Folder Overview

**publishconsole.html (Backup: index.html) <- Frontend | Backend -> pubsub.js**

*publishconsole.html* serves as the console for Publishing of Data from different agencies to help to simulate the flow of data and how it works.

*pubsub.js* allows different agencies to connect to Solace network to publish and subscribe from the console. *publishconsole*.html is also an interface for the console to test the publishing and subscribing of the message within its cloud connection to Solace.

In pubsub, each PubSub declared is equals to one agency. Each agency has some given function. Main functions are connect, publish/send messages based on data.gov.sg, stop sending (stop time interval for publishing), subscribe, unsubscribe, consuming, stop consuming and disconnect from Solace Cloud.

**AccessListCall.js**

This file is a function to handle Access Control for every agency that is connected to the central broker. Agencies can only publish or subscribe based on verified permission from central broker. It is referenced under *Admin.html* and *login.js*.

For example, if a user wants to publish a message onto the central broker, each of the individual broker must add the specific topic on its subscription list. This function will attempt to call all client username in central broker, followed by adding the topic in its own publish list, and lastly adding under each of other subscription list in ACL. The published topic will also be added into SOLACE\_ALL queue as subscription for central analytics to consume and process. This function is done recursively.

**config.js**

Contains sensitive information as simulated NEA and LTA to connect to Solace Cloud and handle SEMP Protocol. Also contains Microsoft Azure API Keys to process the traffic images on central analytics. The file should be hidden from public.

**BrokerRegistration.js**

Attempts to register any new broker on central broker with provided details. Handles creation of new ACL and client usernames, as well as the bridge between central and new broker. It is referenced under *Admin.html*.

Firstly, the function needs to get the necessary data in order to establish a new broker on the bridge. The example is given in the code documentation. Secondly, *BrokerRegistration* attempts to create a new aclProfiles and clientUsernames for central broker. *BrokerBridgingConnection* will then create a bridge between central broker and the new broker to publish and receive messages. Thirdly, *SempAPIRegistration* is called by *BrokerRegistration* and *BrokerBridgingConnection* to execute its action and command on Solace Cloud. The administrative access for SempAPI is based on config.js.

**dataflow.html <- Frontend | Backend -> Centralpubsub.js**

*dataflow* attempt to provide the real-time data flow chart for each agency on the frontend that logins to the web page. On the other hand, the backend serves as the Central Analytics that processes data published from the console and republishes new topic with the processed data. When the chart starts “consuming messages”, the central analytics will start its processing from the queue.

In the javascript section of *dataflow.html*, there is a few functions to perform its role.

*SempAPI*: called to gain administrative access to Solace Cloud account to execute command.

*updateDataFlow*: For every message that is received from queue consumption in *pubsubmap.js*, this function is executed to update the data for graph display. It is collecting the amount of message flow for each respective topic per second.

*updateChart*: updates the chart every second to push the data on the chart and display the message flow.

**Centralpubsub.js <- Main Backend | Sub Backend -> EventMsg.js**

As the backend processes data, it triggers two events: updating the graph and processing the data into analytics. *EventMsg* receives the message and attempts to republish the processed message.

1. Updating the Graph: When the client starts to consume messages, *updateChart* function will be initiated and executes every second. As each of the messages are being consumed, the message flow will be updated via *updateDataFlow*.
2. Processing the data as central analytics: *TempEventCall*, *RainEventCall* and *ImageEventCall* are referenced from *EventMsg.js*. Each message will be processed respectively based on its topic. More details are written in code documentation.

**govtech.js & tempData.js**

Gets the API data from data.gov.sg when it’s being called. Used by the console to publish simulated data. (Reference to *pubsub.js*) Both js file does the same thing but applies for different dataset. *tempData* is meant for temperature dataset while *govtech.js* is more general.

**microsoftanalyse.js**

Calls the analytics result from Microsoft Cognitive API with the published traffic Images. Used by the central broker to process traffic images. (Referenced by *EventMsg.js*, API Key are referenced in *config.js*)

Depending on the image, Microsoft computer vision API will return a list of objects as well as its probability of its accuracy. (etc. [ { "name": "person", "confidence": 0.997822046 }, { "name": "clothing", "confidence": 0.9692062 }, { "name": "indoor", "confidence": 0.962251842 } ] )

**map.html <- Frontend | Backend -> pubsubmap.js, selectorFiltering.js**

Google Map demonstration of the published data through Solace network. Data is decoupled and works independently from one another. *Pubsubmap.js* in charge of receiving subscribed data while selectorFiltering.js attempts to display received data on map.

In *map.html*, Google Maps is being initialized, as well as settings for frontend filtering, brokers’ initialization and Temperature Chart. For clarification, these are some of the breakdowns within its script.

*CustomMarker*: a library that is meant to create custom marker for an event message that is received on the google map. This is to illustrate the happening of events better on design aspect.

*google.maps.rectangle*: this is meant to show the subscription area for each of the markers that are displayed on the map. Regardless if the marker is a hospital or a remote traffic patrol, this rectangle will show the boundary of its area when an event message is being triggered or received within the boundary. Its purpose is to show each marker can subscribe messages based on geographical location.

*Topic Subscriber*: there is a few subscribers that serve different purposes.

1. *subscriber*: general subscription of messages from different topics to display on the map
2. *subscriberImage*: subscription of event messages when user attempts to filter different criteria for traffic events. When user changes its filter, the topic subscription under *subscriberImage* will be adjusted accordingly.
3. *subscriberMarker*: subscription of event messages based on marker’s geographical location. This is to simulate the subscription of nearby event messages within each of the marker’s reach.

*showNewRect*: add text to *infoWindow* panel in google maps wherever there is any changes.

*deleteMarkers*: a function to empty and delete assigned markers on the map.

*filterMarkers*: a function to filter and clear categories of markers on google maps. On google map, any changes that is made to filtering will call this function. There are two different types of filtering of markers on the map: category and image filtering.

1. *category*: category changes (temperature, patrol etc.) made on markers will clear any markers being shown on the map
2. *image filtering*: any changes made in value for each description tags will have subscriberImage to re-subscribe to the relevant filtering to narrow down the event messages that will appear on map.

*infoWindowUpdate*: any markers that is selected on the map will have the panel on the left to be updated. On the other hand, it will also keep track of subscription area of the marker on the map. If the marker moves out of its boundary, *subscriberMarker* has to re-subscribe its topic. (Refer to *google.maps.rectangle*)

*InitializeHospital*: a function that initialize static hospital data on the map. Firstly, this data will be assigned as a marker on the map. Secondly, we assigned each marker the ability to subscribe as *subscriberMarker* when it is clicked. The rectangular bound around the assigned marker will be shown.

*InitialiseTempChart*: Shows the aggregated average of all the temperature sensors on the map. The function will be called if topic is subscribed and message is received.

*pubsubmap.js*: Similar to *pubsub.js*, *subscriberMarker* and *subscriberImage* and subscriber are given the ability to connect to Solace Cloud and subscribe. When messages are received, it will be categorized into different function to get messages registered as marker on the map.

*selectorFiltering.js*: Contains the function needed to get messages assigned as marker on google map. It performs a few functions:

1. *selectorTaxi*: Assign each message for a patrol unit as a marker on the map. When clicked, *subscriberMarker* will be linked to its location to enable remote subscription of event. If the marker exists on the map, it will update its location instead.
2. *selectorTemperature*: Similar to *selectorTaxi*, each marker on the map will be updated with its latest readings. On the other hand, the temperature chart will be initialized when start and update its data chart for any incoming messages.
3. *selectorRain*: *selectorRain* firstly checks for any existing sensors on the to update its rain status. Else, the rain marker will be displayed on the map based on its assigned value. If the value is 0 (means no rain), the marker on the map will be removed.
4. *selectorTemperatureChange*: If the message with topic of temperature change is received, the marker will change its icon temporarily for 60 seconds before reverting back.
5. *selectorImage*: Display event marker on map when message is received, including update on Info Panel and content on *InfoWindow*.
6. *addMarkerEvent*: This function is called when the selected marker moves out of its geographical boundary. *subscriberMarker* will attempt to re-subscribe with its new position and set new subscription topic to receive correct incoming event messages.

**test.js**

The diagram for introductory instruction to display how the data flows from agencies to central broker vice versa. It is used in *homepage.html* and *index.html*.

The test.js uses a few functions to display the diagram. Firstly, *test.js* creates a new pane to initialize a diagram, followed by creation of nodes and linkage of nodes. Nodes will begin to simulate in a data flow manner through fire transition, simulating sending of messages from one node to another.

**login.html <- Frontend | Backend -> login.js**

An interface for agencies to log in their details as a broker if they have established their connection to the central broker. Otherwise, the user will need to create one (via *register.html*) and has to be verified by the administrator. In this login page, the application can either bring its client to admin central (*admin.html*) or agency broker page (*homepage.html*).

**register.html <- Frontend | Backend -> processregister.js**

A page where agencies can create their connections to the central broker. In this section, agencies have to provide credentials and information to register their access control and bridge to the cloud. The registration message will be published through *processregister.js*. It must be approved by the administrator before it can start establishing any bridges to central broker.

**homepage.html (Consists of other client pages)**

Once agencies login, they will be greeted with data exchange panel where it shows a small introduction of its demonstration. This page is only accessible by the agency that is registered on central broker.

**registertopic.html <- Frontend | Backend -> processtopicreg.js**

In this section, established agencies can submit topics that they like to publish onto the central broker. The submission message will be published through *processtopicreg.js*. However, these topics must be approved by the administrator before it can start publishing their specified topics.

**Admin.html <- Frontend | Backend -> log.js (All Sections) / AccessListCall.js (Topic Requests) / BrokerRegistration.js (Register Agency)**

In this page, the administrator has 3 main sections of functionality: *Message Log*, *Register Agency* and *Topic Requests*. Each section is linked to a solace connection under *log.js* individually to consume any messages that are coming from client. (Such as messages published into central broker, registration of bridges and topics are consumed from different queues etc.) *Message Log* describes all the data that has been consumed by central broker. It will receive the message as well as the topic destination from different brokers.

When an agency broker registers itself or a topic to establish relationships with the central broker, the administrator will have to approve it in order to create ACL and bridges with its client. These lists will be shown on *Register Agency* and *Topic Request* section. If the administrator approves the establishment of broker for creation of ACL and bridge under *Register Agency*, the program will execute *BrokerRegistration.js* function; If the administrator approves the publish topic under Access Control List, *AccessListCall.js* will be executed.