**Enterprise Distributed Systems**

**Class Project – Workforce Management System**

**(Spring 2015)**

**Project Report**



**Team 9**

|  |  |  |
| --- | --- | --- |
| **Team Details** | | |
| **Team Member Name** | **Team Member e-mail** | **SJSU ID** |
| Apoorva Reddy Gouni | apoorva.gouni@sjsu.edu | 009540349 |
| Mani Dheep Vipperla | manidheep.vipperla@sjsu.edu | 010093447 |
| Satish Kandimalla | satish.kandimalla@sjsu.edu | 010011612 |
| Sairam Nutheti | sairam.nutheti@sjsu.edu | 010100545 |
| Sravani Gunti | sravani.gunti@sjsu.edu | 010090405 |
| Surabhi Bommenahally Anand | surabhi.bommenahallyananda@sjsu.edu | 010114520 |

# Abstract

The system designed is a 3-tier application that implements the functions of a simple Workforce Management System for security services such as bay alarm and ADT. The different types of objects and managed and implemented in this workforce management system are: Guards, Clients, Buildings, Schedules, Reports and alerts. For each type of object we have implemented an associated database schema that is responsible for representing how a given object should be stored in a relational database. Our system has been built on a distributed architecture which would be explained in detail later. Our system consists of three tiers: client tier, it’s a node application that allows the user to interact with the system; middle ware, we have implemented it using REST based Web Services and used Rabbit MQ for messaging and this is the place where most of our processing is done and; the third tier is the database to store the state of our entity objects, we have used MySQL for storing the data.

# Functionalities

The system designed can handle three types of users: Admin, Guard and Client. User needs to login to the system by entering the username, password and role. Based on the type of the role mentioned the page is navigated to their respective pages. Below are the functions mentioned based on the role of the user. And each user will be assigned a unique id.

## Based On Admin:

As an admin, user can access both client and guard related activities. The admin will have the following features:

* Create, delete and edit client information.
* Create, delete and edit guard information.
* Create, delete and edit building information.
* Create, delete and edit checkpoints information.
* Schedule patrol timings for guards.

## Based on Guard:

As a guard, user can do the following activities:

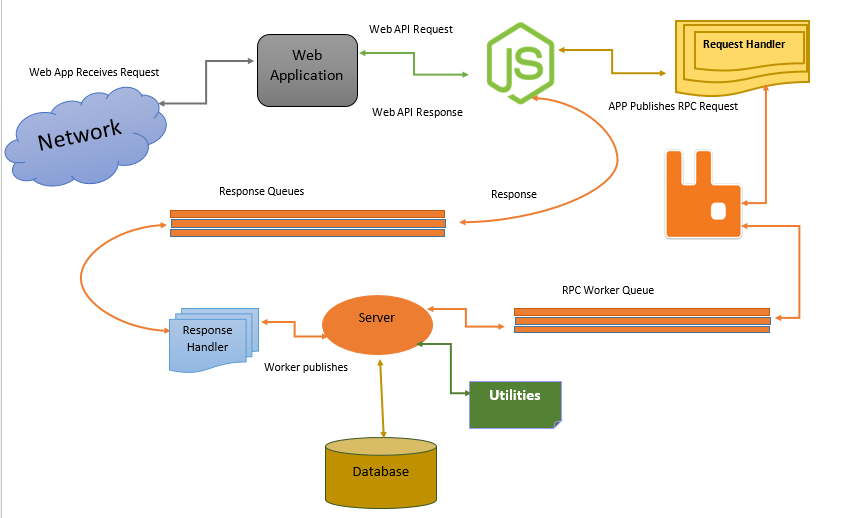
* Retrieve checkpoints.
* Retrieve patrol timings.
* Log Report.
* Retrieve Alert types
* Submit report and alert types.
* Display guard details.
* View Schedules.

## Based On Client:

* View Bills.
* View Alerts.
* View Reports.
* Edit client information.

# Project Architecture

* The below diagram illustrates the solution architecture for the Workforce Management System. It highlights the different components and tools that were used for implementing the application.



## Components

### Network:

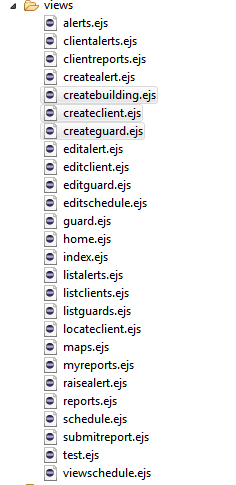
A telecommunication network which allows computers to exchange data is used in the management system application. Data is passed to each other along data connections in the networked computing devices using the computer networks. Packets are transferred as data. Internet is the computer network used, which is known as the best form of network.

### Application (WFMS):

Workforce Management uses web browser as a client. It is created in a browser- supported programming language such as express js, Angular js, which is a JavaScript framework and Bootstrap for styles and relies on a web browser to render the application. As angular js is an open sourced web application framework for developing a single – page applications.

Using the angular js bootstrapper the tasks are performed in three phases after the Document Object Model is loaded: a new injector is created; directives which decorate the DOM are created; directives to specific scope are linked.

The following view pages are part of UI engine:



### Node Js:

For server side programming and networking we have used Node js. The application is written in JavaScript language and run in Node js environment on Microsoft Windows. It provides an event driven architecture and a non-blocking I/O API that optimizes an application’s scalability and throughput. We did not use Apache HTTP software as Node js contains built-in library to allow applications to act as Webserver.

### RabbitMQ server middleware:

This middleware provides a queue based routing mechanism to route the user requests from the UI engine to the back end server. There are multiple ways to use the RabbitMQ server; I have chosen Remote Procedure calls (RPC) technique to establish a communication between the UI engine and the back end server. The UI engine forms the client interface and the back end server forms the server interface in this RPC technique. Creating the RabbitMQ middleware involves the following steps:

**Installing rabbitmq-server:** followed the steps mentioned here [http://www.rabbitmq.com/install-windows.html to](http://www.rabbitmq.com/install-windows.html) install and configure rabbitmq- server on my local machine. On completion of the installation the rabbitmq-server will be running on the host machine with IP address 127.0.0.1 by default. This service manages the static or dynamic queues created by the client interface or server interface.

#### Client interface:

The UI engine defines the client interface for the rabbitmq based RPC.

* UI engine: is the client interface that defines the user request function; (rpc/amqprpc.js and rpc/RequestHandler.js)
* It creates a connection to the RabbitMQ service running on HOST 127.0.0.1.
* Creates a unique correlation ID using the crypto service to be associated with every user request for security purposes.
* Create a timeout service to avoid starving on a server response.
* Set up response queue for the back server to publish its response. The response queue is a dynamically created temporary queue for every user request.
* The correlation ID, user request content and the response queue name are published to the server interface based statically create queue.

#### Server interface:

The back end server defines the server service for the rabbitmq based RPC.

#### Back end server:

is the server interface that defines the user request handling and response; (app.js, GuardService.js/guardController.js, AdminService.js/adminController.js and ClientService.js/clientController.js)

* It creates a connection to the RabbitMQ service running on HOST 127.0.0.1.
* On successful connection, creates three static queues:

“Admin Service” for services done by admin; below are the services performed by admin:

* Sign in,
* Create client,
* List client details,
* List all the clients,
* Retrieve each client detail,
* Buildings corresponding to the client,
* List building details,
* Delete client,
* Edit client information,
* Change the password for user,
* Delete building,
* Create building,
* Edit building,
* List the report of all buildings,
* Get all report details,
* List the report of all clients,
* List the report of buildings,
* List the report for a client based on start and end date,
* Get detailed report of building based on building Id,
* Create guard,
* List the details of guard,
* Get the details of guard,
* Delete a guard,
* Display guard details,
* Edit guard details,
* Create schedule for guard,
* Delete schedule for guard.
* Get all schedule details.
* Edit schedule details
* Create alert types
* Delete alert types
* Edit alert types
* Get buildings based on guard
* Get alerts.

“Client Service” for services done by client; below are the details:

* Search based on alert criteria
* Retrieve alerts based on client Id, start date and end date.
* Retrieve reports based on client details, start date and end date.
* Retrieve bills based on client details, start date and end date.

“Guard Service” for services done by admin, below are the details:

* Retrieve buildings served by guard.
* Retrieve checkpoints.
* Retrieve patrol timings of guard.
* Log reports by guard.
* Retrieve alerts.
* Submit alerts.
* Submit reports.
* Display reports by guard.
* View Schedules of guards.
* Retrieve all guard reports.

* Subscribe for incoming data in these static queues.
* Process the user request and publish the result to the dynamically created temporary response queue from the client interface using the “replyTo” option.

**Back end Server:**

Provides the backend controlling part that binds the database with the UI engine. The back end server has two ends with one end handling the user requests delegated from the RabbitMQ queues and publish the database responses back the UI engine; the other end handles the database connections, adding/retrieving/updating data. The following modules define the back end server functionality:

**RabbitMQ controller –** app.js, adminService.js/adminController.js, clientService.js/clientController.js and Services/guardController.js: This controller;

* Creates connection with RabbitMQ server using the node.js based “amqp” module.
* Creates three static queues.
* Subscribe for data passed to these queues by the UI engine; handle the request by passing to the database controller and publish the result to the result queue subscribed by the UI engine.

**Database controller** – services/connections.js: This handles all the database related calls and creates responses to be passed to the UI via the RabbitMQ server. Every function handles the corresponding sql queries and displays or returns the results to the UI engine accordingly.

**Connection pooling controller -** services/dbConnectionsController.js and services/enableConnectionPooling.js: This handles the connection pooling of database connections. The connection pooling can be enabled and disabled. If it is enabled; on application startup a connection pool of 100 database connections are maintained in a queue (dynamic). For every database query a connection from this queue is retrieved and restored to the queue on completing.

**Note:** If the connection pool is short of database connections another queue is used to maintain a list of waiting database requests. This is developed using the wait queue mechanism. If there is not connection available the user request is pushed to this queue and retrieved on a database connection being available.

### Exception controller – services/errorController.js: This handles all database related, connection pool and wait queue related errors cases and the corresponding exceptions are thrown to avoid system crashes.

* **DB Schema/strategy**
* **Screen Captures of Client application**

# Performance tuning techniques

To improve performance, scalability and robustness of the system, we have introduced few techniques to our system. Using these techniques, the application can be scaled to work with multiple services and users.

Following things have been implemented to project to improve performance:

**Data Caching:**

**Redis:**

Redis is a fast and efficient in-memory key-value store that uses the system memory. It is very fast. The lookup time is very fast in REDIS. It is also known as a data structure server, as the keys can contain strings, lists, sets, hashes and other data structures. We can use the node\_redis module to interact with Redis. Command to install is: npm install redis.

Redis is a server. To connect to it we need to create a client.

Once the connection is established we will read and write the data into the redis store using set and get commands.

Data Caching Mechanism- The application logic is as the following 3 layers: Application Code, Redis Cache, and MySQL Data Store. MySQL acts as a permanent persistent data store while Redis acts as a proxy between Application Code and MySQL store. For every database request, we will first check in the Redis to see if there is any data already cached. If that particular key exists, it means the data is cached. So we will fetch the required data from redis and process the request. If there is no data cached in REDIS, then we will hit the MySQL DB, fetch data & serve the request and then store it in Redis for future access.

**How we implemented:**

1. Import the Redis module.

var redis = require ('redis');

2. Create a client to connect to redis.

var client = redis.createClient (); //creates a new client

3. Once the connection has been established. Basically, you just need to listen for connect events using client. On command. Connect is an event that is triggered when a new connection is created.

4. Write the data into the Redis store using this command client. Set or client. Push command. This command will store a value against the key. We also pass an optional callback to get a notification when the operation is complete. If the operation failed for some reason, the err argument to the callback represents the error.

5. To retrieve the value of the key use the command client.get or client.lget command. The value of the key can be accessed via the callback argument reply. If the key doesn’t exist, the value of reply will be empty.

**SQL Caching:**

In this tuning technique we store the results from database. When the database has to be accessed again with the same query, instead of approaching the database the results are taken from the cache.

**How we implemented:**

We have implemented SQL caching with the help of Hash Maps. The key to this hash map is the name of the API and the value to the key is the rows obtained from the query. So before hitting the database, we will check the cache whether it already has the data from the past operations. If yes, the data from the cache is used, else we approach the database with the query written based on the requirement.

**Benefits:**

Using SQL caching, the necessity of hitting the database is greatly reduced. Hence, the number of services requesting the connections from the pool will be reduced there by increasing the system performance.

**Connection Pooling:**

**What is Connection pooling?**

The process of creating a connection, always an expensive, time-consuming operation, is multiplied in these environments where a large number of users are accessing the database in short, unconnected operations. Creating connections over and over in these environments is simply too expensive. The application allocates a pool of connections to the database. Application programmers don’t create applications, but borrows connections from the pool. So, the connections are recycled.

**How Connection Pooling is implemented:**

Here, we will maintain two queues, one for connections and a waiting queue. All the connections are stored in queue, once there is a request for the db connection, a connection is popped of the queue and provided. The service will return the connection immediately after its interaction with db ends. When there isn’t any connection available, the service request is put into the waiting queue.

**Benefits:**

The main benefits to connection pooling are:

* Reduced connection creation time.

Although this is not usually an issue with the quick connection setup that MySQL offers compared to other databases, creating new connections still incurs networking and driver overhead that will be avoided if connections are recycled.

* Simplified programming model.

When using connection pooling, each individual thread can act as though it has created its own connection.

* Controlled resource usage.

If you create a new connection every time a thread needs one rather than using connection pooling, your application's resource usage can be wasteful, and it could lead to unpredictable behaviors for your application when it is under a heavy load.

**RabbitMQ:**

This middleware provides a queue based routing mechanism to route the user requests from the UI engine to the back end server. There are multiple ways to use the RabbitMQ server. We have chosen Remote Procedure calls (RPC) technique to establish a communication between the UI engine and the back end server. The UI engine forms the client interface and the back end server forms the server interface in this RPC technique. Creating the RabbitMQ middleware installation involves the following steps:

1. Installing rabbitmq-server: followed the steps mentioned here http://www.rabbitmq.com/install-windows.html to install and configure rabbitmq-server on my local machine. On completion of the installation the rabbitmq-server will be running on the host machine with IP address 127.0.0.1 by default. This service manages the static or dynamic queues created by the client interface or server interface.
2. Client interface: The UI engine defines the client interface for the rabbitmq based RPC.

UI engine: is the client interface that defines the user request function ;(rpc/amqprpc.js and rpc/RequestHandler.js)

1) It creates a connection to the RabbitMQ service running on HOST 127.0.0.1.

2) Creates a unique correlation ID using the crypto service to be associated with every user request for security purposes.

3) Create a timeout service to avoid starving on a server response.

4) Set up response queue for the back server to publish its response. The response queue is a dynamically created temporary queue for every user request.

5) The correlation ID, user request content and the response queue name are published to the server interface based statically create queue.

6) The response queue name is published with the “replyTo” option.

**Server interface:** The back end server defines the server service for the rabbitmq based RPC.

**Back end server:** is the server interface that defines the user request handling and response.

It creates a connection to the RabbitMQ service running on HOST 127.0.0.1.

On successful connection, creates 3 static queues: client, admin, guard.

Subscribe for incoming data in these static queues.

Process the user request and publish the result to the dynamically created temporary response queue from the client interface using the “replyTo” option.

Back end Server: Provides the backend controlling part that binds the database with the UI engine. The back end server has two ends with one end handling the user requests delegated from the RabbitMQ queues and publish the database responses back the UI engine; the other end handles the database connections, adding/retrieving/updating data.

**SQL Escape Statements:** SQL Injection is one of the more popular application layer hacking techniques that is used in the wild today. It is a trick that exploits poorly filtered or not correctly escaped SQL queries into parsing variable data from user input. The idea behind SQL injection is to convince the SQL application to run an SQL string that was not premeditated. In order to avoid SQL Injection attacks, you should always escape any user provided data before using it inside a SQL query. We have done it using the mysql.escape (), connection.escape ().

**Individual Member Contribution**

|  |  |
| --- | --- |
| **Name** | **Contribution** |
| Apoorva Gouni | * Involved in designing and creating guard GUI of homepage * Involved in implementation of Guard functionalities. * Implemented Connection pooling. * Involved in testing the application. * Involved in preparing project report. |
| Mani Dheep Vipperla | * Involved in implementation of code to list the reports, search the reports and to retrieve report details. * Involved in unit testing * Involved in preparing the project report. |
| Satish Kandimalla | * Involved in implementation of code to create, delete, edit or list client, building and checkpoints. * Implemented Redis SQL Caching. * Involved in unit testing * Involved in preparing the project report. |
| Sairam Nutheti | * Involved in database designing. * Implemented the functionality of to do list, dashboard maps, and scheduling on the homepage of work force management system. * Involved in UI Integration. * Involved in testing the application. * Involved in preparing project report. |
| Sravani Gunti | * Involved in designing GUI and also implemented code for client side functionalities like, viewing the bills, reports and alerts. * Implemented client and server side validations. * Involved in unit testing * Involved in preparing project report. |
| Surabhi Bommenahally Anand | * Involved in implementation of code to create, delete and edit guards. And also in searching, listing the guard values based on given attributes. * Involved in unit testing * Involved in preparing the project report. |

* **Observations and lessons learnt**