**Internal Organisation Management System**

***A PROJECT REPORT***

***Under the guidance of***

***Details***

******

*Submitted By*

**Prince Arora**

**1402002098**

******

*In partial fulfilment of the requirement for the award of the degree*

**Of**

**Master of Computer Applications**

**Department of Information Technology**

****

February, 2017

****

**BONAFIDE CERTIFICATE**

**Certified that this project report titled “**Internal Organisation Management System**” is the bonafide work of “**Prince Arora, 1402002098**” who carried out the project work under my supervision.**

**SIGNATURE**

**GUIDE**

**<<Name >>**

**Abstract**

**Companies are following process for all the new joining. One of them is familiarising new employee to their internal system, which takes lots of time and most of the time it gets very confusing for the employees. This document/ Project is designed to make this process easy and self-understandable.**

**Some of the feature provided:**

**Email Client:** Email client to check or compose internal emails.

**Chat Box:** Chat box to communicate with any employee in team and share files.

**Project management:** Project management to check project state and milestones.

**Employee management/ Team management:** Manage team/ employee.

**Assets management:** Manage office assets and assets assigned to any employee.

**Notifications:** Notifications for events and other actions

**Table of Content**

|  |  |  |
| --- | --- | --- |
| **Chapter No** | **Title** | **Page No** |
|  | **Abstract** | **1** |
|  | **List of Figures** | **2** |
|  | **List of Tables** | **3** |
| **1.** | INTRODUCTION   * 1. Introduction   1.1.1 Problem Statement  1.1.2 Resolution  1.1.3 Purposed System | **5** |
| **2.** | REQUIRMENT SPACIFICATIONS  2.1 SRS Diagram  2.2 E-R Diagram  2.3 Class Diagram  2.4 Data Flow Diagram |  |
| **3.** | SYSTEM DESIGN  3.1 Modularization  3.1.1 Dojo Components  3.1.2 REST Services Components  3.2 Data Integrity & Design  3.2.1 Data Security  3.2.2 Design  3.3 Procedural Design  3.4 User Interface Design |  |
| **4.** | IMPLEMENTATION  4.1 Dojo Implementation  4.2 REST Implementation |  |
| **5.** | RESULTS  5.1 Input and Output Screens |  |
| **6.** | Limitations and Future Application of the project |  |
|  | Conclusion and Bibliography |  |
|  |  |  |

INTRODUCTION

1.1 INTRODUCTION

1.1.1 Problem Statement

After hiring any new employee in any organisation, its organisation’s responsibility to give proper training to the employees to make them understand all the internal portals and processes that is needed. Most of the time it’s hard for employee to understand all at once because most of the time these processes are divided into different portals and are not divided according to the user’s rights.

Working on different portals to perform these action is time consuming and also hard to understand for new employees and takes to for them to familiarize with the system. If we have all the options at same place it will be self-explanatory for these employees and also they can manage their dashboard according to their convenience. For example a manager can keep the project management part option and can hide other sections and they can easily access those from same place when needed. If we keep these information at same place then all the employees will know what they need to work on and what is the status for their project and also other events and updates that are mandatory for users. And if they need to find something they can get it from this dashboard.

Most of the time organisations are dependent on mails or any chat system for their internal communications. And it gets confusing for employees if they have to search for anything important and they are not sure how it was communicated with them either mail or chat. If we can integrate these both at one place then it will be easy to them to search and also manage their meetings and schedules.

1.1.2 Resolution

Resolution for the above issue if to provide all these functionalities at one place and give appropriate rights to user so that can manage the dashboard and they are aware of their rights. All the updates and notifications can be found at same place and all the communications will be at same place. A search that can find things from all over their dashboard will be defiantly helpful for them as they don’t have to look for thing in different places where they are not sure where exactly they will find it.

If all the activities that are mandatory for an employee can be handled at one place then it will be easy for new employees to pick up these processes and will save lots of time to them. And Since it is a web application it can run anywhere.

1.1.3 Purposed System

The system that is purposed here is a single page application that will contain section/ Modules based on user’s permissions and roles. These modules can include Mail client, Chat Box, Project management, Asset management, Team Management and Notifications.

There are lots on web toolkits and frameworks to create single page application (AngularJS, ReactJS, ExtJS etc) one of them is Dojo Toolkit. Dojo provides modular ways to create a single page web applications. It provides some predefined components also that can help creating an application ex: fx to add animation and effect in html, tree to add tree structure in html page.

This single page application will be communicating with a REST services that will be doing all the persistence operations and the performing all the tasks based on business logic defined. These REST services will be communicating with Data storage (MySql).

* 1. Architectural Diagram.

**Browser with single page application**

Employee

**MySQL**

**Database**

**REST Services**

REQUIRMENT SPECIFICATIONS

2.1 SRS Diagram

Client 2

Client 3

Client 1

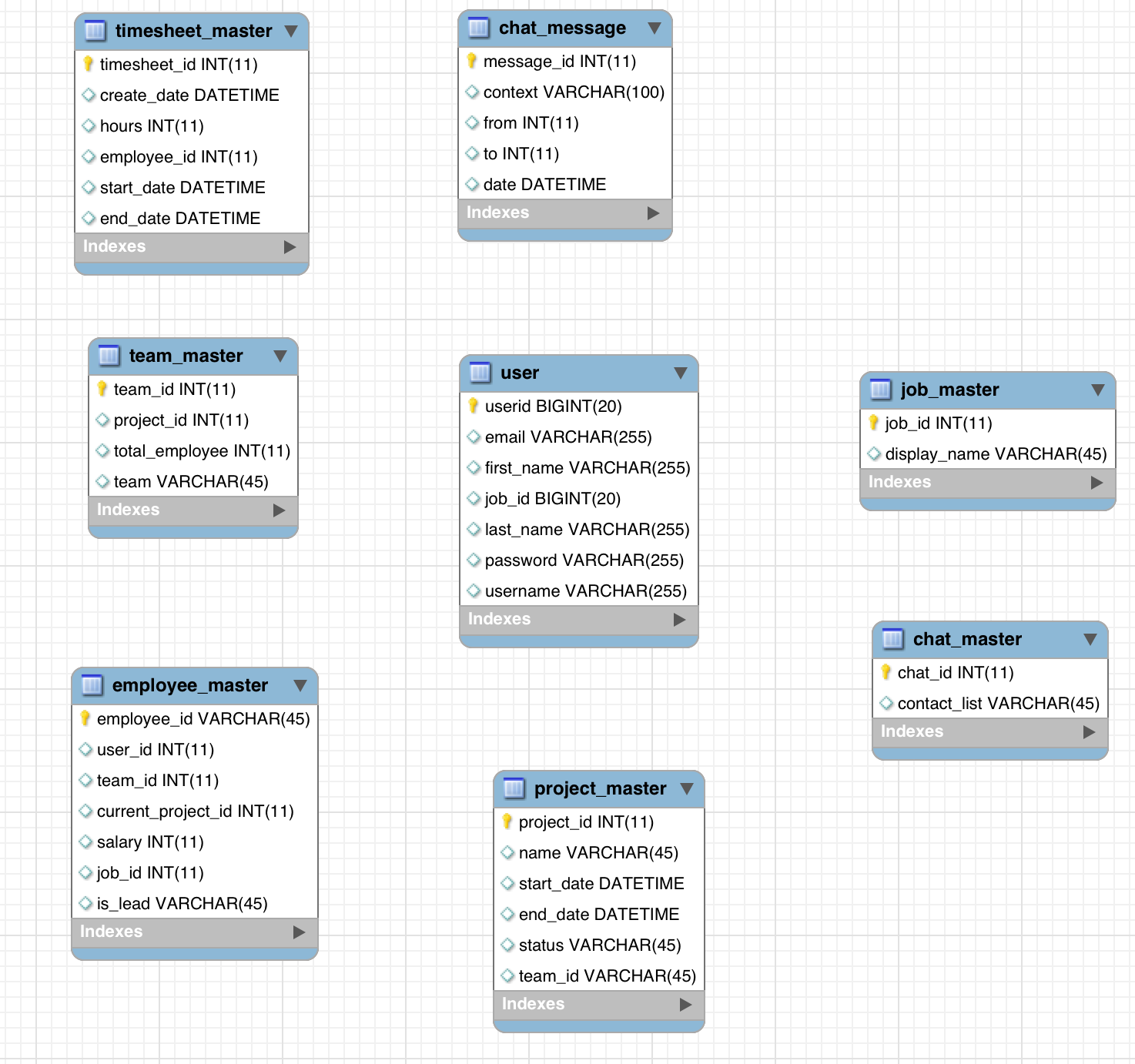
**MySql Data Store**

Dojo XHR request to use REST services

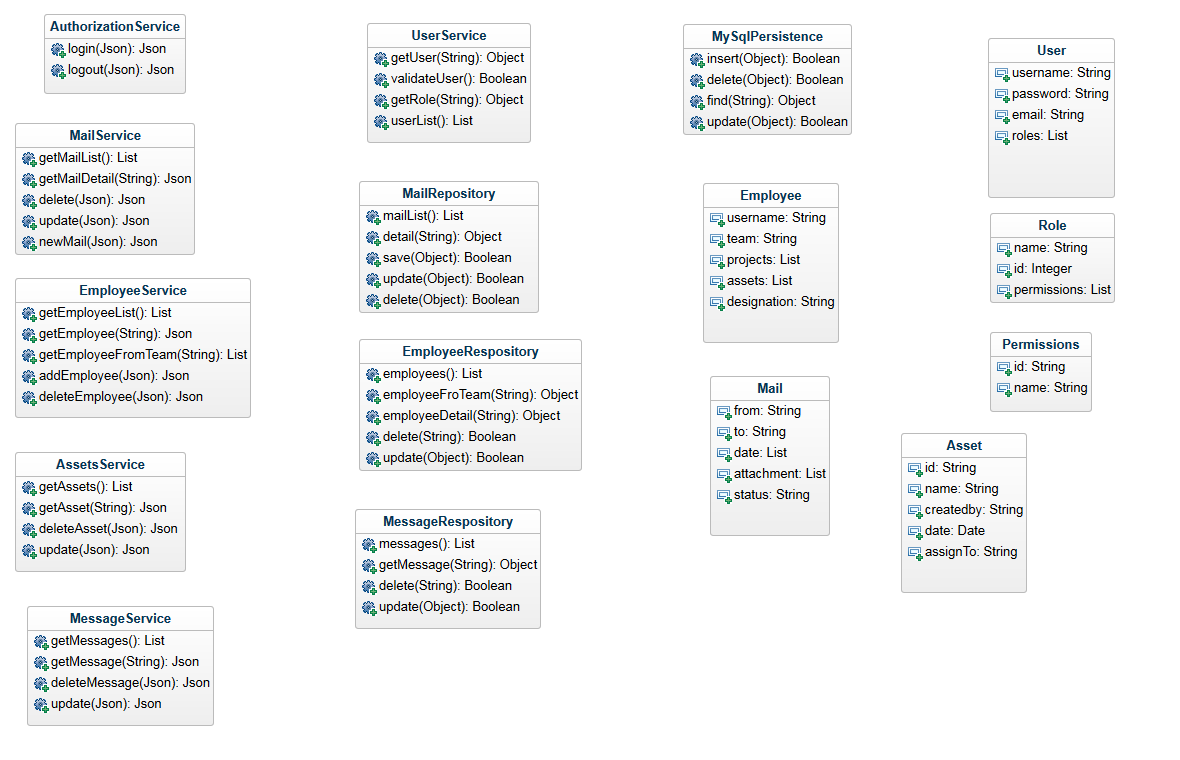
**Application Server containing REST Services (Apache Tomcat)**

**Web Server (Single Page Application hosted here)**

2.2 E-R Diagram



2.3 Class Diagram

****

2.4 Data Flow Diagram

Validate Username and password from DB

Send username and password to Services

**User Login**

Login user based on token generated.

Send token back to client browser/ web server.

Check user details and create token

Request for Login.

**Database**

**Application Server**

**Web Server**

User

**Common Data flow:**

Request for data

Convert and display JSON data as html.

Required data for user in JSON format

Query data from Database

Validate login token from request and fetch required data from database

Return data to application server.

Send user request with login token to REST services

**Database**

**Application Server**

**Web Server**

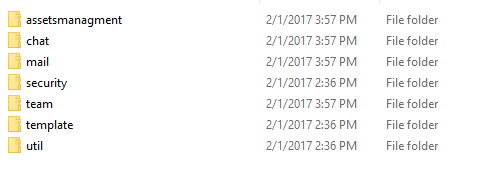
User

System Design

3.1 Modularization

**3.1.1 Dojo Components**

Dojo Toolkit provides us flexibility to divide application in to small modules and widgets. Functionalities are divided into packages in dojo application. Every functionality in dojo application will be in a separate package/ folder. This folder will contain module code and html template file.

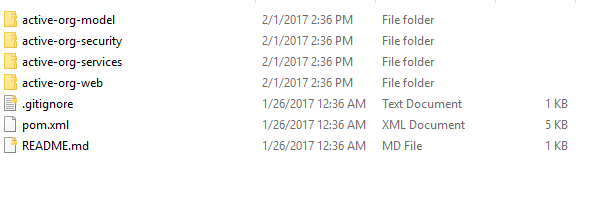


Every folder contains module js files and html templates. And will be loaded when needed from mail configuration file.

This will help to keep the application modular and will be easy to maintain and extend application functionality. Modules will be kept separate of each other and if need to add more modules then will be easy to add.

3.1.2 REST Components

REST Services are built using Java and Maven. Maven is a build tool for java based applications. This helps to maintain dependencies and project structure and also to divide project into multiple modules.



Pom.xml is a configuration file for maven build tool. This contains all the information about the modules we are using.

All the directories are separate modules and can use each other if needed.

3.2 Data Integrity & Design

3.2.1 Data Security

Security is handled in both REST and frontend application. The frontend application maintains a token, which is generated by services. And based on service validation it maintains user state. If this token is not valid then user will be asked for valid username and password and will not be able to access anything from front end. After getting valid username and password from user, it is passed to REST services to generate a token and validate user, if success then user will be granted with appropriate permissions based of the roles available in database.

Token Base Authentication

Information stored on websites varies widely in the amount of information which is available either publicly or privately. On some sites, a full-fledged database of personal information may be available -- from addresses and phone numbers to email and cha t contacts. On other sites, users may list only an email address and a website URL. More than that, sites may offer varying privacy levels for various sets of information. On Orkut, for example, users can set the level users must be in order to see each piece of information. This presents many issues when users of these sites want to start using FOAF for outside sources -- distributed data for profile information, or other similar means. Typical web-authentication is cookie-based, and offering a similar authentication to spiders or crawlers requires providing login information that most people would be reluctant to provide. (I certainly wouldn't want to give my password out to every website on the internet that asks for it.) There are solutions to this pa rticular problem, and in general, they aren't all that difficult to use. By adding authentication in a way that protects both the privacy of the user and their login information, you can increase the use of FOAF in remote applications.

No application wants to give out permanent access to user information. That type of access is more than most remote sites will ever need, in terms of both length of time and of the level of access. However, most sites are not designed to offer temporary viewing of resources. Thus, when users want to share resources - such as a FOAF file - between sites, they are forced to either save the information locally or deal without the information which the tool in question cannot see. When taking advantage of a tool's security levels, hosting a file locally completely defeats the purpose of the security levels in the first place. The solution to these problems is to offer temporary authentication via a non-password scheme. There are a number of possibilities as to how to perform this authentication technically. However, in order to perform distributed authentication while protecting both users privacy and passwords, the only way to authorize such authentication is via a token based authentication system.

The general concept behind a token-based authentication system is simple. Allow users to enter their username and password in order to obtain a token which allows them to fetch a specific resource - without using their username and password. Once their to ken has been obtained, the user can offer the token - which offers access to a specific resource for a time period - to the remote site. Using some form of authentication: a header, GET or POST request, or a cookie of some kind, the site can then determine what level of access the request in question should be afforded.

The type of changes this type of authentication requires is obviously dependent on the current implementation of your site. Example code I might be able to write in Perl or PHP would not only be language and implementation specific, it would also be application specific. However, some general principles should be considered in both the creation of a process to obtain tokens and the process of using them. Simplicity for users, robustness for interoperability, and protection of user data are all important f or your application, and each can fall by the wayside in attempting to design a system which fits user expectations.

A typical process for token based authentication would work as a communication between two sites: a remote site R and a base site, where the data is already stored, B. A user wishes to provide temporary authorization to site R to read his FOAF file at sit e B.

User selects "Import FOAF" or similar option

Site opens "Import FOAF" page, allowing user to select from a drop down selection of sites from which to import

User selects "Site B" from the list, and as a result, a box appears which allows the user to enter an authentication token. The site may optionally provide a link to the token creation page at this step to ease the task for the user.

User visits Site B Token Generation page. Once there, the user is asked for a username and password, as well as a resource to which site R should have access.

User receives authentication token from Site B, and copies token to form box on Site R.

Site R fetches the requested resource, using the authentication token in question. The token, rather than a typical cookie based authentication session, identifies the access level at which the site can obtain information.

There are a number of implementations which can be used to determine the access level, depending on the type of information which should be made available and what is currently available. The system I have the most knowledge in is LiveJournal, and as such, I will use that as an example of how to add token-based authentication to a resource.

LiveJournal has a number of security levels throughout the site. You can protect your entries to be readable to only you, only your friends, or up to 30 subgroups of your friends using unioned filters. However, this same level of protection and choice is not available to you when you are storing your profile information. Unlike Orkut, LiveJournal makes minimal options available for selecting the security of your profile information. Fortunately, for this example, this adds a level of simplicity not available in most cases.

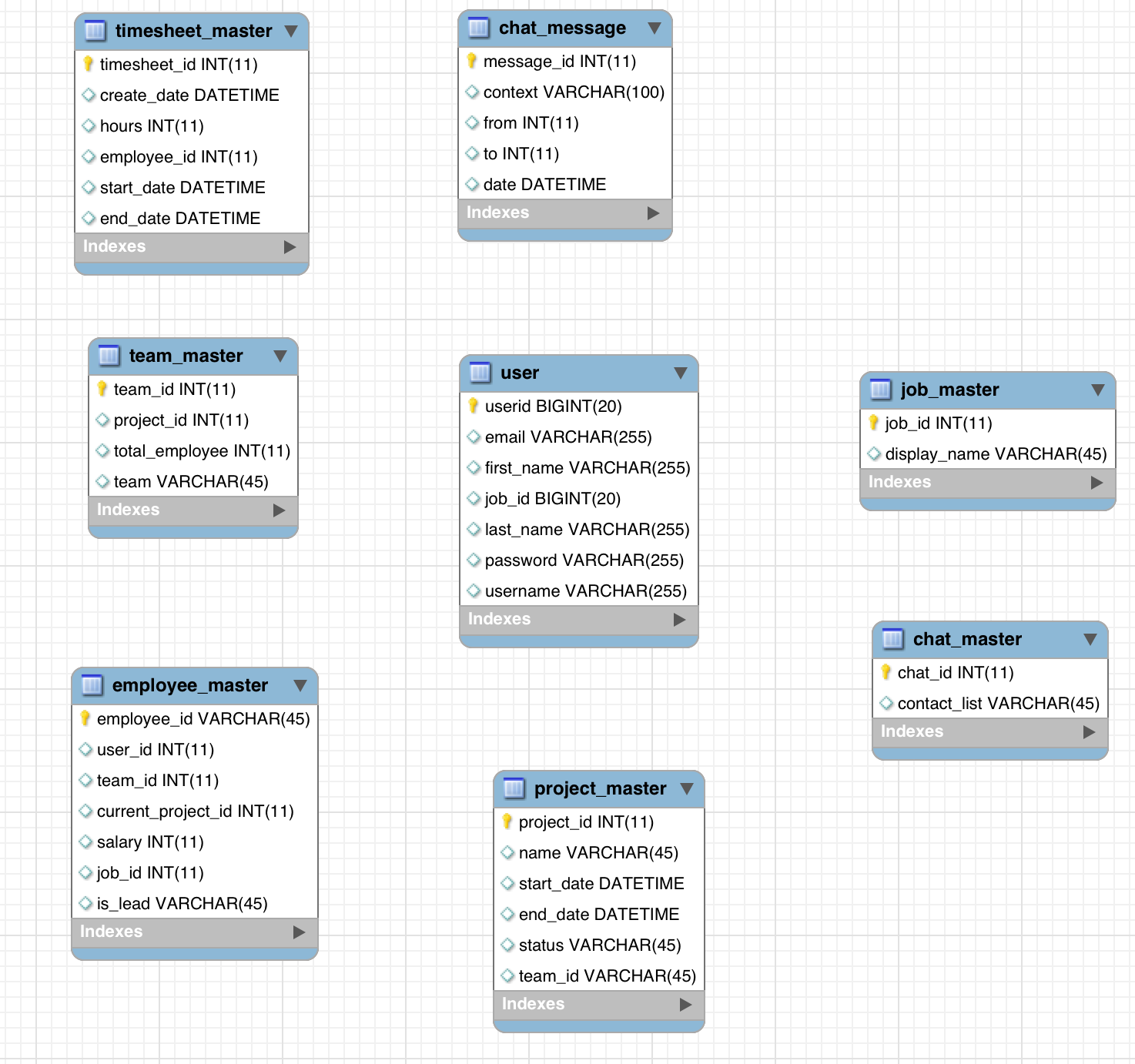
When a request is received, LiveJournal passes the request to the FOAF creation instance, which then fetches the information unconditionally. Once the information has been retrieved, LiveJournal parses through the data manually and builds up a FOAF fil e based on the security options that are set. In order to add token based authentication to the system, we must add a check into the FOAF creation routine that allows us to override the "public" security available to most agents with a valid authentication key. In LiveJournal's case, to provide access to contact information at owner level, we need add only about 15 lines of code and a small MySQL table to store valid authentication keys. (A way to allow users to request such keys is not included in this code.) The attached diff file contains an example of how this works. As most of you will be able to understand, we are checking the offered authentication token against the tokens stored in the database within the past 24 hours. Although this is a relatively minimal example - LiveJournal has few security levels for profile data, and this change did not interfere in any way with existing APIs or functionality - it is obvious that in many cases, creating an authentication token backend is as simple as an extra if clause.

The frontend for using such an authentication system must be simple if users are to understand and use it. By providing the standard login template with a slightly modified interface to provide more information and documentation about authentication tokens, as well as the option to choose a resource specific resource, users can feel that they are in a similar environment - entering login data for the purpose of obtaining more specific information about themselves on the site in question. Once the information is verified, the user can then be offered a token which they can copy paste into another application, or perhaps the ability to have the token sent to a specific email address. Keep in mind, however, that these tokens offer, for a limited amount of time, a level of authentication, and in most cases, you will want to afford the same level of protection over these tokens: whether it be SSL encryption, or some other method - as you would over login information.

3.2.2 Design

Data is design as relational database. The design contains multiple tables and have relation between each other.

ER Diagram with relations and design:



3.3 Procedural Design

Start

­­

Response with data requested as json

403 Response that will redirect to login

Valid Token

Invalid Token

Validate User

Response

Request

Database

Response in 403 Error

Get Data from DB

Validate User token

Request for Data

3.4 User Interface Diagram

dasdas

**Chat box**

**Notifications (List/detail)**

**Projects (List/ Detail)**

**Mail (List/ Detail)**

**Assets (List/ Detail)**

**Team (List/ Detail)**

**Dashboard**

**Login Form**

IMPLEMENTATION

4.1 Dojo Implementation

The dojoConfig object (known as djConfig prior to Dojo 1.6) is the primary mechanism for configuring Dojo in a web page or application. It is referenced by the module loader, as well as Dojo components with global options. It can further be used as a configuration point for custom applications, if desired.

The old object name of djConfig is deprecated, but any existing code using it will continue to work up until 2.0. At the time of writing, most documentation still uses djConfig; the two names are directly equivalent, but we'll adopt and encourage use of the new dojoConfig name from here on.

Code: /static/js/config.js

dojoConfig = {

has: {

"dojo-firebug": true,

"dojo-debug-messages": true

},

parseOnLoad: false,

baseUrl: "static/js",

packages: [

{ name: "dojo", location: "libs/dojo" },

{ name: "dijit", location: "libs/dijit" },

{ name: "dojox", location: "libs/dojox" },

{ name: "app", location: "app"}

]

};

Main file that will initialize configuration and all the components needed.

**Code: /static/js/init.js**

dojo.require("dojo.dom");

dojo.require("dojo.parser");

dojo.require("dijit.registry");

dojo.require("dojo.domReady!");

dojo.require("dijit.dijit");

dojo.require("dojo.html");

dojo.require("app.util.htmlUtil");

dojo.require("app.util.common");

dojo.require("app.security.login");

dojo.addOnLoad(function(){

//Parsing htmls with dojo components and widgets.

dojo.parser.parse();

//Bind click action to show login form

dojo.connect(app.util.htmlUtil.getById(app.util.common.CONS.userInfoNav), "onclick", function(e) {

app.util.htmlUtil.show( app.util.common.CONS.loginWrapper );

});

//Click function for close button for login form.

dojo.connect(app.util.htmlUtil.getById(app.util.common.CONS.closeLogin), "onclick", function(e) {

app.util.htmlUtil.hide( app.util.common.CONS.loginWrapper );

});

//Click function for login form submit

dojo.connect(app.util.htmlUtil.getById(app.util.common.login.submit), "onclick", function(e) {

app.security.login.requestLogin();

});

});

Security implementation for login handling:

**Code: /static/js/app/security/login.js**

define("app/security/login", [

"dojo/\_base/declare",

"app/util/common",

"app/util/rest",

"app/util/htmlUtil",

"dojo/cookie"

], function(declare, common, rest, htmlUtil, cookie) {

var login = {

requestLogin : function() {

var response = rest.login( htmlUtil.getValue(common.login.usernameId) , htmlUtil.getValue(common.login.passwordId));

this.loginCallBack(response);

},

loginCallBack: function(response) {

if (response.status) {

//creating cookie to add login token cane in service response.

cookie("loginid", response.token, { expires: 10 });

htmlUtil.hide( common.CONS.loginWrapper );

} else {

//Showing error in case of invalid login.

console.log("error id: "+ common.CONS.loginErrorId)

htmlUtil.setHtml( common.CONS.loginErrorId, common.error.invalidLogin );

}

}

};

return login;

});

Main Html page to initiate page source:

**Code: /index.html**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<title>Home</title>**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<link href="https://fonts.googleapis.com/css?family=Raleway" rel="stylesheet">**

**<link rel="stylesheet" href="./static/css/claro/claro.css">**

**<link rel="stylesheet" href="./static/css/bootstrap.css">**

**<link rel="stylesheet" href="./static/css/font-awesome.min.css">**

**<link rel="stylesheet" href="./static/css/custom.css">**

**<script src="./static/js/config.js"></script>**

**<script src="./static/js/libs/dojo/dojo.js"></script>**

**<script src="./static/js/init.js"></script>**

**</head>**

**<body class="claro">**

**<div id="container">**

**<div class="row">**

**<section id="header">**

**<div class="col-xs-12 col-md-8" id="nav">**

**<nav id="mainNav">**

**<ul>**

**<li>**

**<a href="#" id="init-mail"><i class="fa fa-envelope"></i></a>**

**</li>**

**<li>**

**<a href="#" id="init-team"><i class="fa fa-users"></i></a>**

**</li>**

**<li>**

**<a href="#" id="init-assets"><i class="fa fa-th-large"></i></a>**

**</li>**

**<li>**

**<a href="#" id="init-projects"><i class="fa fa-sliders" aria-hidden="true"></i></a>**

**</li>**

**</ul>**

**</nav>**

**</div>**

**<div class="col-xs-6 col-md-4">**

**<nav id="userinfo">**

**<ul>**

**<li>**

**<a href="#" id="nav-notification"><i class="fa fa-bell" aria-hidden="true"></i></a>**

**</li>**

**<li>**

**<a href="#" id="nav-user" style="position: relative;"><i class="fa fa-user-circle"></i></a>**

**<div id="user-detail-nav" class="hide">**

**<p><a href="#" id="show-login">Login</a></p>**

**<p><a href="#" id="show-logout">Logout</a></p>**

**<p><a href="#" id="show-profile">Profile</a></p>**

**</div>**

**</li>**

**</ul>**

**</nav>**

**</div>**

**<div style="clear:both"></div>**

**</header>**

**<section id="breadcrums">**

**<p>Home</p>**

**</section>**

**</div>**

**<div class="row">**

**<div class="col-xs-12 col-md-12" id="main-content-section">**

**<div class="mail-module" data-dojo-type="app/mail/mailinit">**

**Loading...**

**</div>**

**<div class="assets-module" data-dojo-type="app/assetsmanagment/assetinit">**

**Loading...**

**</div>**

**<div class="team-module" data-dojo-type="app/team/teaminit">**

**Loading...**

**</div>**

**<div class="project-module" data-dojo-type="app/team/projectsinit">**

**Loading...**

**</div>**

**</div>**

**</div>**

**</div>**

**<div id="login-wrapper" class="hide">**

**<div class="row">**

**<div class="container">**

**<div id="login-form" class="col-md-4 col-md-offset-4">**

**<a href="#" id="close-login"><i class="fa fa-times" aria-hidden="true"></i></a>**

**<div class="form-group">**

**<label for="username">Username:</label>**

**<input type="text" class="form-control" name="username" id="username">**

**</div>**

**<div class="form-group">**

**<label for="password">Password:</label>**

**<input type="password" class="form-control" name="password" id="password">**

**</div>**

**<p class="error-message" id="login-error"></p>**

**<button type="button" class="btn btn-success" id="loginSubmit">Login</button>**

**</div>**

**</div>**

**</div>**

**</div>**

**</body>**

**</html>**

Basic html page with dojo is created. Now we have to create out modules and initialize them on this page:

1. Mail Module:

The mailinit.js file under app/mail folder is responsible to initialize this module. And this folder will also include html files that will contain templating for this module.

Templates will be under app/mail/html

**Code: app/mail/html/container.html**

<div data-dojo-type="dijit/layout/BorderContainer" id="layoutContainer">

<div data-dojo-type="dijit/layout/ContentPane" data-dojo-props="region:'leading'" style="width: 200px;" class="col-md-4 side-tree">

<div id="mail-folders">

list of folders here

</div>

</div>

<div data-dojo-type="dijit/layout/ContentPane" data-dojo-props="region:'center'" class="col-md-6">

<div id="mail-list">

List here

</div>

<div data-dojo-type="dijit/layout/ContentPane" data-dojo-props="region:'bottom'">

<div id="mail-list">

detail here..

</div>

</div>

</div>

</div>

This will create two sections in this module. List section and a content section. List section will include folder listing for mails. And content section will include mail list and a section for detail view.