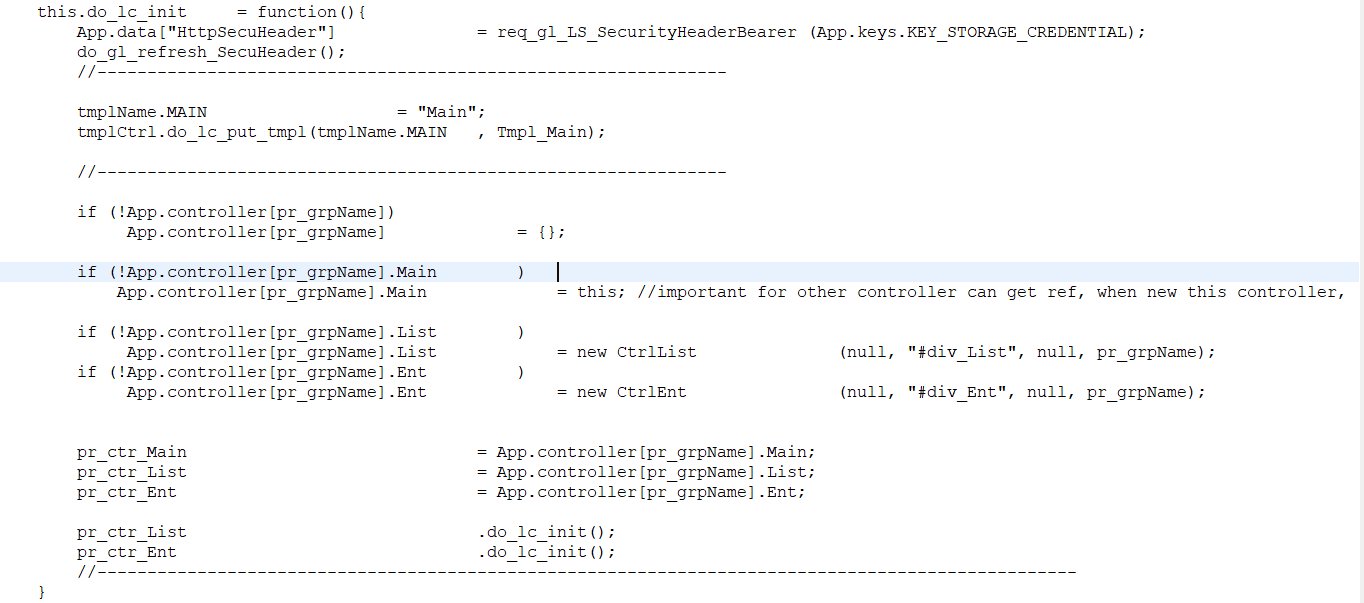
* File Main.html: Describe 4 div position, corresponding to 4 parts of this view, with ID:



Notice: To divide main content to left side and right side panel, please use this **fixed code**:

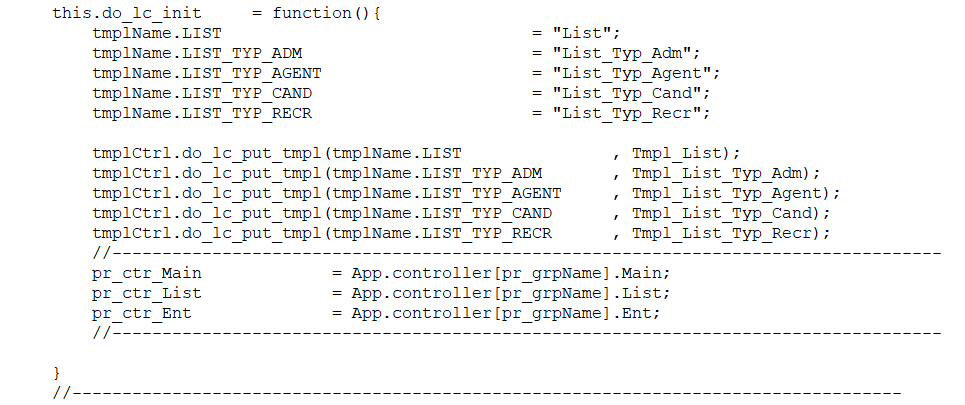
<div class=*"row panel-container"*></div>

* File Main.js
  + Show template Main, then introduce all child-templates to **App.data** for later use.
  + Call two controller of two child-templates with **do\_lc\_init()** function.

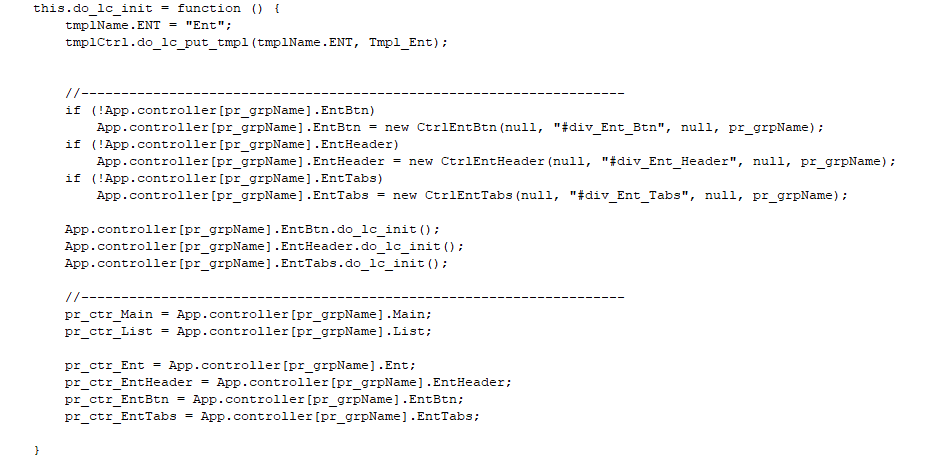


* + As we known, function **do\_lc\_init()** is to prepare templates and some initial configuration, binding event,…

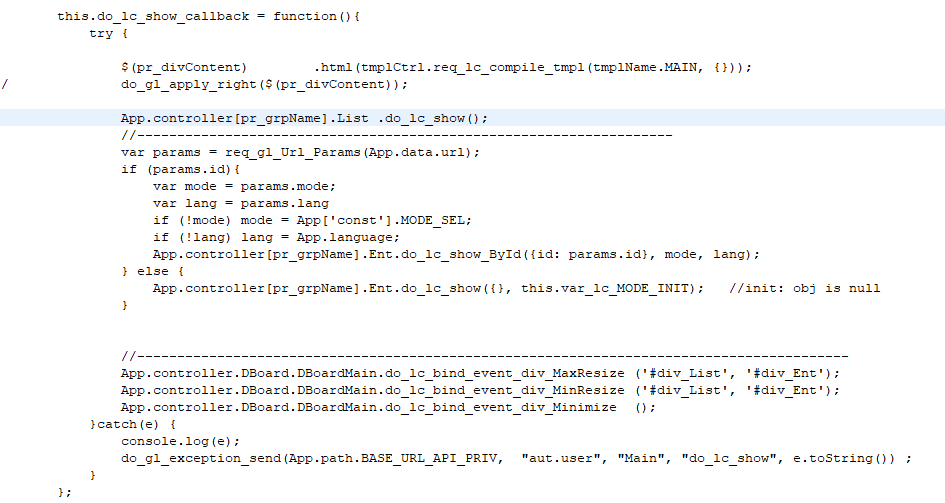
**list.js**

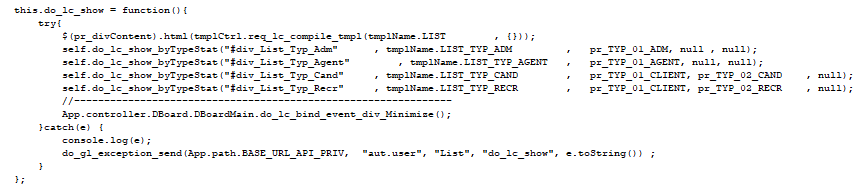


**Ent.js**



Launch **List.js** to show the group contact list. The function **do\_lc\_show()** was called by the function do\_lc\_show\_callback() in class Main.js.

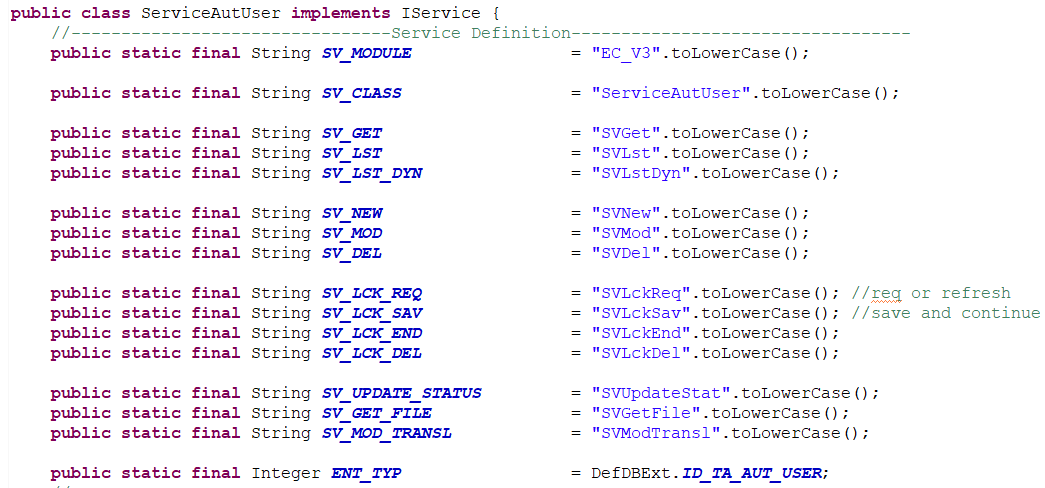




## **Setup in Server side**

| **Files**  ServiceAutUser.java (package com.hnv.api.service.priv.aut)  TaAutUser.java (package com.hnv.db.aut)  ServiceComContact.java  ServiceComContactGroup.java |
| --- |

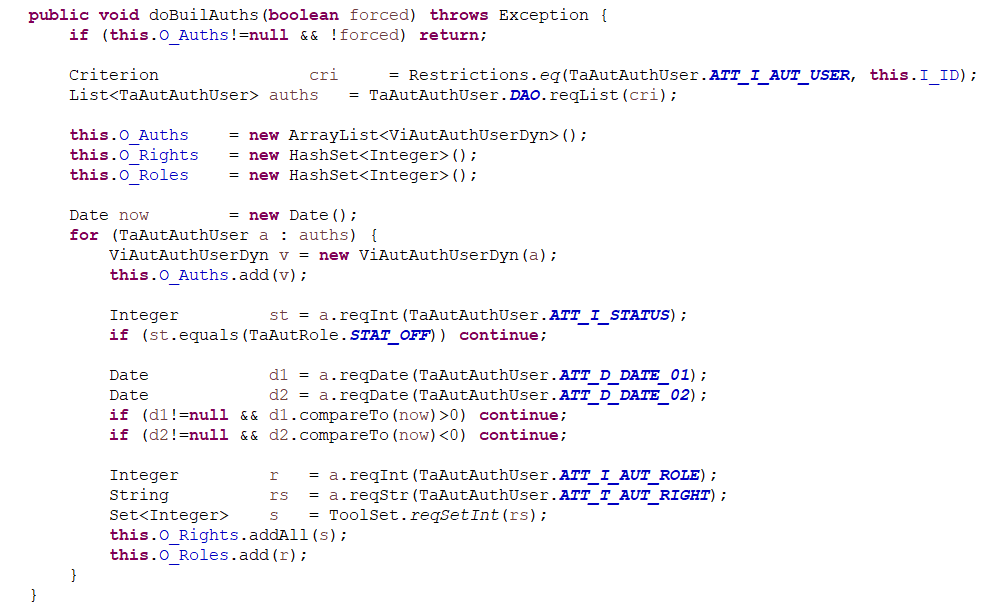
* Define server class and services in this server. Make sure that this class is located in declared **pack\_service** in **APIConfig** (see 4 b.) Frequently services are: Add, Modify, Delete, Get list, …

****

* Verify entity’s update permission in package com.hnv.db.aut By example: To allow read, add, update, delete a contact, in TaAutUser.java, change corresponding params of RIGHTS to true:



* Check carefully if matching column of entity is nullable or not. That’s important because you have to pack in **ref** all required data to send to service. If service, in data mapping progress, cannot recognize data for a not null column, this will cause bug.
* If an entity is related to another entity (A contact is in a group contact by example), sometime a **transient object** is necessary. In this case, when we have already a Contact entity, we can call to build its groups that contain this contact, as following:



## **Some important functions in client side**

* Show pop-up message success or error



* Show multi-option message box with button (for a form, a confirmation, …)



* Setup CSS, add search, sort option for a list, …



* Function to bind event for every row in a table (a list):



* Display translated message (string) in selected language (translation defined in JSON files on library **WebContent/www/js/lib/i18n**): Function **$.i18n("message")**

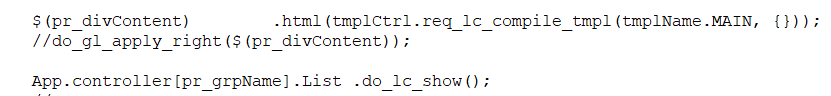


* Put templates to variables. Show HTML content mixed with data (see more in Handlebar)

Function: **tmplCtrl.do\_lc\_put\_tmpl(template\_name, file\_HTML\_defined);**

Function: **tmplCtrl.get\_compile\_tmpl(template\_name, data);**

****

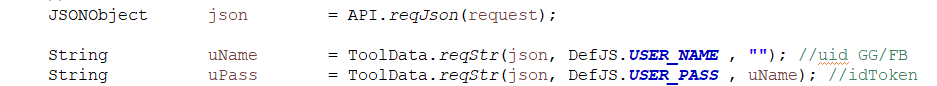


With this function, we can add any HTML external content to a div. Theses HTML Files URL need to be declare only in controller that calls them; and variables that contain these HTML content need to be declared also in App\_view. In this project by example, we need a form to add new contact to group. Then, we declare new template name in App\_Com\_Contact.js and import to controller files:

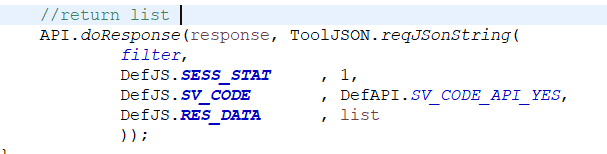


## **Some important functions in server side**

* Parse data from REF. Data parsed is a string. Function: API.reqJson(request)(keyword);



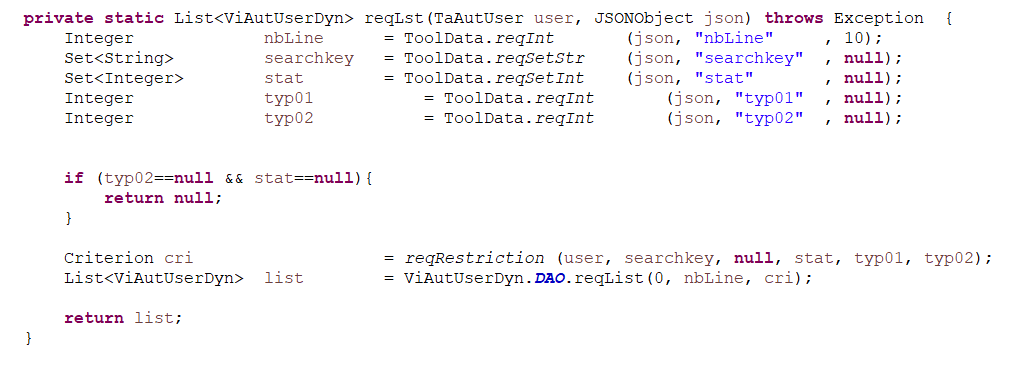
* Returns to client side: Function API.doResponse(response, stringJSON);



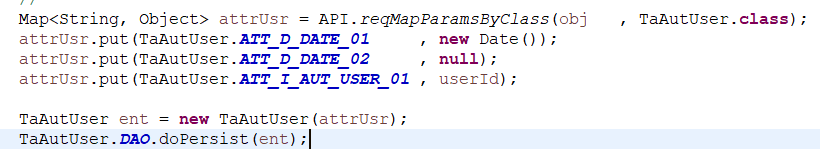
* Request List Data: There are several ways to get a list of entities in service: Get a list by SQL or by define function DAO.



///



* Map Entity



* doPersist / doMerge / doDelete : Functions to add, update and delete an entity from database.

Note: If function history record is activates, doPersist, doMerge and doRemove extend one parameter: **userID** to record this action.







* Restrictions

Restrictions is a function that permit filtering an entity’s list by one or plural criteria in server side. Bellows are some examples:



# **Get data from Dom**

This is a function that allow us quickly get all data input from DOM (a form) and put into ref, ready to make an ajax request.

To use this function, we need add class **objData** to every input of selected DOM.



Here is request to get input data from this form to REF. If a required input is empty, REF will be set null.



# **LockTool**

LockTool is a tool to prevent conflicts occurred when two or more users access in a same time to an object / entity. To process modifying or delete an object, we need sure that this object is not currently accessed by others.

Concept: Before moving to editing / deleting an object, a request will be sent to server to know if this last is locked or not. If object is currently locked (that means another person is now making change in this object), request will be refused. If not, object will be temporary locked and request is allowed.

After modifying / deleting or no more access to selected object, this object will be released and lock record will be delete.

This will explain how to setup a lock before modifying an object:

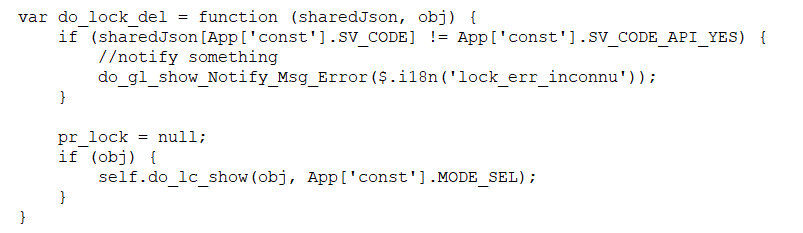
* Request a lock for selected object to modify.



* If object is not locked, service will lock it, and return **sv\_code** 20000 and a form to modify appears. The lock information will be put to App.data[lock\_group\_object] for unlock later. Else, service returns **sv\_code** error.



* After modify / cancel modify, selected object need to be released. Remember to recall **lockId** from **App.data[lock\_group\_object]** to unlock.



* To delete an object, we do the same thing, but check lock object and delete object are in the same service, so just simply make this AJAX request:

