**Sayso Bot – Functional Guide**

**Introduction:** Every company conducts surveys for different purposes. **ITC InfoTech India Limited (I3L)** is not an exception to that. **I3L** conducts surveys for variety of reasons like getting feedback of training program, service desk, ERP and other portals, new product launch, event organization, company performance and many more. Right now different department used to deal with these surveys separately, i.e. the surveys are not under single umbrella. The purpose of this project is to make all these surveys centralized and conduct them from one single place which is nothing but the **chatbot** application. After completion, response of all surveys will get stored in **SharePoint**, and on top of that data we are going to apply analytics to figure out what all are going well and what the areas of improvement within the organization are.

Every bot is a **web application**. Like other web applications, Sayso also has 3 primary building blocks; **User Interface** (UI), **Application Logic** and **Backend Services**. In this document we will focus on UI and application logic.

The components of Sayso can be broadly categorized into 3 parts;

1. **Sayso Chat UI**
2. **Sayso Offline Connector**
3. **Sayso Botservice**

Let get into details of each component one by one;

**Sayso Chat UI:** This is the view (UI) component of the application. User responds to surveys through this chat UI. In case of cloud deployment, Azure provides **webchat** for this purpose. Here we have customized the webchat and then deployed to our on-premise server (**msweb02**). Following are the important aspects of this component;

1. This is the customized version of **MS webchat** (build version).
2. The customizations are mainly done on **styles**, customization related to functionalities are reflected on **script**.
3. There are 3 primary parts of this chat UI;
   * 1. index.html
     2. botchat.css (styles)
     3. botchat.js (script)
4. This component is using node package “**botframework-directlinejs**” internally to communicate with botservice (**Sayso Botservice**). Please refer to following URL to know more details about this node package; *https://www.npmjs.com/package/botframework-directlinejs*
5. Any backchannel activity is received in “**index.html**” file using **directline** instance. To know more about backchannel activity, please refer to following URL [*https://docs.microsoft.com/en-us/azure/bot-service/nodejs/bot-builder-nodejs-backchannel?view=azure-bot-service-3.0*](https://docs.microsoft.com/en-us/azure/bot-service/nodejs/bot-builder-nodejs-backchannel?view=azure-bot-service-3.0)
6. Any **event/activity** is posted to backend **botservice** using same **directline** instance.
7. Following are the list of customization performed on styles and scripts;
   * 1. Disable controls of adaptive card after first write operation (**botchat.js**).
     2. Message from bot come with name as “Maya” (**botchat.js**).
     3. Styles of buttons and cards like border, color, width, display, etc. property (**botchat.css**).
     4. Remove buttons from adaptive cards after first click (**botchat.css**).
     5. Date format of adaptive card from “**mm/dd/yyyy**” to “**dd-mmm-yyyy**” (**botchat.css** & **botchat.js**).
     6. Post an event to backend with userid and username to authenticate user and initiate conversation from bot’s end (**index.html**).
     7. Receive events from botservice to show progress bar, current survey name and add line-break after survey completion.
8. This chat UI component is deployed as an express application, where the application servers **/index.html** on request.
9. This component knows the address (host URL) of connector component. Using this address only UI communicate with the service.

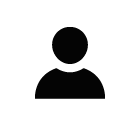
**Sayso Bot Architecture:**

**MSWEB02**

**Café Xpress Page**

**Sayso Botservice**

**Sayso Chat UI**



**User**

**iframe**

**Sayso Chat Container**

**Sayso Connector**

**Sayso Offline Connector:** This component is the offline version of MS **botconnector**. This component acts as a bridge between the chat UI and botservice. Following are the primary insight of this component;

1. This component receives user’s message from chat window, sends it to backend service, receive response from botservice and delivers them to chat window.
2. This component uses node package **offline-directline** to perform the operations. For more information, please refer to the following link; [*https://www.npmjs.com/package/offline-directline*](https://www.npmjs.com/package/offline-directline)
3. This component is developed and hosted as an **express** application.
4. This component conatins 2 URLs;
   * 1. Host URL of **botservice**
     2. Host URL of connector itself
5. In order to function properly, both chat UI and botservice needs to be up and running. Otherwise this component will throw **UnhandledPromiseRejection** exception.

**Sayso Botservice:** This component is heart of our bot. All the functional logic related to bot is implemented in this component. This component is developed using MS Bot Framework. Following are the insights of this component;

1. This component uses “**botbuilder**” package to implement bot functionalities and “**restify**” to create server instance.
2. This component receives user’s message from **offline-connector** component.
3. In order to get survey list, survey questions, submit response, etc. this component use different web APIs.
4. The widgets to present survey list, survey queries are pushed from this component. This is not related to rendering message to chat, it just tells **what** to render not **how** to render.
5. This component sends event/activities related to survey progress, ongoing survey name and line break after completion of survey.
6. This component receives 2 parameters from chat UI;
   * 1. **Userid** – *session.message.user.id*
     2. **Username** – *session.message.user.name*

**Café Xpress page to Hosting Sayso:** Cafe server deploy **Sayso** and **MayaHR** bot as separate webparts. So in café page there are 2 primary components related to bot;

* MayaHR Webpart
* Sayso Webpart

**Sayso Webpart Components**:

1. Bootstrap Popover
2. Sayso container to load iframe
3. Refresh button to reload chat
4. Close button to end conversation

**Sayso Webpart Funtionality:**

1. Get encrypted user token (Sharepoint token) from Café page API (via Ajax call).
2. Get **GUID** from Azure service by passing encrypted token.
3. Get logged in user’s name.
4. Get official email of logged in user.
5. Get no of active surveys (**count**) by calling API with user email.
6. Display popover with **username** in the title and **survey count** with some message in popover body. Give a link to load sayso chat.
7. On clicking popover, sayso container will get loaded which holds sayso chat UI as iframe.
8. The iframe accepts **userid** and **username** as 2 query parameters.
9. Sayso container displays username at the top of container along with refresh and close button.
10. Refresh button reloads the chat with new **GUID**.
11. Close button makes the sayso container disappear and refresh the chat.
12. On closing sayso container, popover will get displayed again along with updated survey count. If there is no survey at **pending/saved** state, then popover will not get shown.

**Botservice of Sayso:** Functionalities of Sayso is implemented in botservice component. Following are the primary tasks performed by botservice;

1. Authenticate User
2. Display Active Survey List
3. Post Remind Me Later Message
4. Present Survey Questions One by One to User
5. Submit Survey Response
6. Submit Partial Response
7. Handle Branching Logic
8. Handle “**Take All**” action

**Authenticate User:**

1. Get hold of GUID by accessing variable ***session.message.user.id***.
2. Pass GUID to Azure service to get encrypted user token.
3. Decrypt the token using **AES-CBC-256** algorithm (**crypto** node module).
4. Pass the original token (**Sharepoint** Token) to café service to get user details.
5. Store user details in memory.

**Display Active Survey List:**

1. Get list of active surveys from API (/iSurvey/api/GetActiveSurveySummary/{**email** }).
2. Iterate through each survey, if remind me option is enabled then get next reminder date and store then in memory as a list (as **conversationdata**).
3. Push all survey **name** to adaptive card along with corresponding **no of days to expire**.
4. For each survey, add a button in adaptive card with label as **survey name** to initiate that survey.
5. If no of surveys to take greater than 1, then add a “**Take All**” button to card.
6. If all active surveys has remind me option enabled and all of them expiring in future dates, then push a “**Remind Me Later**” button to card, otherwise don’t give this option.

**Post Remind Me Later Message:**

1. If remind me option is enabled for a survey, then calculate the next reminder date by checking “**RemindMeDays**” field of survey object.
2. If reminder date of any survey falls in weekend, **prepond** the date to **last Friday** of week.
3. On clicking “**Remind Me Later**” button, the surveys along with **listid** and **next reminder date** gets posted to web API.
4. Survey **name** along with next **reminder date** gets displayed on chat window on clicking the button.

**Present Survey Questions One by One to User:**

1. Get list of survey queries by calling API with Id **(*/iSurvey/api/GetSurveyById/{id}***).
2. If survey status is **pending**, then begin the survey from very first question.
3. After accepting user’s response of a question, store them in memory as an array of object.
4. All survey queries gets stored in bot’s memory after fetching from API. They gets stored in memory as an array of objects.
5. After accepting each response from user, bot examine the branching logic associated with that query (if any).
6. After accepting all answers from user, set of response along with queries gets submitted to backend by calling web API.
7. During the survey, after a specific interval**, partial response** gets submitted with status set as “**saved**”. On completion, the status is changed to **completed**.
8. After accepting each response from user, bot sends % of completion status as an event. This event is caught and handled in front-end to show the **progress bar**.
9. At the beginning of survey, another event is passed to front-end to display the **ongoing survey name**.
10. After completion of every survey, another event is passed to add a **line-break** to chat window.
11. If survey response status is saved, then “**PartialResponse”** field of survey object gets examined and resume point gets decided by checkingalready answered questions and corresponding branching logic.

**Submit Survey Response:**

1. Upon completion of each survey all survey questions along with their response gets submitted via web API.
2. Submission status will be marked as “**completed**” in response object.
3. After successful submission, user gets an confirmation message followed by a line-break denoting end of survey.

**Submit Partial Response:**

1. For each survey, partial response gets submitted after a specific interval (The interval is mentioned in **.env** file).
2. In case of partial submission, survey status will be marked as “**saved**” in response object.
3. If user closes the chat window/page before completion of survey, then partial response will get saved and next time that survey will get **resumed** from the position upto which **partial response was submitted**.

**Handle Branching Logic:**

1. After accepting each response from user, branching logic of that query object gets examined by checking “**Branch**” property of question object.
2. If no branching logic is present, then index of current question gets incremented by 1.
3. If present, then logic is examined and based on that next question is decided.
4. In case of partial survey, at the time of resuming, the last answered question object is examined. If there is no branching logic associated with that query, then automatically the very next question to be asked.
5. If present, then survey resume point is examined by checking submitted answer of last question (**partially saved** one) and corresponding branching logic is survey question object.

**Handle Take All Action:**

1. If there is more than 1 active survey is present for a user, then only he/she will get “**Take All**” option.
2. **Take all** will conduct all active surveys one after another.
3. After completion of each survey, there will be confirmation message along with a line-break.
4. Partial response and complete response functionalities are same as other surveys.
5. After completion of all pending/saved survey, user will get a final confirmation followed by a **goodbye message** denoting end of conversation.