Conversational AI

Overview

Conversational AI

The primary goal of Conversational AI is to receive text based queries from end users (“utterances”), interpret those queries as known or unknown actions(“intents”) via its dialogue engine, and generate and respond to those queries with text based responses.

A digital AI assistant powered by Conversational AI is referred to as an Agent, and each Agent belongs to an Organization. Agents can be created to fulfill specific use cases, for example a customer support chatbot, or internal employee knowledge base.

Management

Conversational AI also comes with Conversate UI (CUI); a powerful Agent management dashboard that provides tools to create and manage intents, train intents on the utterances that should invoke them, and custom tailor the responses given back to end users.

Architecture

The Conversational AI platform infrastructure includes Services for interpreting requests from supported Clients, creating/managing identities for end users interacting with the system, interfacing with third party APIs to gather end user related metadata, and interpreting human language as actions via the dialogue engine. At the core of this architecture is Conversate which is the main hub for the dialogue engine and for end user response generation.

The End User Request guide is a great way to familiarize yourself with the Conversational AI architecture and its services.

Clients

Agents are capable of supporting various end user request surfaces such as SMS messaging from a mobile device, or text messaging via popular third party embedded chat widgets such as Glia. These request surfaces are known as Clients

Each client communicates with the Conversational AI platform via logical pipelines known as channels. These communications are routed through the main gateway to the Conversational AI platform called the Channel Connector.

The Conversational AI platform provides its own set of embedded clients which can be integrated with an existing website or mobile device to provide users with an Agent powered chat widget.

End User Request

At the center of Conversational AI is the request/response cycle that receives human language, text-based queries, and produces response payloads including end user facing text.

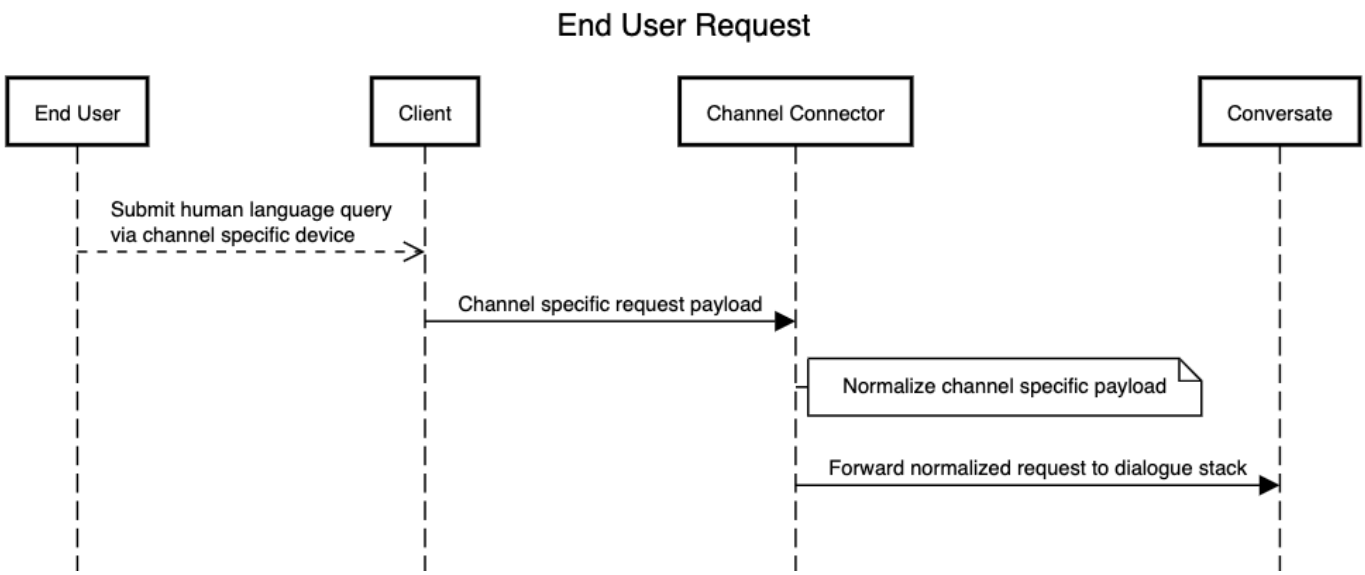
The request/response cycle can be initiated via supported Clients which are routed over logical pipelines called “Channels”.

Generally speaking, each channel will have its own physical device or digital application that end users interface with to provide human language queries (aka their Clients). For example, a user can communicate with Conversational AI using their mobile device as the client which consumes the SMS channel.

Request

All incoming client requests are first received by the Channel Connector regardless of the client type. Then Channel Connector converts the channel specific payloads to a format that the dialogue engine can consume. The dialogue engine payload request includes the original end user’s utterance, and any additional request metadata

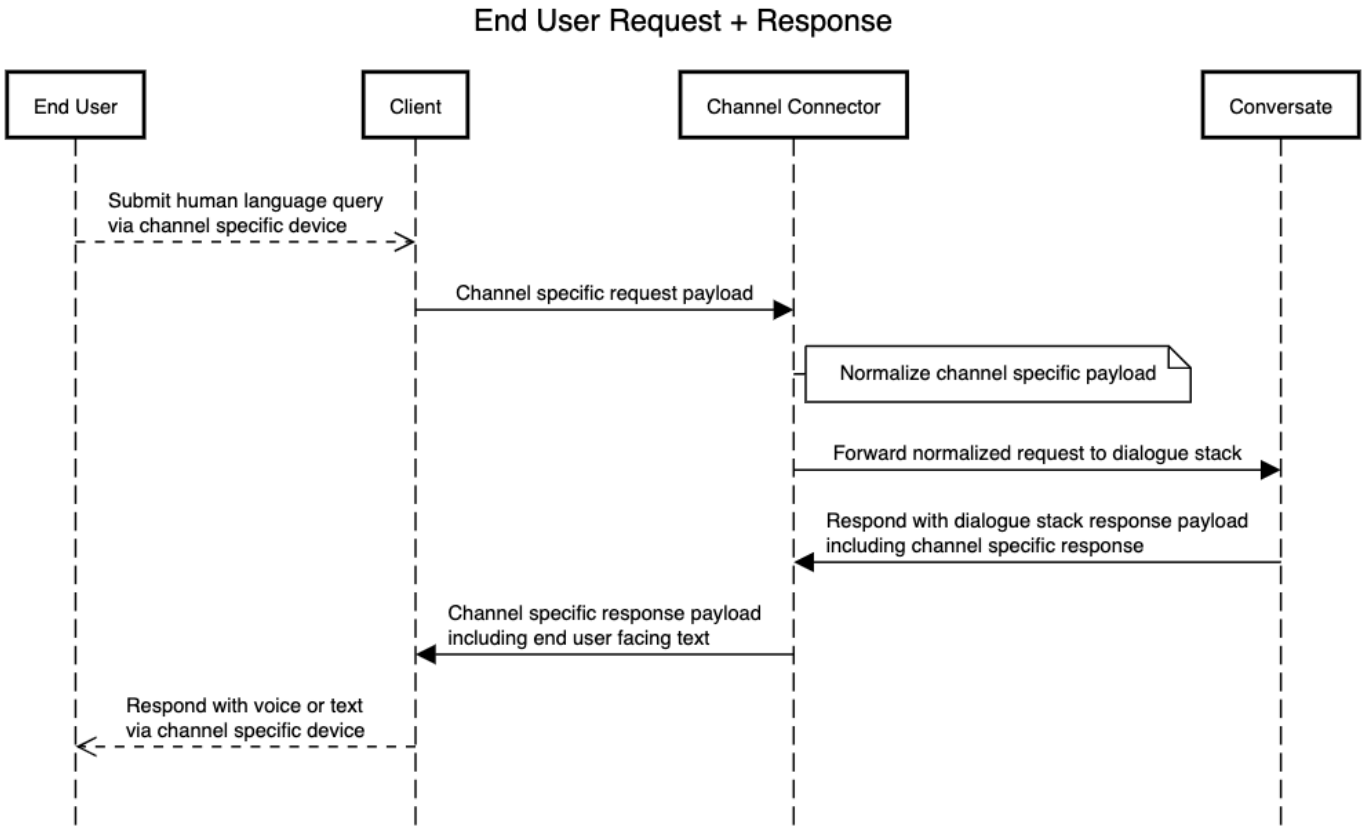
Channel Connector will pass normalized payloads to Conversate which creates and manages “Conversation Records” for sets of interactions based off of request context. Conversate is also able to determine if additional information is needed from the end user, or third parties on behalf of the user, and is able to modify request flows asynchronously to gather required information.



Response

Once a request is parsed and sent to Conversate all necessary intents will be invoked, and a normalized response payload will be returned to Channel Connector. These response payloads include end user facing response text, but are not formatted in a way that can be understood by the end user's client.

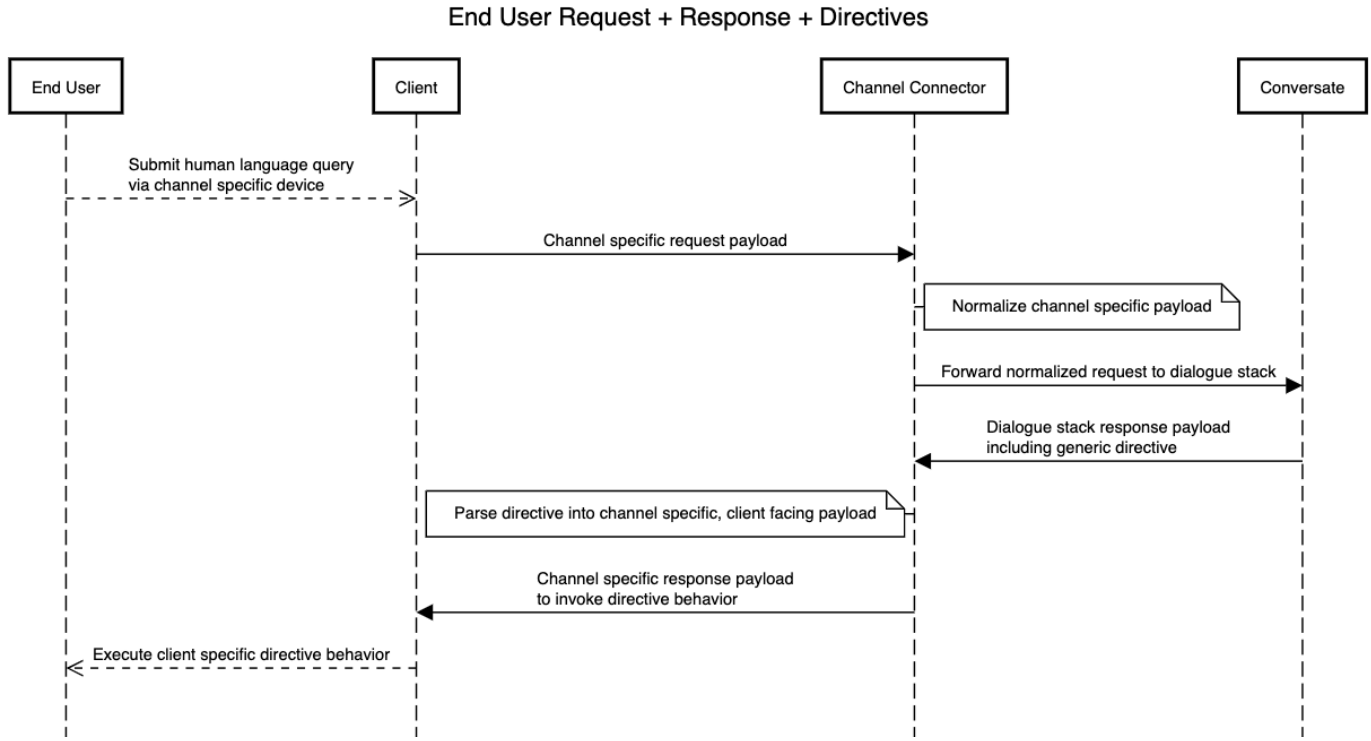
The Channel Connector is responsible for parsing all normalized response payloads returned by Conversate following the channel provider's specifications. These channel specific response payloads are then returned to the client, and the end user will receive the intended response text.



Directives

During the course of a conversation it may become necessary to invoke system or device specific behavior to facilitate an end user's request. An example would be if an end user requested Personal Identifying Information (PII) or secure data (such as account and balance information), and the system was not authorized to gather that data on behalf of the user. In this case, the Account Linking directive would be returned with a response payload.

Directives are propagated through the Conversational AI system as simple flags or metadata payloads (for example “accountLinking=true”. It is the responsibility of the Channel Connector to invoke the required behavior within all Clients by converting system directives into channel specific flags/metadata/response payloads.



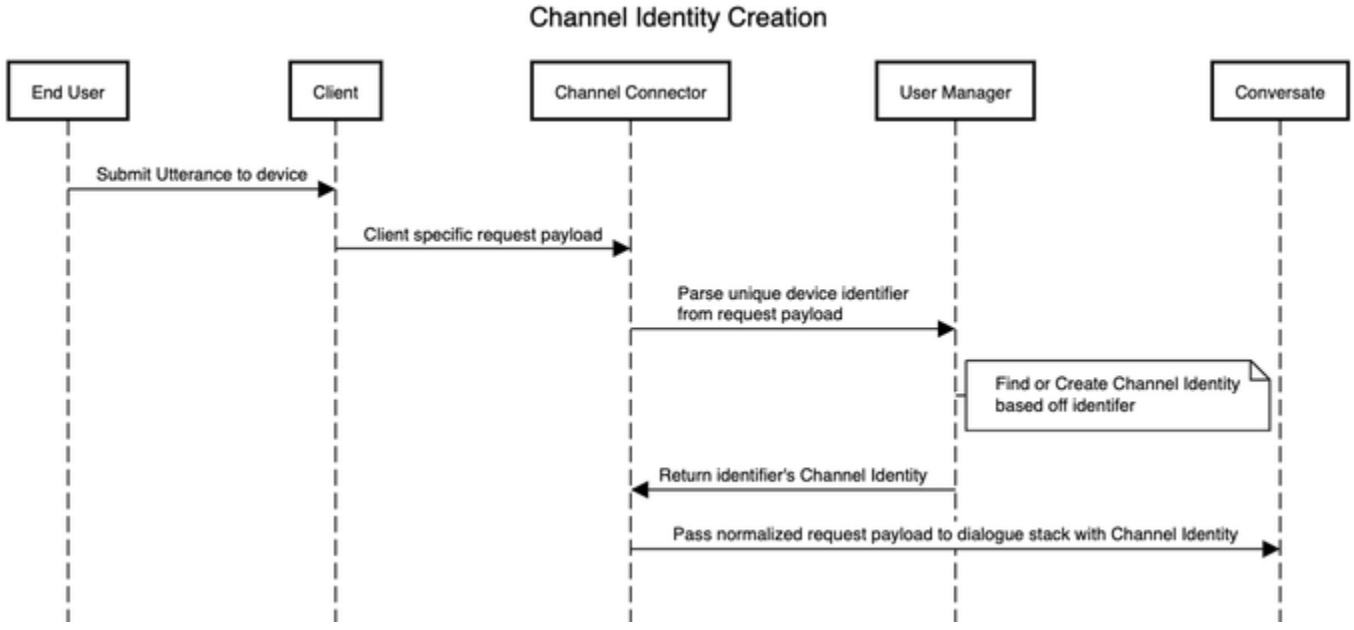
End User Identification

From the beginning of a conversation end users have virtual identities created and assigned to them. Whether a user is requesting PII, or if they’re simply asking a frequently asked question, it’s important to be able to unique identify them within the Conversational AI system.

Channel Identity Record

Each of the supported Clients has a unique identifier attached to the device or instance. For example, a phone number is used as an identifier when end users are communicating with an Agent using their mobile device over the SMS channel. Likewise, Embedded clients (Embedded Web /Embedded Mobile) will also assign device identifiers to each of their digital instances.

These unique identifiers are used to update or create what is known as a Channel Identity, a unique identity representing the device itself; a process that happens at the beginning of every single incoming request to the Conversational AI system.

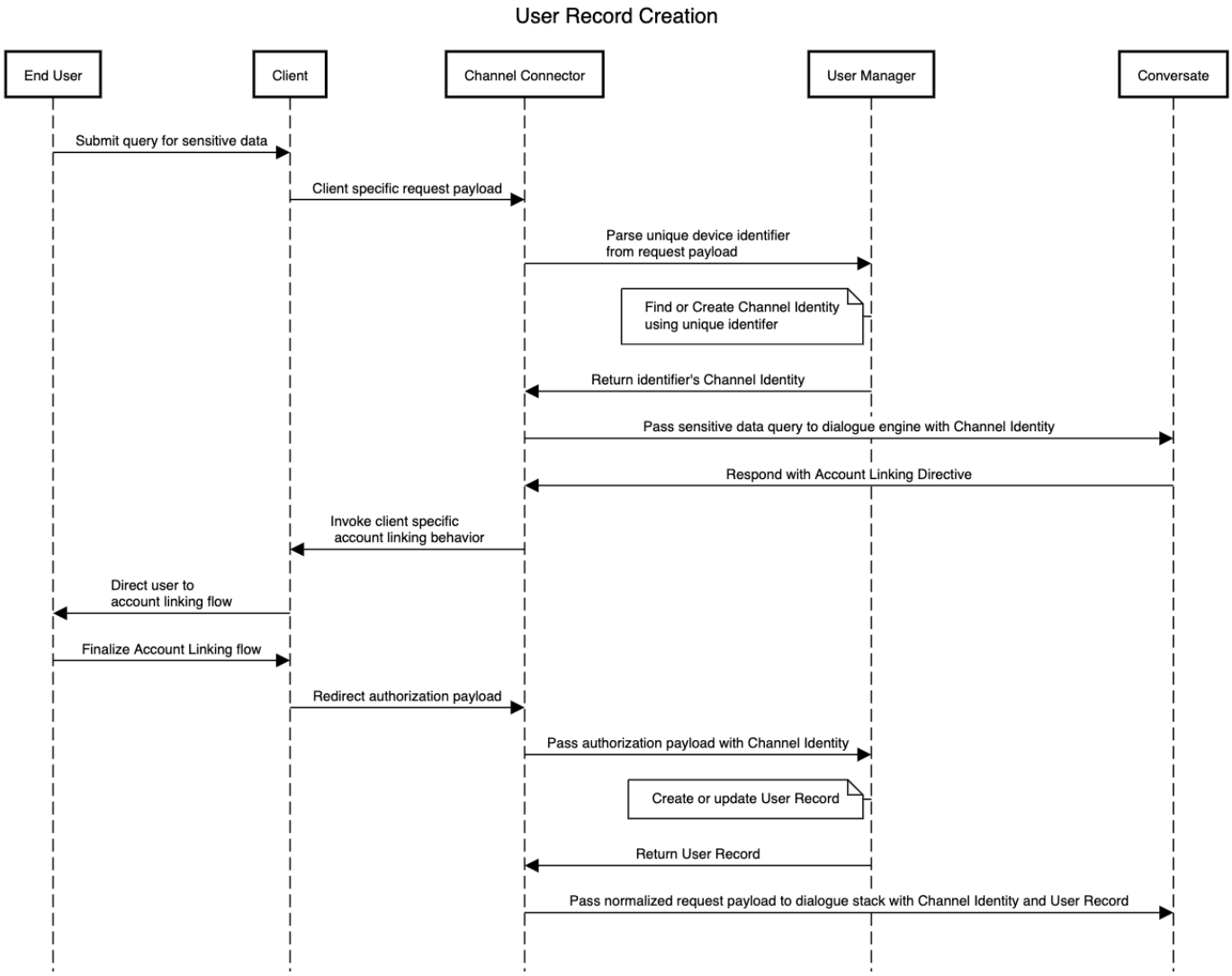


Channel Identity records are important because response generation, and conversation management, require knowledge of the user on a per request basis. Since conversations themselves are an abstract concept representing a set of turns in the End User Request cycle, the Conversational AI system needs some way to keep track of the user between turns. This is the critical role that Channel Identity records play.

User Record

Channel Identity records are critical for identifying devices that have communicated with the Conversational AI system, but additional identities are required if an end user wants to retrieve PII data from an Agent.

End users must complete an Account Linking flow to grant authorization to an Agent. The Agent will then be able to gather third party data on the end user's behalf. At the end of this flow, the newly gathered credentials will first be used to gather provider specific profile data belonging to the end user. Included in third party profile data is a unique value used by the provider to identify the user within their own infrastructure. This value, and the credentials gathered from the Account Linking flow, are combined and stored as a User Record. Additionally, an association is made between the User Record and the Channel Identity that initiated the Account Linking flow.



By gathering this third party metadata, creating a User Record, and associating it to the end user's Channel Identity, an Agent can identify the end user as a known third party data provider's user. The credentials stored within the User Record can then be used to gather third party data, and the end user can be addressed using their provider derived profile information (first/last name, etc).

Conversation Fulfillment

The core functionality of Conversational AI is to receive text based queries ("utterances"), interpret these queries as known or unknown actions ("intents"), and to respond with pre determined text based responses.

Utterances

An utterance is any ambiguous human language text received from an end user. While it is possible to send automated queries to an Agent using Quick Replies, the vast majority of requests handled by Conversational AI are initiated by an incoming utterance from an end user.

Utterances can be made up of any set of human speech. It is the job of the dialogue engine's NLU to tokenize and interpret this ambiguous text as some known action the Agent is able to perform, or a request for information the Agent is able to retrieve on behalf of the user. These actions are referred to as "intents".

Intents

An intent can be seen as a preprogrammed action invoked by an utterance which produces a set of variant text based responses, and potential directives such as a request for end user Account Linking, or an initiation of a Human Handoff session.

An Agent can be trained to invoke a specific intent by configuring each intent with a variety of sample utterances. These utterances are given to the Agent's machine learning architecture, and its AI training data set is updated to be able to identify similar utterances as an intent invocation.

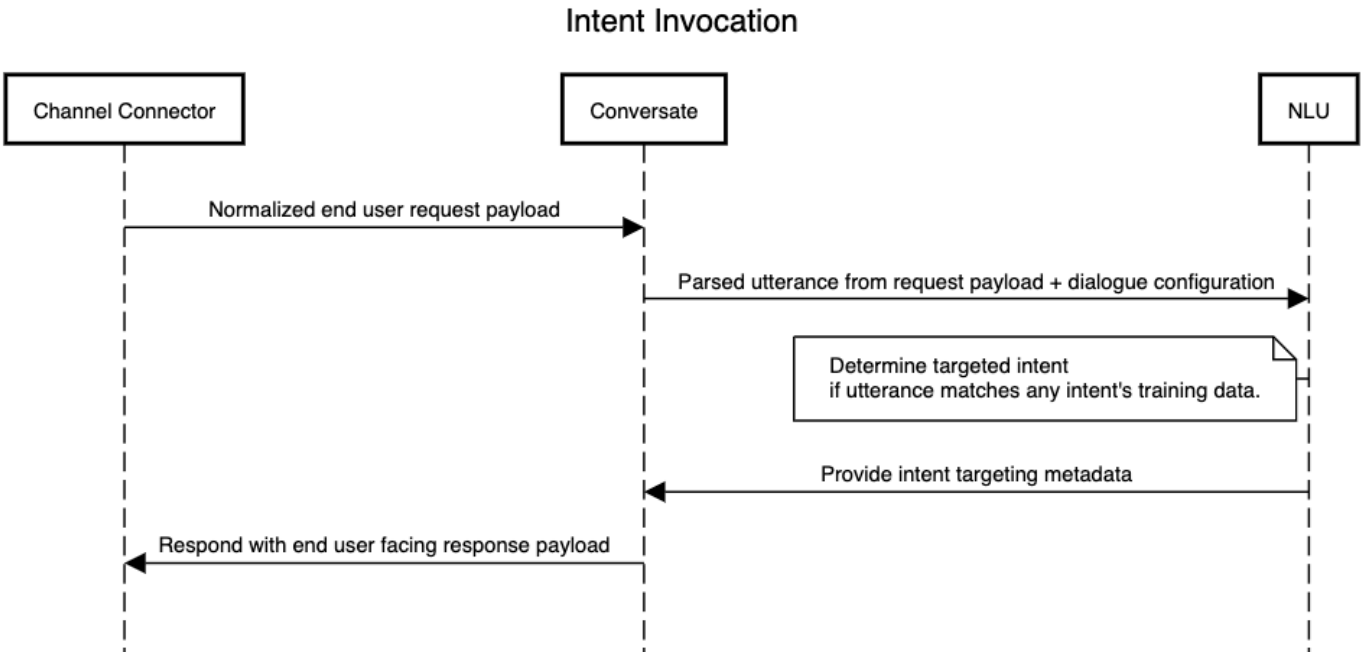
Administration of an Agent's intents is handled via the Dialogue Explorer component of the CUI Agent management dashboard.

Conversation Flow

As described in End User Request , all utterances are routed through the Channel Connector and converted to a normalized payload for Conversate which will determine which conversation the dialogue is a part of, and how the utterance should be handled within the context of the conversation.

Conversate will first redirect the utterance to the NLU along with a set of training data. The NLU will determine if the utterance matches with a known set of sample utterances belonging to an intent.

Any matching intents will be invoked, and end user facing response text will be gathered from them.



Integration Services

When an NLU response payload supplies a targeted intent the Conversate service does not actually perform any intent invocation logic itself. Rather, the Conversate service redirects intent processing, and response generation, to separate services known as Integration Services.

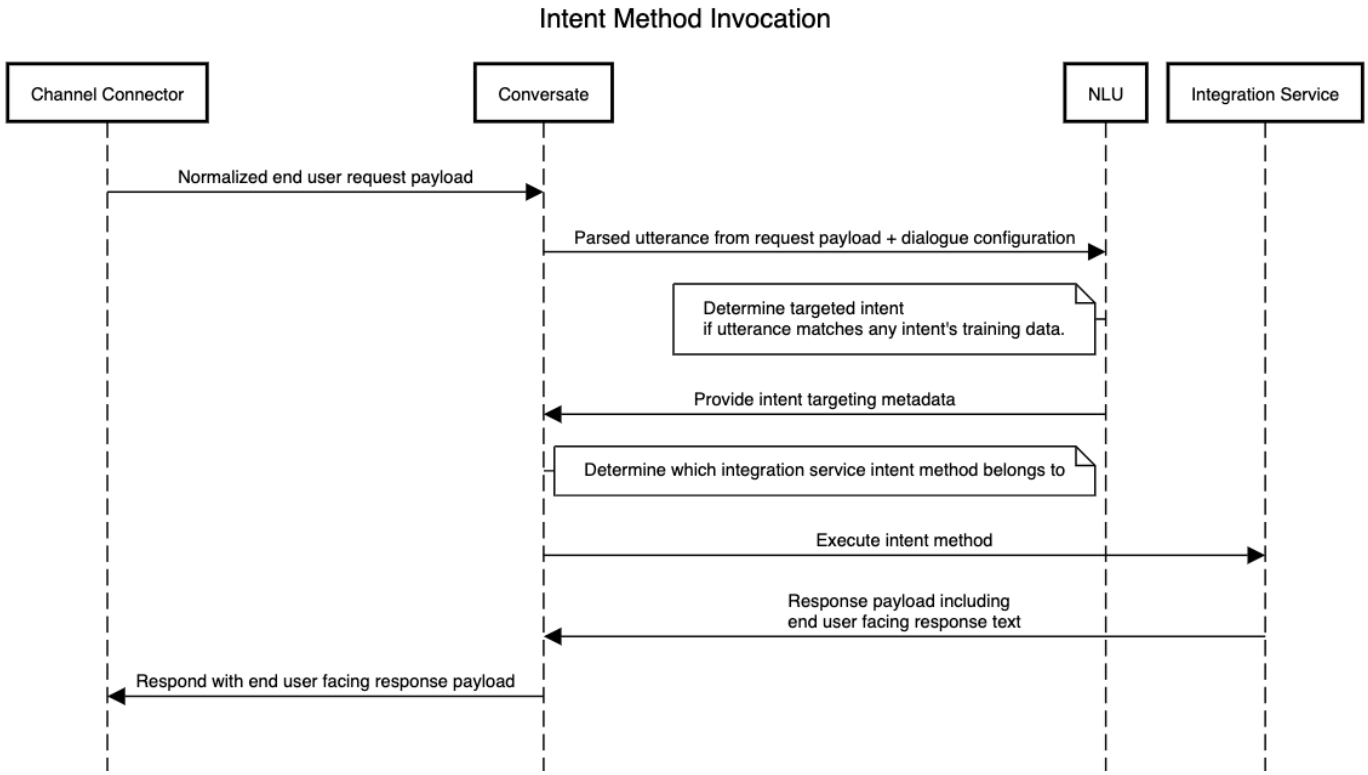
As their name implies, Integration Services are the integration point for third party data consumption as well as custom Agent behavior. They are responsible for determining if intents are able to be executed based off of the conversation's context, processing end user responses attached to intents, and hydrating response templates with dynamic data related to the conversation.

Integration Servicesalso provide the ability to execute custom programmatic logic that may be attached to an intent, for example directly requesting end user data from a third party API.

Intent Methods

An intent can include simple text responses, but they can also be configured to re route the conversation to subroutines called Intent Methods.

These packets of functionality allow the Agent to interface with third party data providers, securely gather data from them on behalf of the user, and determine what steps should happen in the remaining conversation for example invoke additional intents or return intent response text. Intent methods are provided, and executed by, Integration Services.



Entities

During the course of a conversation an end user may request details about a specific bank account, or the Agent may require contextual information from the end user to facilitate a request. Any piece of information that can be identified by the NLU, and mapped to a known data model, is referred to as an “Entity”.

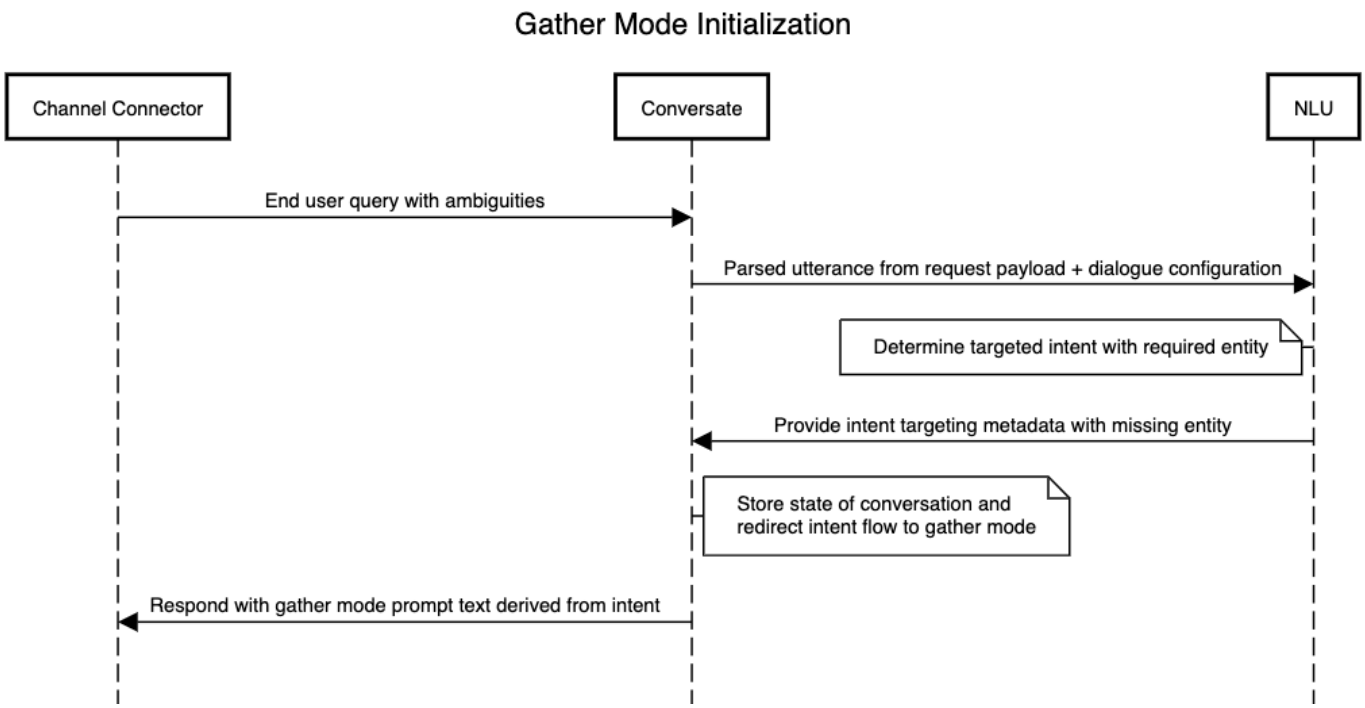
Gather Mode

Some entities are gathered on behalf of the end user upon the completion of the Account Linking flow, for example account entities (name, last four digits). This information is supplied to the NLU during the entity recognition process to determine if an end user is referencing a known entity.

Other entities need to be gathered from the end user, or data provider, to generate an Agent response. For example, if a request required an additional passcode for security an Agent could request a generic number entity from the end user.

When an Agent requires this extra information from a user, it puts itself into a state known as Gather Mode. While in this state, an Agent will not accept additional queries during the current conversation until the required entity data is provided by the user.

The response text used to prompt the user for missing entities is configured by dialogue engineers and directly attached to the intent being gathered for.



Assistant Management

Overview

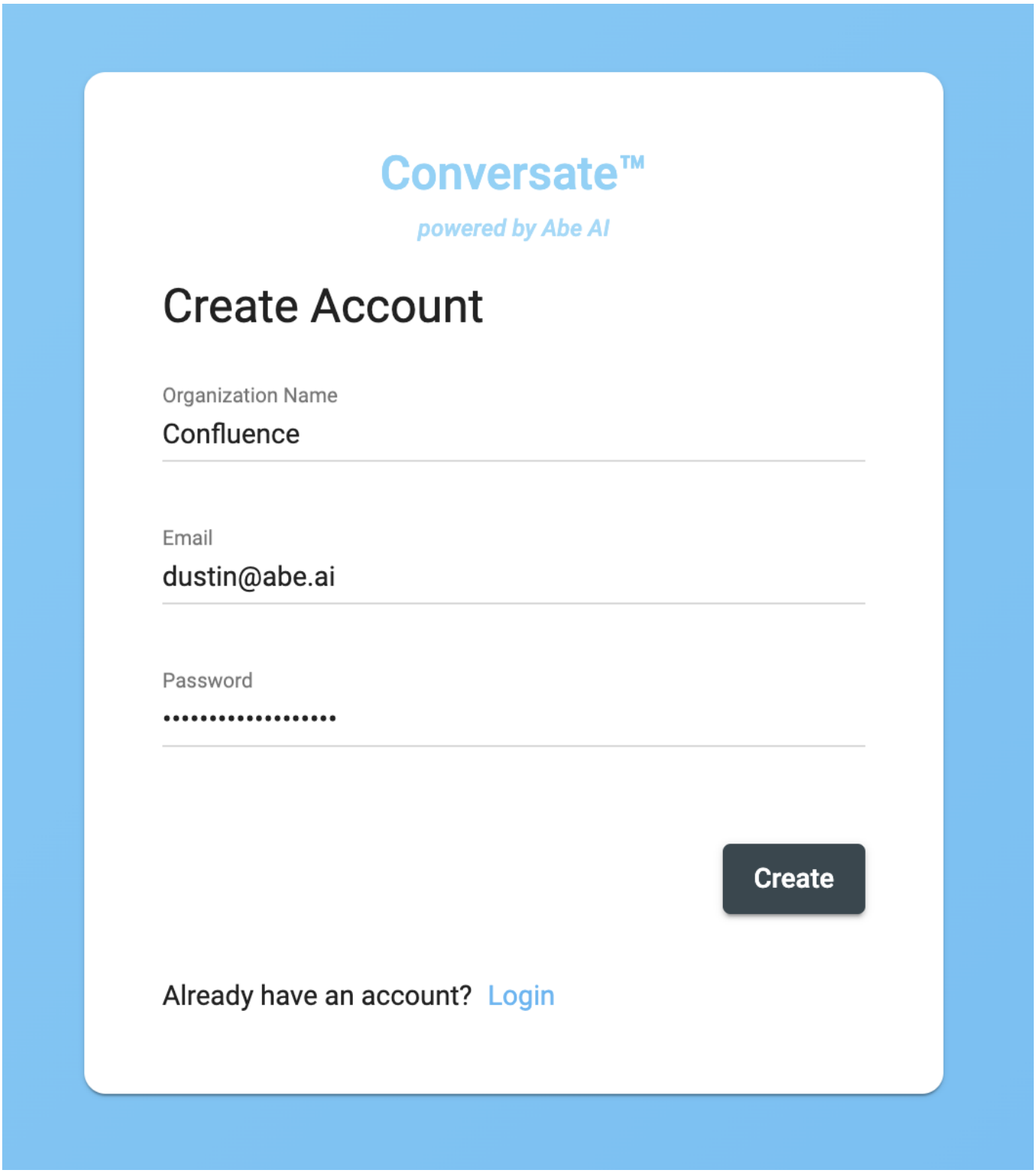
Assistant Management encompases setup and management of Assistant settings, channel settings, authentication settings, and conversation flow and design. Assistants belong to Organizations and are created and managed via the CUI client application.

General Hierarchy

- Organization
  - Admin Application
  - Assistant
    - Integration Services - Managed by CUI'sIntegration Services Management view
    - General Settings - Managed by CUI's General Assistant Settings view
    - Dialogue Settings - Managed by CUI's Dialogue Settings view
    - Channel Settings - Managed by CUI's Channel Settings view
    - Human Handoff Settings -Managed by CUI's Handoff Settings view
    - Campaigns -Managed by CUI's Campaign Manager view
    - Test User Settings -Managed by CUI's Test User Settings view
    - Authentication Settings -Managed by CUI's Authentication Settings
    - UI Applications (Conversation Designer, Contact Center)

Create Organization

In order to create assistants, an organization must exist first. To create an organization, you must go to CUI app with the URL path `/account/create`.



Under the hood, the create button will trigger the https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637185039/Auth+State#CreateAccount action. After successful completion of that action, users will be redirected back to login. Once logged in, you will be on CUI's App Picker view.

ABE AI

Organization: **Confluence**

Assistant

First Agent

Select an Application

Admin

You will notice that an assistant “First Agent” has already been created. This occurs as part of our organization creation process. You will also notice you have access to the Admin App as your account was marked as an administrator during the organization creation. You will also notice that the the First Agent does not provide any applications (this is probably needing to be changed as the assistant should probably have Conversation Designer by default).

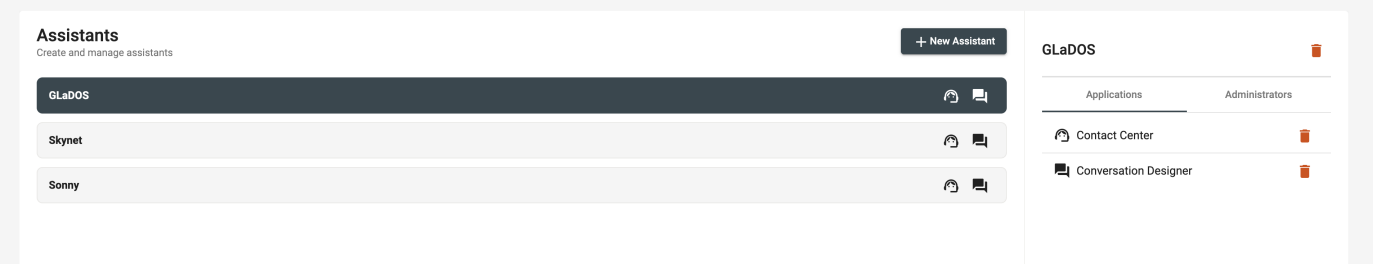
Creating Assistants

Prerequisites

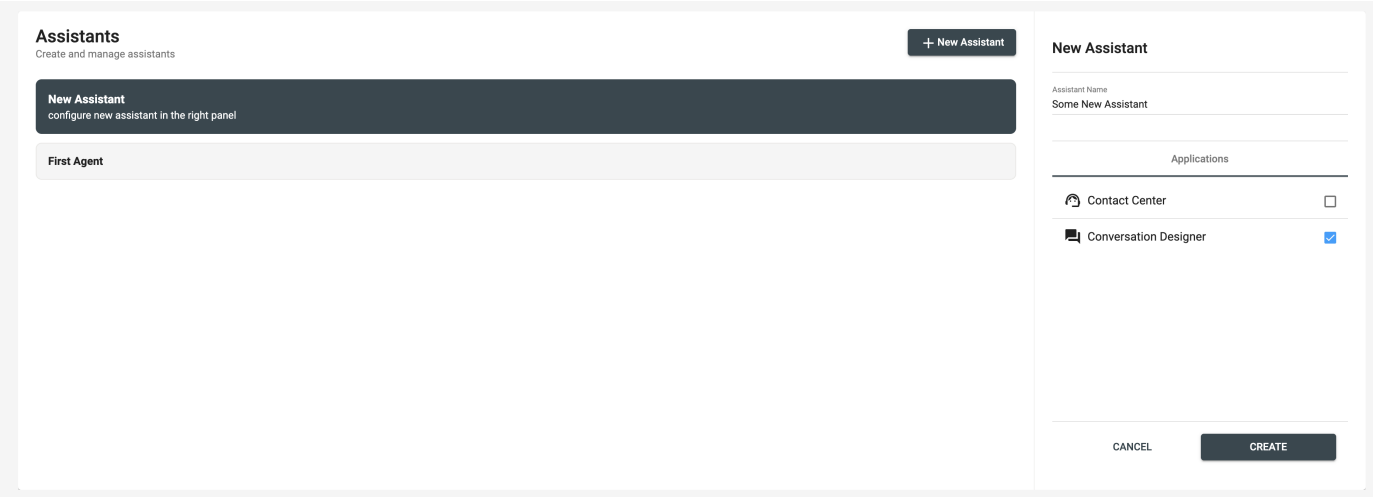
- Access to an Organization
- Access to the Admin Application

Process

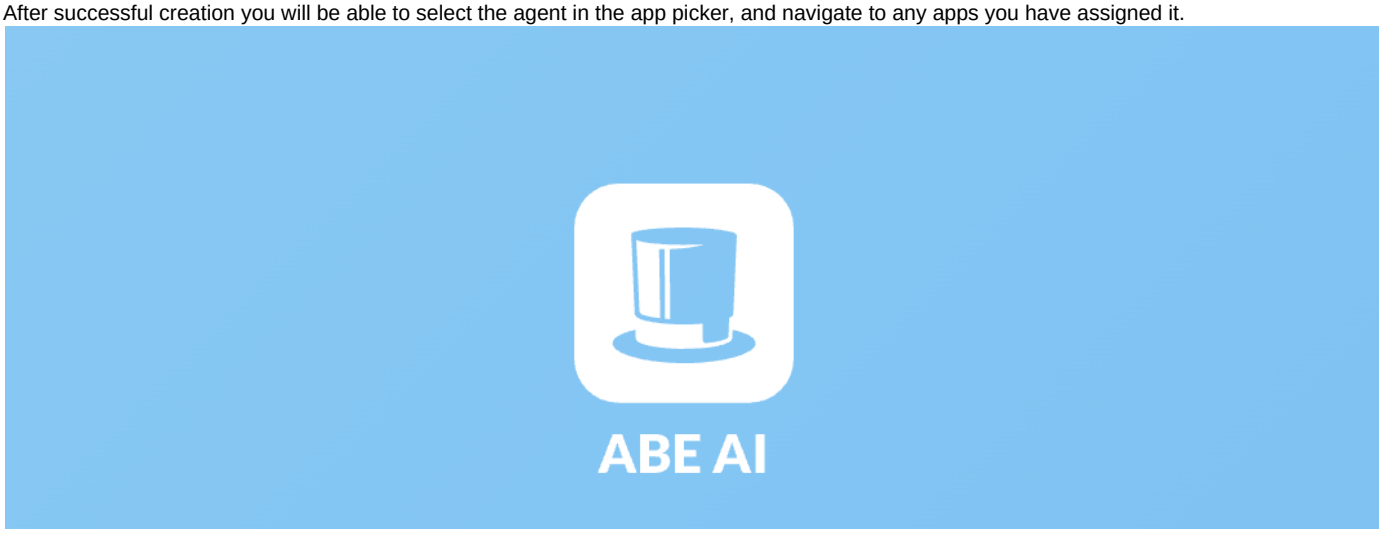
To create Assistant’s you must go to the Assistants view inside the Admin App.

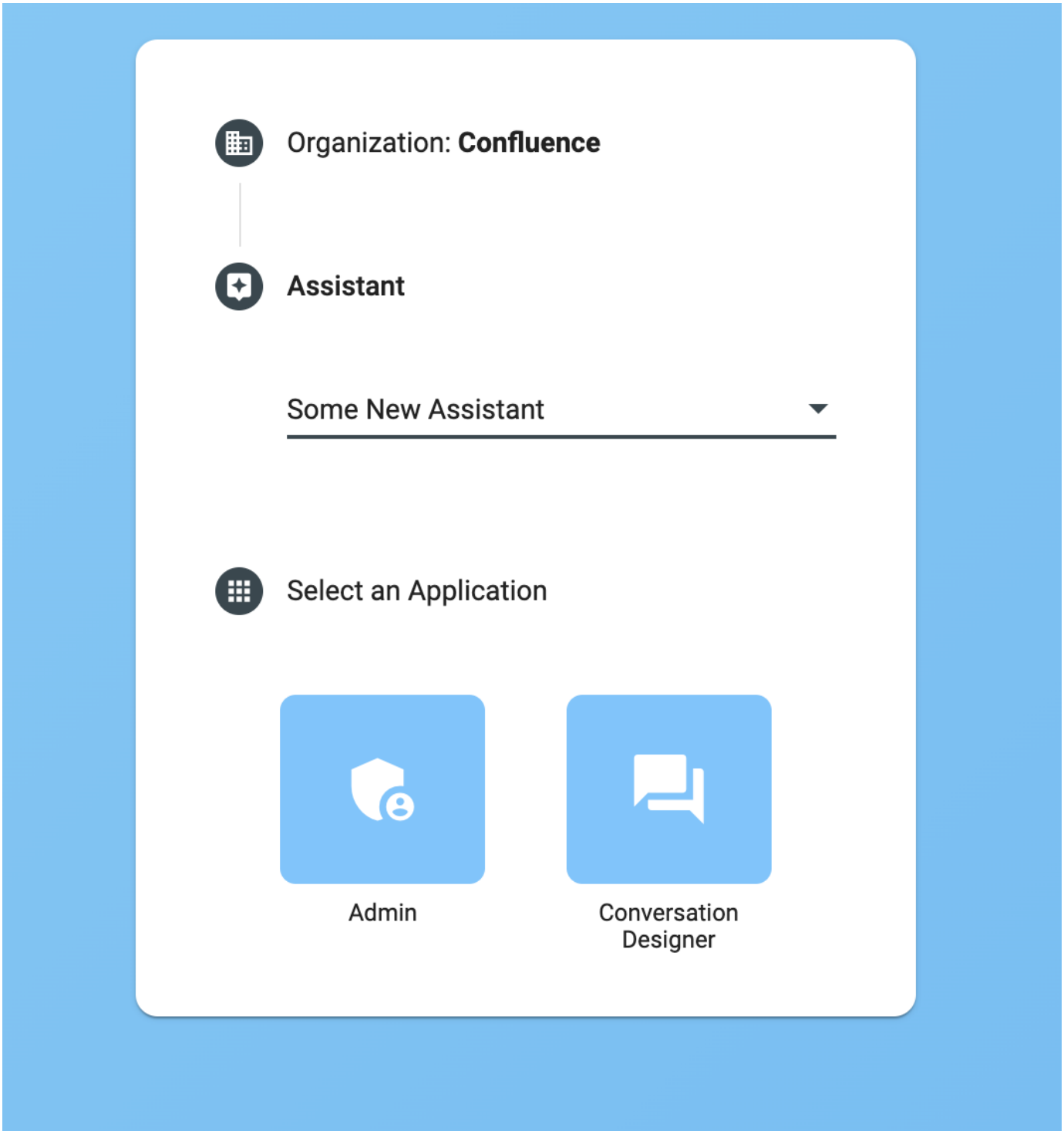


There you will be able to add applications to the assistant. To create a new Assistant, at least 1 application is necessary to be added. Upon pressing create, the https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637152263/Authenticated+Account+State#CreateAgent action will be triggered which will send the create agent payload to Conversate.



After successful creation you will be able to select the agent in the app picker, and navigate to any apps you have assigned it.





Assistant Settings

Once an assistant is created, there are various settings that can be set. These settings can all be configured via CUI's Agent Settings module.

Conversation Management

Assistants need to have their intents and training data set up in order for end users to conduct a meaningful conversation. Intents, and their training data are configured via CUI's Dialogue Explorer module.

Clients

Overview

Agents are capable of supporting various end user request surfaces such as SMS messaging from a mobile device, or text messaging via popular third party embedded chat widgets such as Glia. These request surfaces are known as Clients

Each client communicates with the Conversational AI platform via logical pipelines known as channels. These communications are routed through the main gateway to the Conversational AI platform called the Channel Connector.

The Conversational AI platform provides its own set of embedded clients which can be integrated with an existing website or mobile device to provide users with an Agent powered chat widget.

Table of Contents

Contact Center

Overview

Contact Center is a front end service that allows human assistants to conduct live conversations with end-users. It supports connections to conversations in both Embedded Web and Embedded Mobile. It utilizes Twilio's Flex platform and utilizes Channel Connector as a pass through for communication between the Human Agent and the end user.

Users can access Contact Center through the CUI application selector, provided the user has been assigned to an appropriate User Groups and has the appropriate Permissions .

Table of Contents

Twilio Flex

This React application handles much of the data retrieval and organization into its own established Redux store. From making websocket calls for basic engagements, to handling video calls and account management, Twilio Flex covers a majority of our needs.

App Initialization & Configuration

Overview

Contact Center's Redux store is broken into a few different segments and have added functionality for async updates.

Additional information about Redux pattern can be found here.
Twilio themselves have also provided a detailed guide to using the store in this application here.

Human Agent features

Human agents have a limited set of features within Contact Center. They are able to:

- Accept or Deny incoming human hand off requests
- See the end user's chat bot history
  - Only the current chat bot history is displayed to the human agent
- Text chat directly with end users
  - Available to both Embedded Web & Embedded Mobile
- Video chat with end users
  - Video chat requires the microphone and camera permissions from the browser
- Co-browse with end users
  - This includes:
    - Viewing the end users current screen
    - Ability to draw on an end user's screen
  - This requires the explicit permission from the end user
  - Available to Embedded Web
    - Feasibility to Embedded Mobile is being explored
- View their performance metrics
  - These metrics are generated by the Flex platform
- Fill out and complete an Agent survey at the end of an interaction with an end user
- Set their current status

Manager Features

- Includes the features of a Human Agent

- Ability to observe an active human hand off engagement
  - This allows managers to view current human hand off engagements and the ability for a manager to join in an engagement to further assist the human agents
- While observing a conversation, have the ability to whisper to a Human Agent
  - This allows direct communication between the Manager and the Human Agent without the End User seeing the messages
- While observing a conversation, have the ability to join an Engagement with an End User
  - This allows a manager to join an existing Engagement between a Human Agent and an End User
- Create or Delete Queues
- Enable the Everyone queue
  - This is a unique queue where every human agent that is assigned to the current Contact Center instance, can be sent a human hand off engagement even if the human agent is not assigned to a particular queue.
- Configure Queues & teams
  - This includes for Queues:
    - Engagement Order
      - First in, First out
        - The earliest received engagement is given priority
      - Last in, Last out
        - The latest engagement is given priority
    - Set Human Agents' reservation activity
      - This allows the human agent's status to be automatically changed when an incoming engagement is received
    - Set Human Agent's Assignment activity
      - This allows the human agent's status to be automatically changed when an incoming engagement is accepted
    - Set the Max Reserved Agents
      - This determines how many agents in a queue can possibly receive an incoming engagement request
        - The minimum is 1
        - The maximum is 50
    - Choose which survey a Human Agent and Customer receives based on the queue
    - Assign Teams to a Queue
  - This includes for Teams:
    - Creating a Team
    - Assigning Human Agents to a team
    - Assigning a team to a queue
- Survey Management

- This includes for surveys:
  - Creating and editing a survey
  - Creating and editing questions for surveys
  - Preview a survey
  - Adding questions to surveys
  - Deleting a survey
  - Deleting a question
  - Choosing the Survey type
    - Human Agent
    - Customer
- This includes for Questions:
  - Creating a new question
  - Editing an existing question
- User Management
  - This includes the ability to:
    - Set a Human Agent's Display name
    - View the Teams a Human Agent has been assigned to

CC - Tech Stack Details

Contact Center is built as a plugin for Twilio Flex, an existing application base that has both frontend and backend components.

Twilio Flex

This React application handles much of the data retrieval and organization into its own established Redux store. From making websocket calls for basic engagements, to handling video calls and account management, Twilio Flex covers a majority of our needs.

Twilio Flex Plugin Documentation

Twilio Flex Chat Documentation

Material UI

As our plugin replaces most of the visual elements of the Twilio Flex application, Contact Center leverages Material UI components to create our various pages, forms and interfaces.

Emotion

Contact Center makes use of Emotion's React package, so that when coding specifically styled components, using them looks and feels like simply using a normal React component.

Twilio Flex Plugin Integration

Contact Center is built upon a React application base called Twilio Flex.

Basic information for making a plugin can be found in Twilio's documentation here.

App Initialization & Configuration

In `AbeContactCenterPlugin.tsx` we define some important configurations that control how the app looks and behaves.

Component Configuration

Much of the base application has been overwritten and replaced by custom components due to various needs that the base application did not meet. This is controlled in the `modifyFlexViews` section of this initialization.

```
flex.MainHeader.Content.remove('logo');
flex.MainHeader.Content.add(<AbeHeaderContainer key="AbeContactCenter-
HeaderContainer" />);

flex.TaskListContainer.Content.replace(
  <AbeTaskListContainer key="AbeContactCenter-TaskListContainer" />,
);
```

Here is an example of the adding, removing, and replacing components in this configuration.
For replacing and removing, its important to know/find the correct name of the existing Twilio component.
You can find a list of these components in the Twilio Flex documentation here.

Component Replacement Quirks

Replacing existing components has some quirks that must be kept in mind around matching the same structure of input as the original component.

```
import { ITask, TaskContextProps } from '@twilio/flex-ui';
type Props = TaskContextProps & OwnProps & StateToProps &
DispatchToProps;

// start of the component:
const AbeTaskCanvasHeader = (props: Props) => {
```

AbeTaskCanvasHeader component is a good example of this, wherein we want to get data to this component from the parent that renders it, in the same manner as the original component did. To do this we investigate what those properties are by checking Twilio Documentation or investigating Twilio Flex's code directly, and then import those interfaces and build on them for our component.

Routing

Additional routes are registered in this same application initialization process as the component configuration. Routes are constructed in `src/services/abeRouting.tsx` which gets its list of routes from `src/components/index.tsc`. Individual routes' configuration can be observed by investigating their `.route.tsx` files.

Store Initialization & Configuration

This application uses a Redux store that is managed by the base Twilio Flex application. Contact Center makes small adjustments to it to fit its needs.

The store has been expanded to include a multitude of various data points in different state segments, these are defined thoroughly in the `src/store/index.tsx`. The store's extra reducers and states are registered on application initialization by reading from this mapping and can be expanded easily by following the same semantics and adding to the array.

Details for the contents of the store and their functions can be found Redux Store Structure and Details

Redux Store Structure & Details

Overview

Contact Center's Redux store is broken into a few different segments and have added functionality for async updates.

Additional information about Redux pattern can be found here.
Twilio themselves have also provided a detailed guide to using the store in this application here.

Store Structure

The structure is mostly straightforward with each store reflecting a section of the site and/or a particular set of data.

Flex

This is the original store for the application, and controls a significant amount of data regarding everything. Examples of this include who is logged in, what page are they viewing, and the entirety of the engagement (Task) details. This is often interacted with on Contact Center's side when specific details regarding a task or view is necessary.

Error
This store is for creating and tracking various errors. These errors are handled from the AbeErrorHandler component.

Live
This store manages details regarding Twilio Rooms, video calls and cobrowse. This includes the various tracks used for such, configuration, and actions being performed on these features.

Manager
This section is used to keep track of settings and selected tasks of the Manager View, wherein a Manager can watch and participate in multiple chats.

Modal
This section is used to handle modal displays and behaviors as they are generated. These are consumed and displayed/handled from the AbeModal component.

Profile
Keeps track of various user account information needed to direct and initialize the application. This information comes from the AWC user, which can be explained in more detail in our services documentation; Auth Web Client.

Queues
Queue store is for displaying and handling editing of Queues on the Queue page and other related components.

Teams
Queue store is for displaying and handling editing of Teams on the Team page and other related components.

Users
Not to be confused with Profile, Users contains information regarding all agents (that the current user has permission to see).

Surveys
This section is responsible for handling all Surveys and Questions created for use by Agents or customers.

Routing
This fledgling section will be used to handling extra routing information such as which tab / sub-page the user is or was on. Currently only used for the hybrid Teams/Queues pages.

Store Configuration

The store's configuration is defined in `src/store/index.ts`. Though this is the overall structure, individual stores are configured in their associated code sections under `src/store/{the store}/index.tsx`

Async Behavior

To achieve async behavior for updating the stores, such as making a change to an entity like a Queue that needs to be modified in the backend, Contact Center makes use of Redux Promise Middleware. Using this, Actions can be established to not only fire on completion of an API request, but also kick off other actions/API requests on completion as well. Good examples of this exist in the Queues and Teams stores, specifically their actions.

Store Sync Behavior

When some entities are modified from this application, such as Teams or Queues, all other active users of the site must have the updated information on that entity ASAP. We accomplish this through our sync behavior, that can be found in `sync.service.ts`.
We make these sync updates through Twilio's Sync packages, which have us establish a message stream and post or detect update posts through it.
Whether or not a store slice needs to be synced is the presence of a `.sync.ts` file in its directory and the corresponding mapping object in its `.index.ts` file being updated to include a syncHandler entry. This file defines what behavior must occur when a sync event occurs for it.

CUI

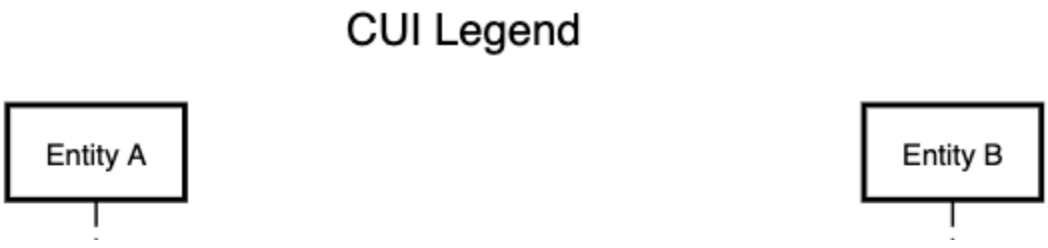
Overview

CUI is short for Conversate UI. CUI is the entry point for anyone on the business side of the product. It provides single sign on (SSO) logins and cookie management, an admin module that allows administrators to control what other users have access to, a module for conversation designers to create intents and how they interact with users, another module for conversation analysts to view past conversations end-users have had, and additional modules for rolling back bad configurations, as well as setting up channels and other Assistant settings. It is all one application, but it is split into a "sub-applications" like structure, and presented to the users as multiple applications.

Service Dependencies

- Connects directly to: Conversate API
- Redirects To: Contact Center

Reading CUI Sequence Diagrams



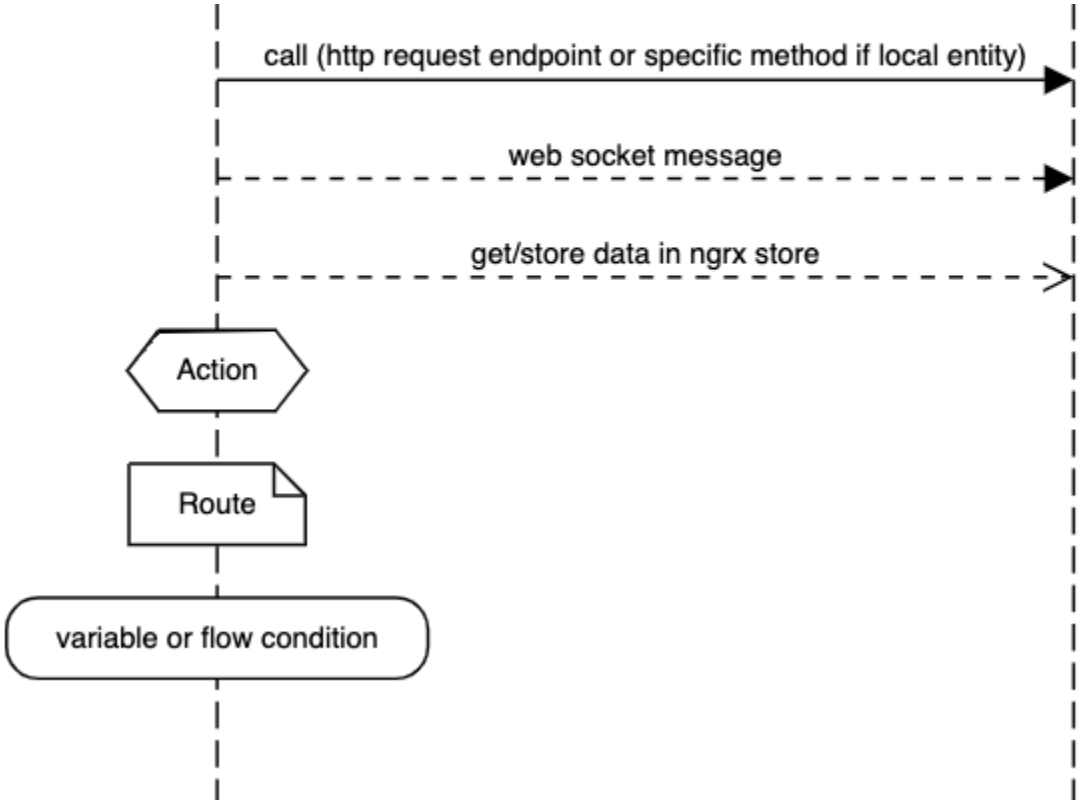


Table Of Contents

- Conversate App (Conversation Designer) — Conversate App module is a rather simple module on its own, whose main responsibility is to act as a proxy router to its sub modules. Conversate App's sub modules have many roles, and these sub modules are where users interact and configure all things related to the setting up the A.I. assistant's, their conversation flows, their enabled channels, and more.
  - Conversate Sub Modules
    - Agent Settings — Provides views to allow users to manage Assistant Settings
      - General Assistant Settings — This view is designed to hold basic settings for the assistant. At this time, it only lets users set the Assistants time zone. These settings are stored in the AWC Agent Config
      - Channel Settings — Channel Settings view allows users to enable/disable channels as well as manage each channels configuration. Each channel may require different set up configuration, and a form will be generated to support the individual channels needs.
      - Dialogue Settings — The Dialogue Settings view lets users perform actions around the Assistant's dialogue flow including setting thresholds for confidence during token matching, and setting global variables.
      - Integration Services Management — The Integration Services management page allows CUI users to add and edit configured Integration Services.
      - Handoff Settings — The handoff setting page is a view that allows user to configure or remove Human Handoff Providers.
      - Campaign Manager — A view to manage campaigns for the assistant.
      - Test User Settings — Configure the test user to be used for the Dialogue Explorer's simulator
      - Authentication Settings — Configure the authentication settings for user sign-in.
      - API Keys — View and manage the assistant's API keys
    - Compliance — Compliance module lists updates to the assistant's dialogue flow (such as intent creation/modification) and settings.
    - CUI Config — A viewless module responsible for housing common logic for cui config management.
    - Debug — Debug module is a hidden module within the conversate app. We do not link to this route from anywhere, users must know the route exists, and manually route there via url. It can be used to set/reset feature flags.
    - Dialogue Explorer — Dialogue Explorer is where conversation designers (CDs) manage and test an A.I. Assistants conversation flow.
    - Dialogue Explorer LINC (deprecated) — a visual node-based editor for conversation training data.
    - Dialogue Route Handler — A simple module designed to handle a shared route routing to conditional modules.
    - Feature Toggle — Module responsible for all things related to Feature Toggles
      - Feature Toggles Editor Component — A component to edit the current state of any registered feature toggle
      - Feature Toggles Service
    - Integration Service — a viewless module containing models and singletons responsible for interacting with integration services
    - Supervisor — The Supervisor module gives the ability to analyze conversations that users have had with the A.I. Agent. It houses a service, a guard, and UI views.
    - Validator (Deprecated)
  - Conversate Store — conversate store is a store that contains data only used in conversate and it's sub modules. It used to be that all sub modules owned their own state reducers and actions, but now Conversate Module owns that code directly.
    - Agent Settings State — The Agent Settings state is responsible for housing information about the assistant's configuration for how end users of the assistant interact with Auth Web Client, and which client channels are connected. The UI to control those aspects are in Authentication Settings and Channel Settings respectively.
    - Compliance State — The compliance state is used to gather and store previous snapshots of the assistant config.
    - CUI Config State — the CUI Config state holds all objects related to managing an assistant's dialogue flow.
    - Feature Toggles State — The feature toggles state is responsible for hydrating and storing the current status of any feature toggle defined in our system
    - Integration Service State (ConfiguredServices) — The integration service state is responsible for housing all information regarding an assistant's configured integration services.
    - Linc Editor State (Deprecated) — The linc editor state is used to store objects needed to run the deprecated linc editor module, including the local working copy of the linc config for the assistant.
    - Navigation State — The navigation state is responsible for storing information about the top nav UI.
    - Supervisor State — The supervisor state houses data needed to run the Supervisor module, including conversations, and the active conversation.
- Stores — CUI's Redux Pattern Structure
- Tech Stack Details — CUI is built on the Angular framework, using version 10.
- Terminology — NGRX Action

Applications

CUI, as an angular based application, utilizes modules to package code. At the highest level we call these modules `applications`. These modules are lazy-loaded by the angular router when a matching route is activated.

Admin App

Admin module is where administrators of Organizations, or Assistants, can assign/remove access for other users, and control permissions.

Conversate App (Conversation Designer)

Conversate App module is a rather simple module on its own, whose main responsibility is to act as a proxy router to its sub modules. Conversate App's sub modules have many roles, and these sub modules are where users interact and configure all things related to the setting up the A.I. assistant's, their conversation flows, their enabled channels, and more.

Core

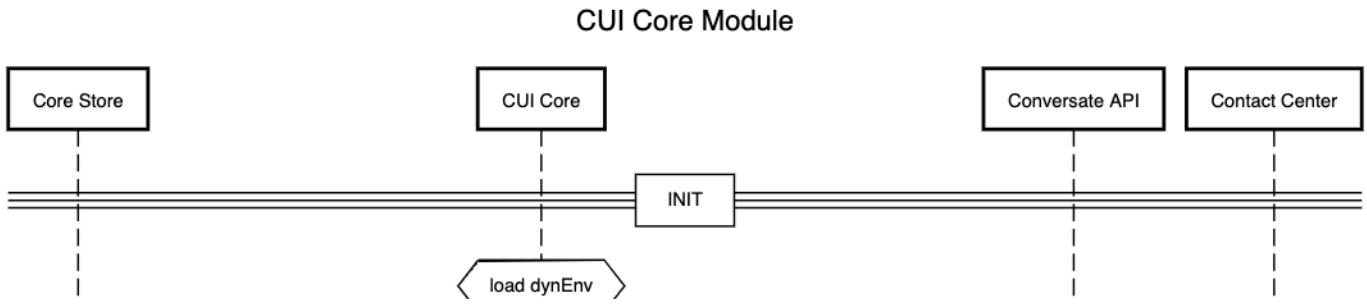
The Core Module is CUI's main module, and entry point. It is responsible for handling user authentication and authorization, and the base router. All routing within CUI starts with the core module router, which sends the router to lower level lazy loaded modules, based on url matching.

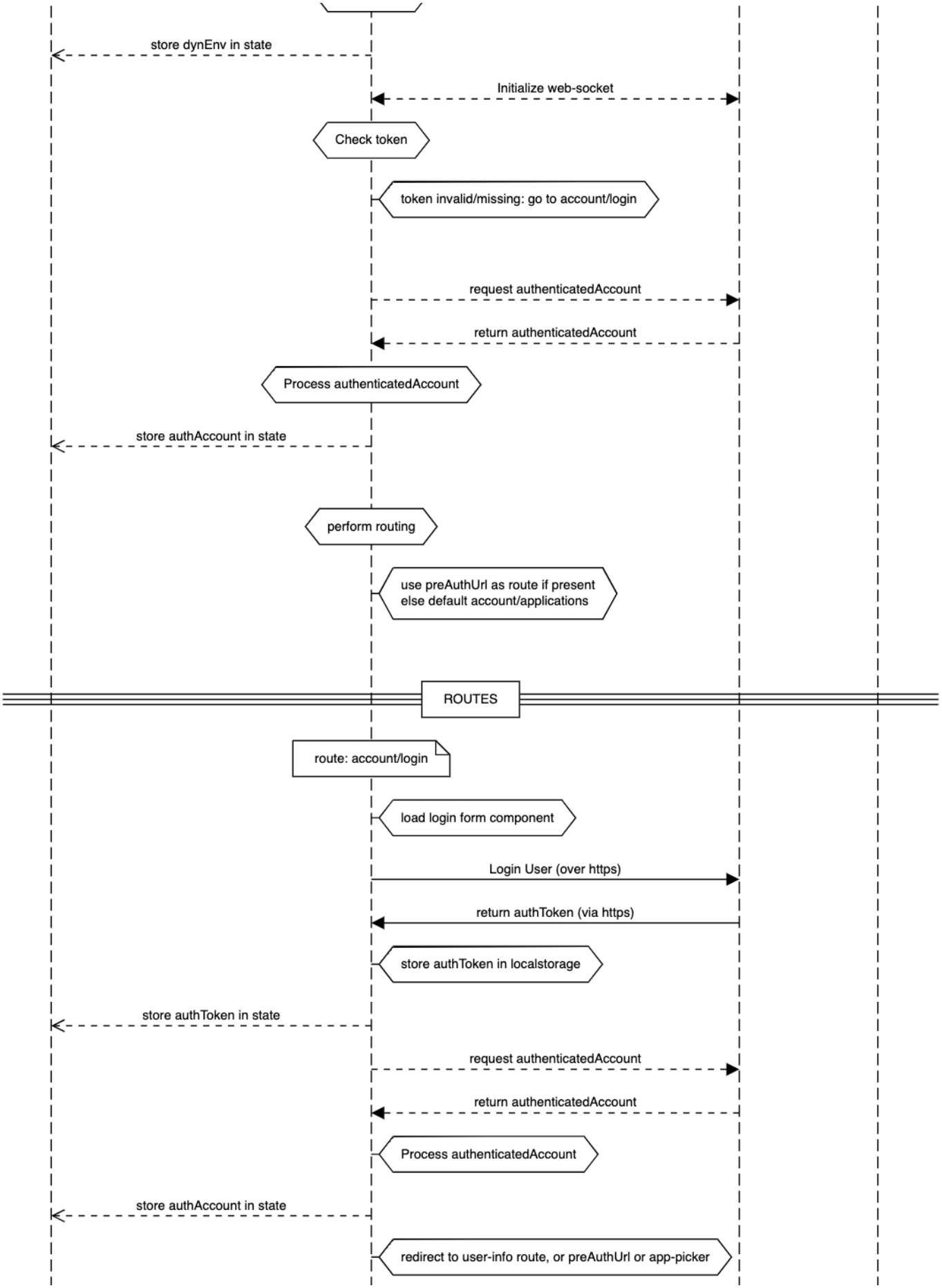
Core

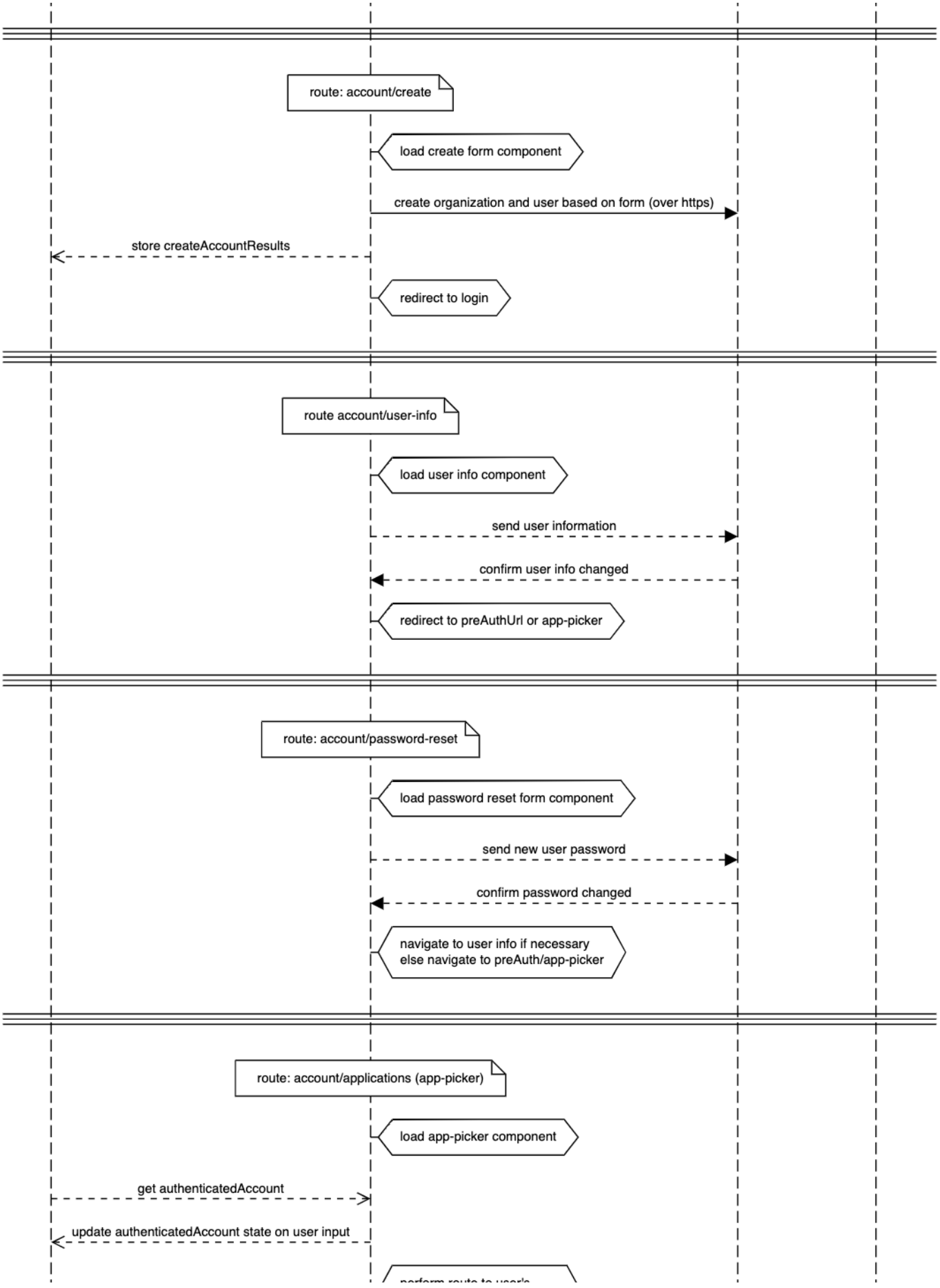
Table Of Contents

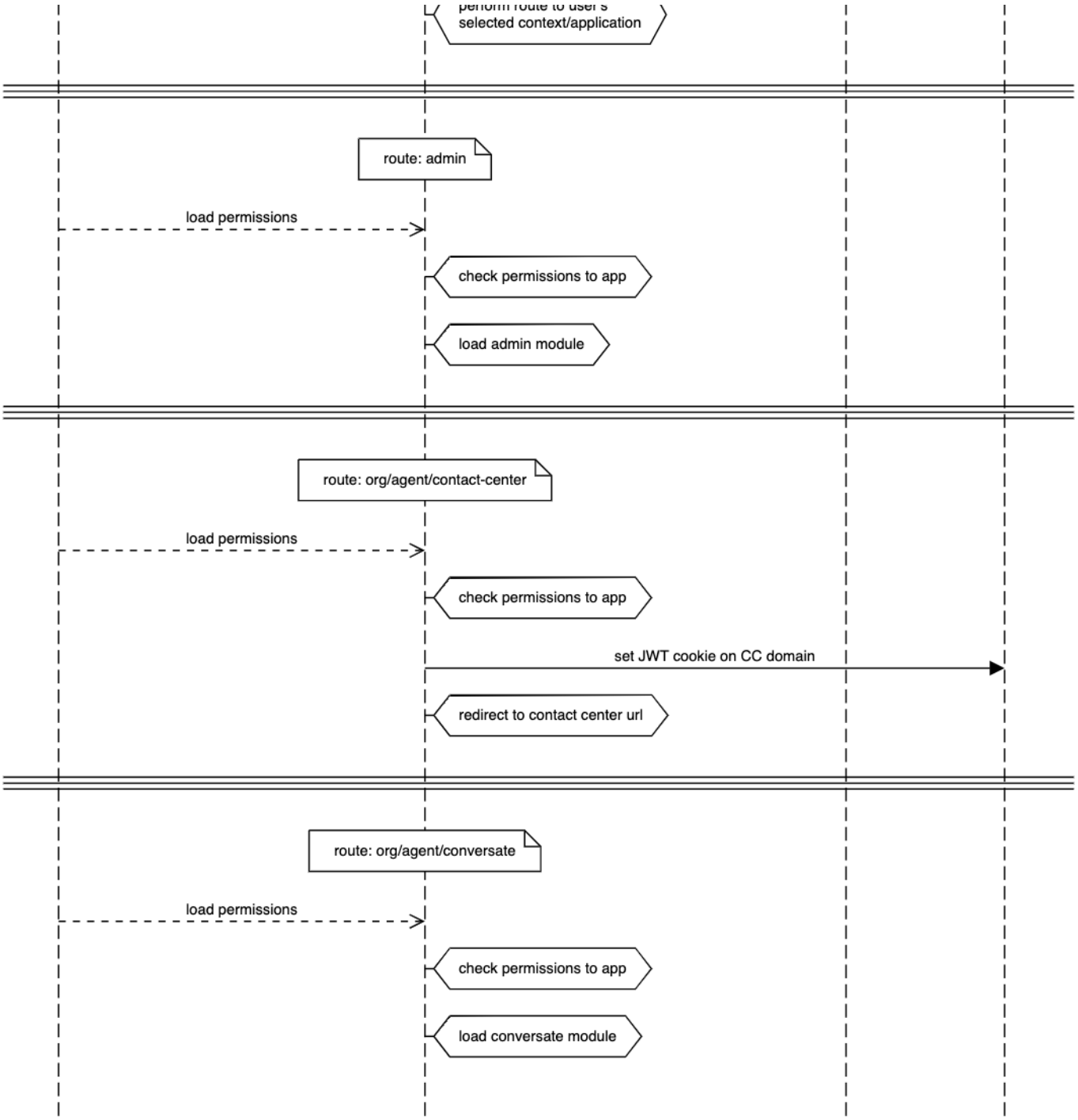
Overview

The Core Module is CUI's main module, and entry point. It is responsible for handling user authentication and authorization, and the base router. All routing within CUI starts with the core module router, which sends the router to lower level lazy loaded modules, based on url matching.




store dynEnv in state
Initialize web-socket
Check token
token invalid/missing: go to account/login
request authenticatedAccount
return authenticatedAccount
Process authenticatedAccount
store authAccount in state
perform routing
use preAuthUrl as route if present
else default account/applications
ROUTES
route: account/login
load login form component
Login User (over https)
return authToken (via https)
store authToken in localstorage
store authToken in state
request authenticatedAccount
return authenticatedAccount
Process authenticatedAccount
store authAccount in state
redirect to user-info route, or preAuthUrl or app-picker


route: account/create
load create form component
create organization and user based on form (over https)
store createAccountResults
redirect to login
route account/user-info
load user info component
send user information
confirm user info changed
redirect to preAuthUrl or app-picker
route: account/password-reset
load password reset form component
send new user password
confirm password changed
navigate to user info if necessary
else navigate to preAuth/app-picker
route: account/applications (app-picker)
load app-picker component
get authenticatedAccount
update authenticatedAccount state on user input



Services

Socket Service

The Socket Service is a singleton object that initializes our connection to Conversate via websocket, and provides methods for sending/handling events over the socket.

Account Service

A singleton service object that provides methods for the Authenticated Account State to connect to Conversate via the Socket Service .

Account Helper Service

A singleton service with 2 methods: `handleCreateUserResponse`, and `_userPasswordResetText`. They are both responsible for loading a modal with specific information into the UI view.

Auth Service

A singleton service object that provides methods for the Auth State to connect to Conversate via the Socket Service .

DynEnv Service

A singleton object that provides a single method, `getDynEnv`, used by the DynEnv State to hydrate the env variables.

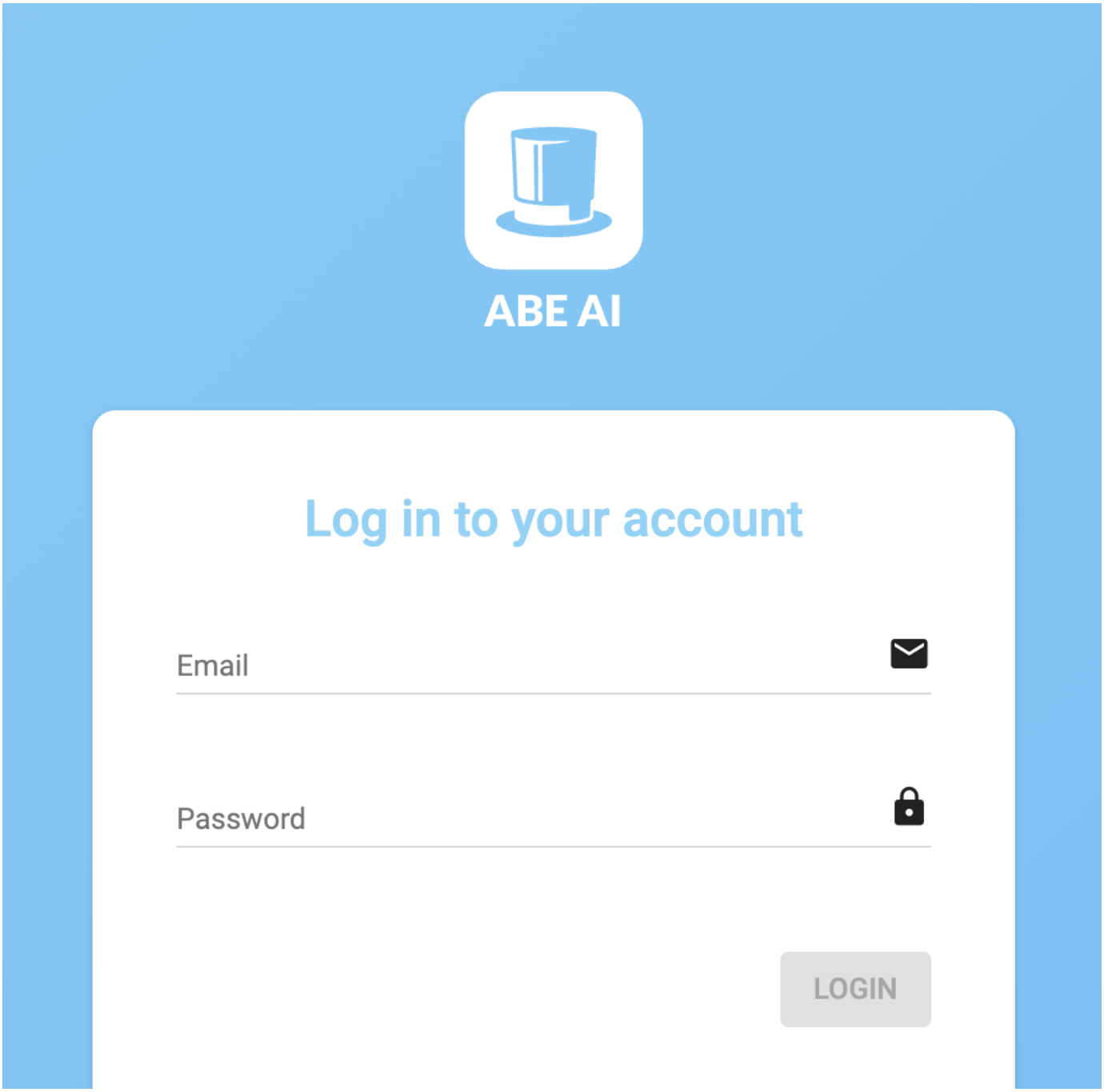
Raven Service

A singleton object that provides a single method, `launchRaven`, which will establish a connection between the browser and sentry.

UI Views

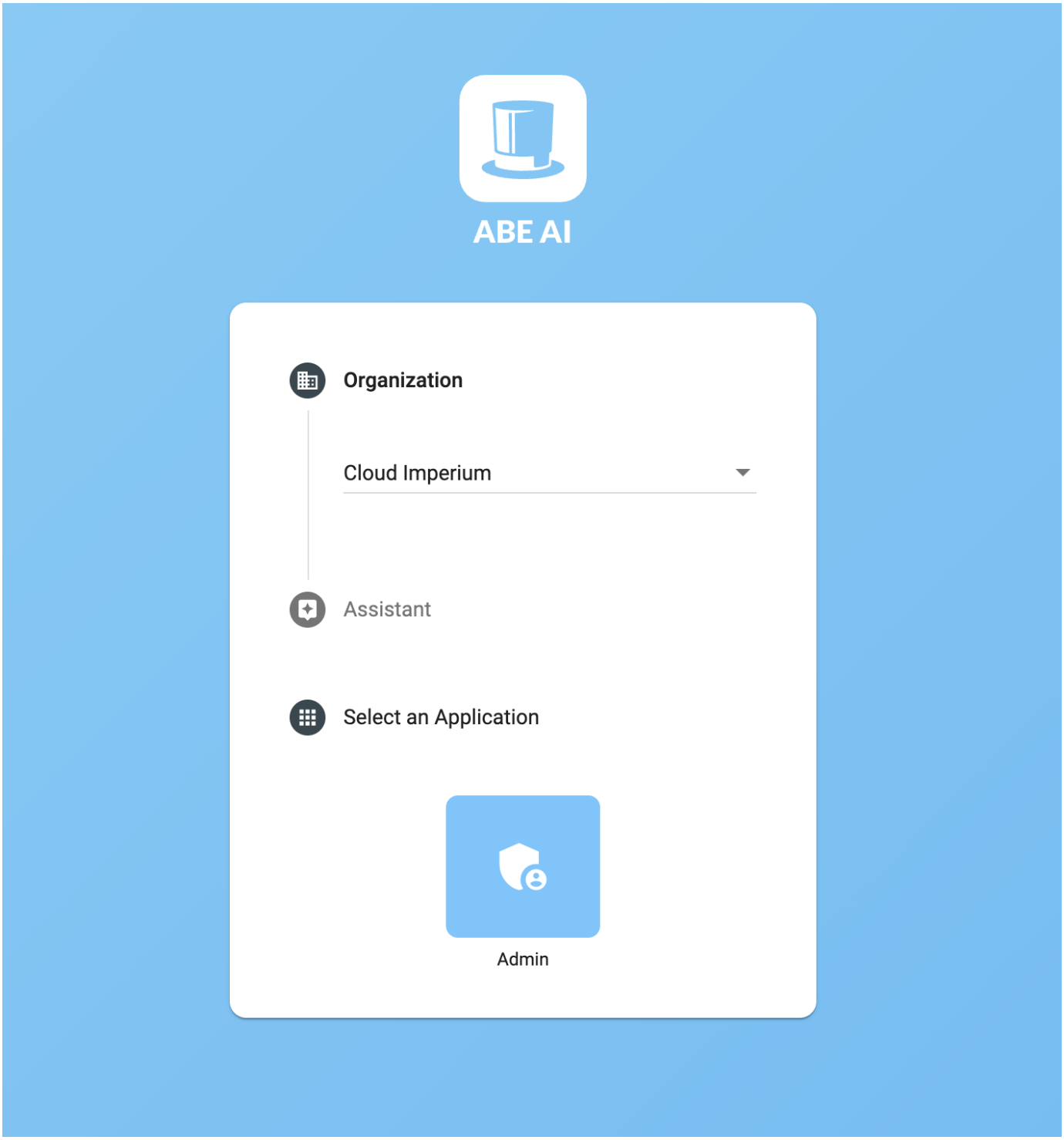
Login/Create User

These login/create pages use standard HTTPS requests to interact with Conversate API, they are the only pages to do so, as all other communication with Conversate API is done via web-socket.



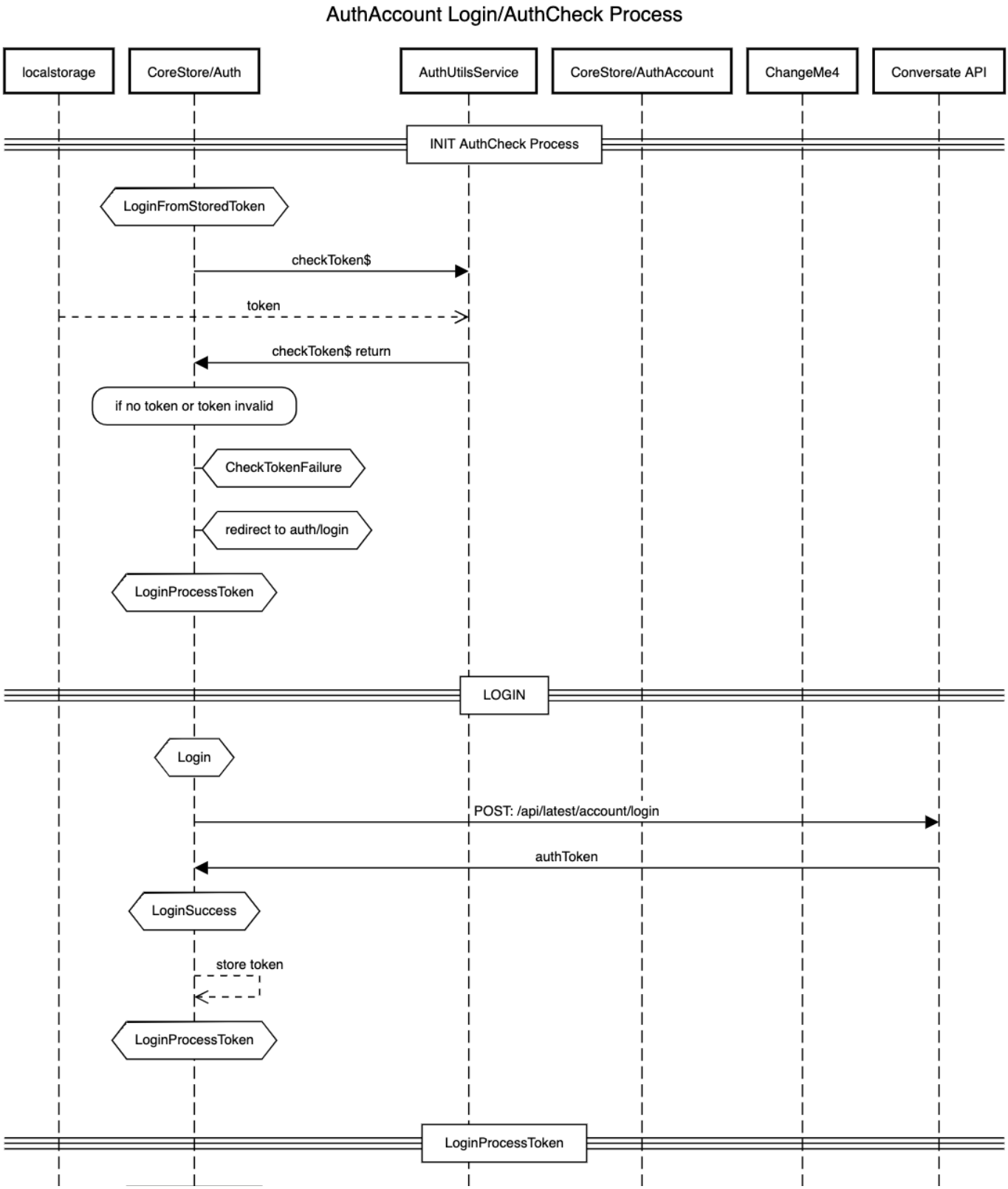
App Picker

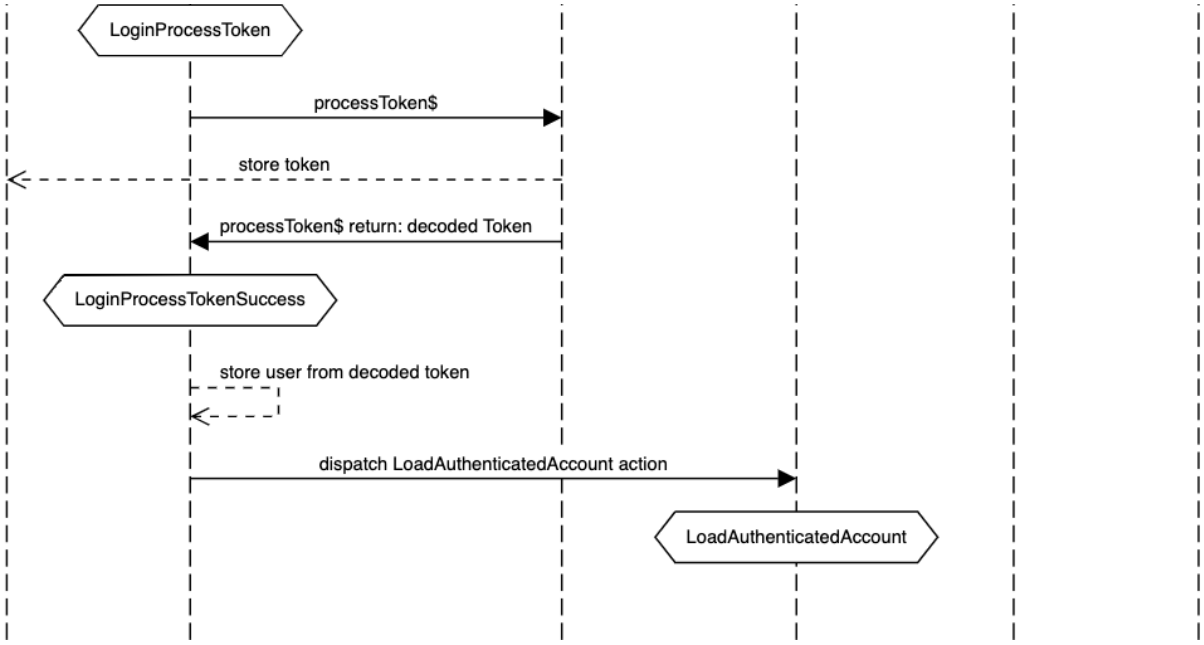
App picker is a single page, that allows users who have access to both CUI's conversation designer and Contact Center to choose which app to load into. It also allows users who have access to multiple organizations or assistants, to select the context of the active organization and assistant. Users can only directly access 1 assistant at a time, per browser tab. Users must select an organization, assistant and application based on permissions availability (Conversation Designer, Admin, Contact Center). Contact Center does not actually live in CUI, and users will be redirected to the contact center URL.



Overview

During CUI's initialization, we automatically run our AuthCheck process before we do anything else. This process check for a JWT `authToken` stored in `localstorage`, and checks to see if it is still valid. If the token is found or valid, we decode the token and store it in the coreStore/auth state via the `LoginProcessToken` action (see below). If missing or expired, we return users to the login page, and set whatever URL they were trying to access as the `preAuthUrl`. This url is used during the `redirectIfNecessary` action, which will either redirect to a preAuthUrl or the app picker at `/account/applications`.





Core Store

Overview

Core store is the main store in CUI, or the “forRoot” store. It is intended for housing information used across multiple apps.

States

Auth State

The Auth state is used to store the authToken, user information, and if there is a URL redirect needed after login (if user tried to go to url without a valid authToken, we attempt to send them there after login).

Authenticated Account State

The Authenticated Account state is used to store information about the current user and their permissions, as well as the current context for what the active Organization and assistant are. This information is referenced from many other modules.

DynEnv State

The DynEnv state is where we house the result of dynEnv loading the environment variables for CUI. The hydartion of this state is crucial for the initialization of CUI. Without it, we cannot establish our websocket connection to Conversate nor our connection to our logging service Sentry.

Auth State

Overview

The Auth state is used to store the authToken, user information, and if there is a URL redirect needed after login (if user tried to go to url without a valid authToken, we attempt to send them there after login).

State Structure

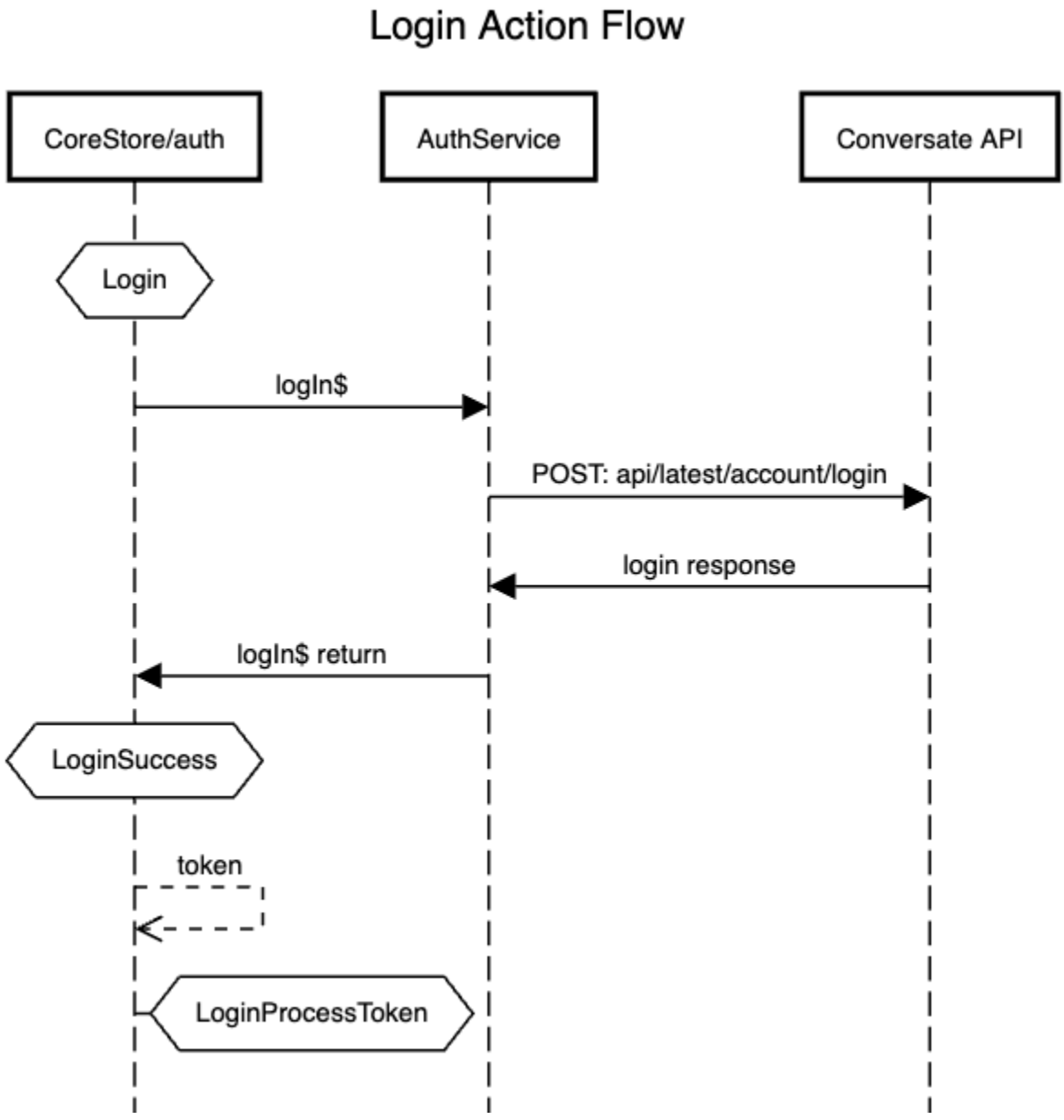
```
{
  user: {
    id: UUID;
    account: UUID;
    namespace: string;
    email: string;
    exp: number;
```

```
    hasLoggedOut: boolean;
    isLockedOut: boolean;
    error: RESTError | null;
  } | null;
  token: string | null;
  loaded: boolean;
  loading: boolean;
  validating: boolean;
  authenticated: boolean;
  preAuthUrl: string | null;
  createAccountResults: CreateAccountResults | RESTError | null;
}
```

Action Sequence Flows

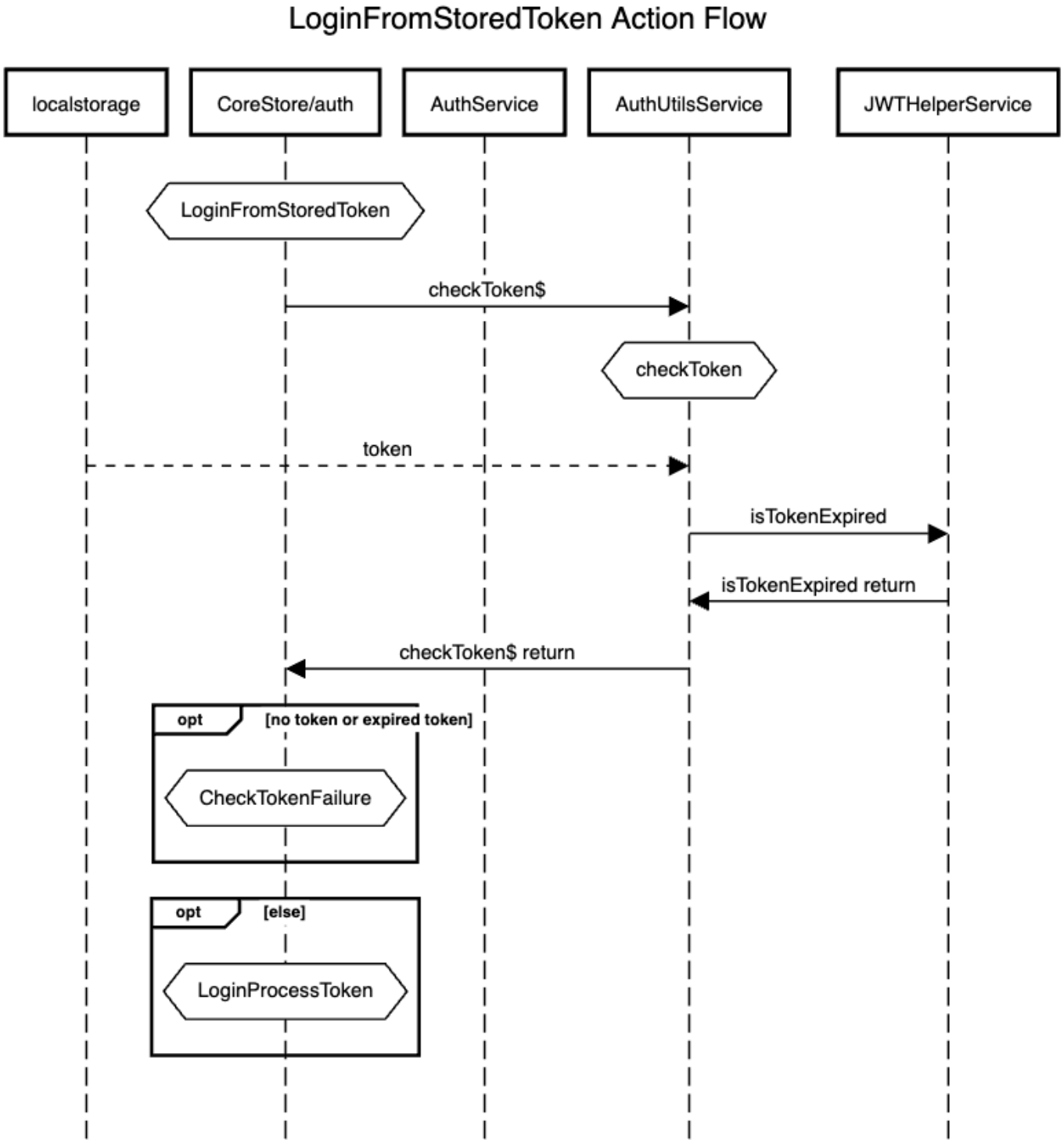
Login

Login from login form. Unlike the rest of CUI communication to Conversate, this is done over HTTPS instead of WS



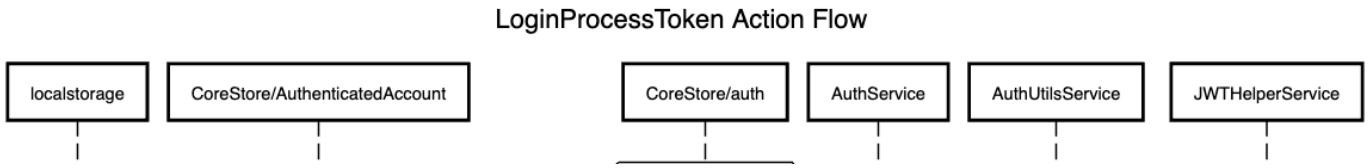
LoginFromStoredToken

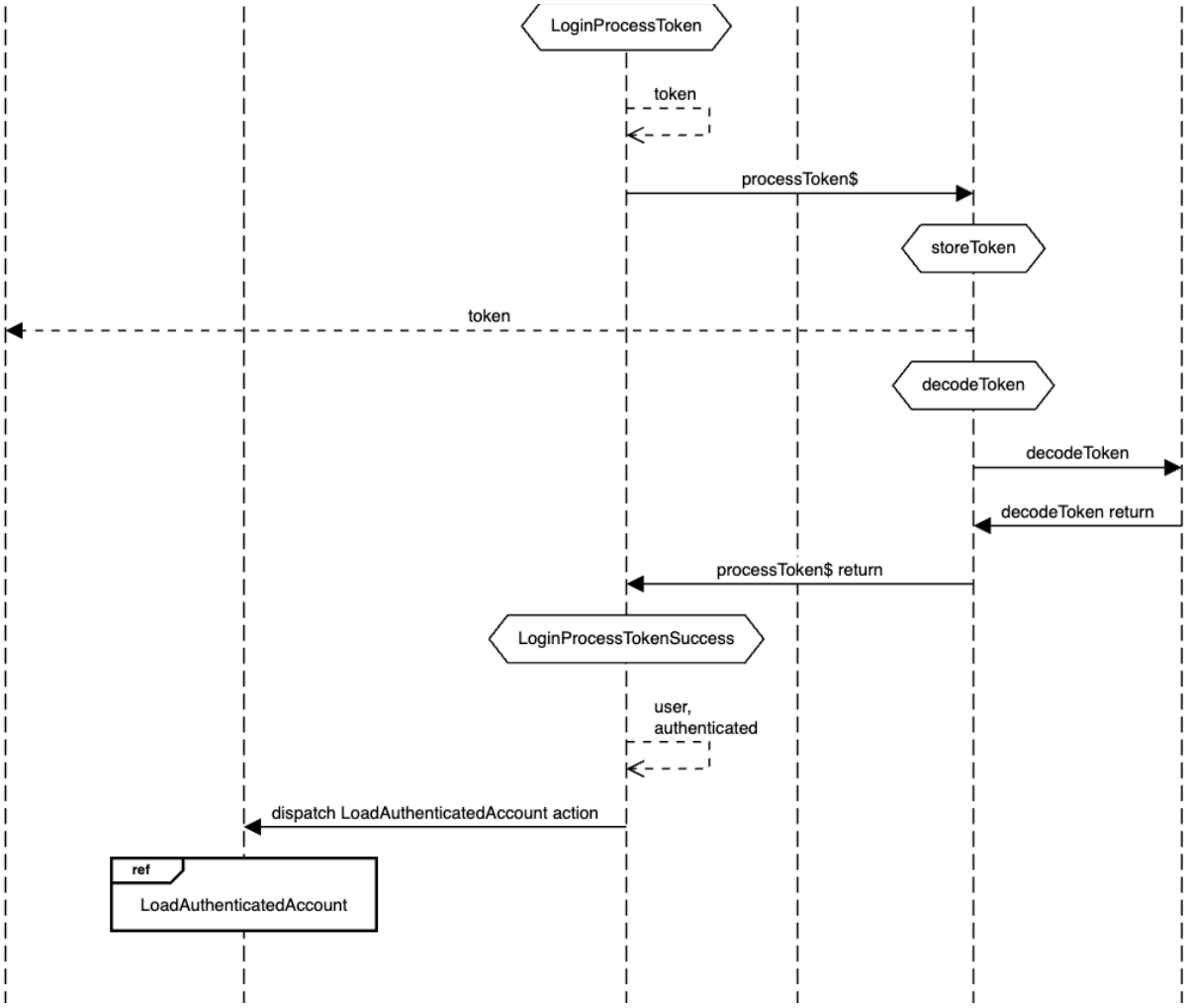
This action is executed upon application init. Checks for a auth token in localstorage and checks its expiry.



LoginProcessToken

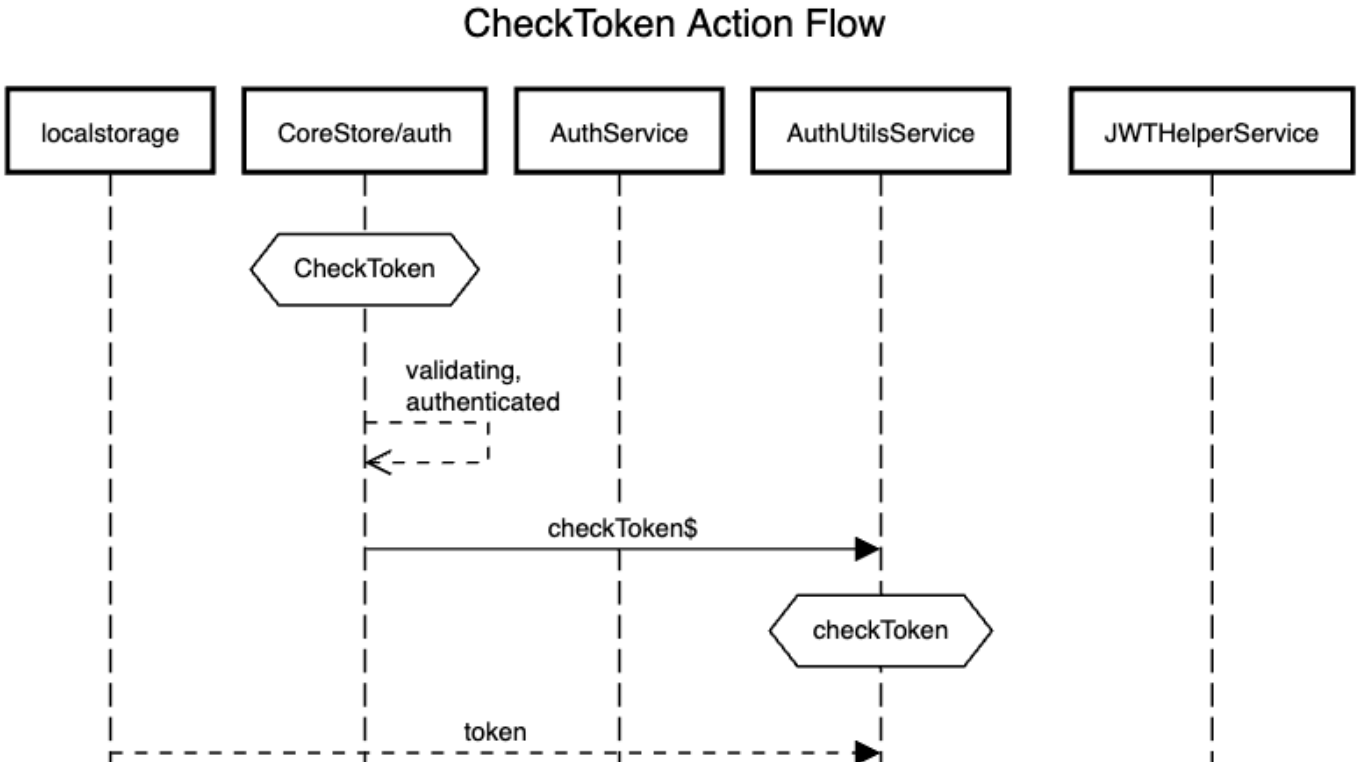
Process a jwt token after login or after login from stored token

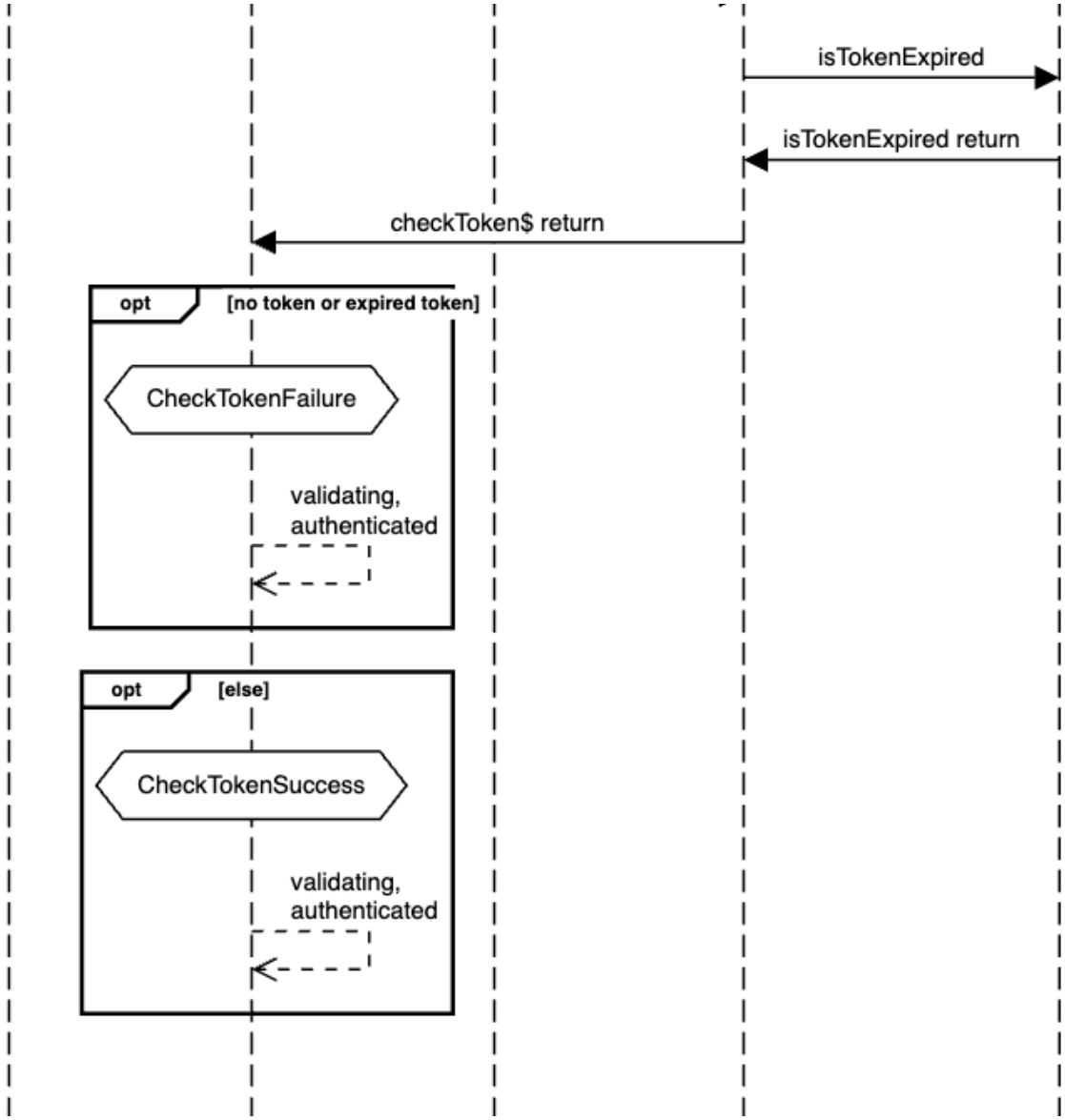




CheckToken

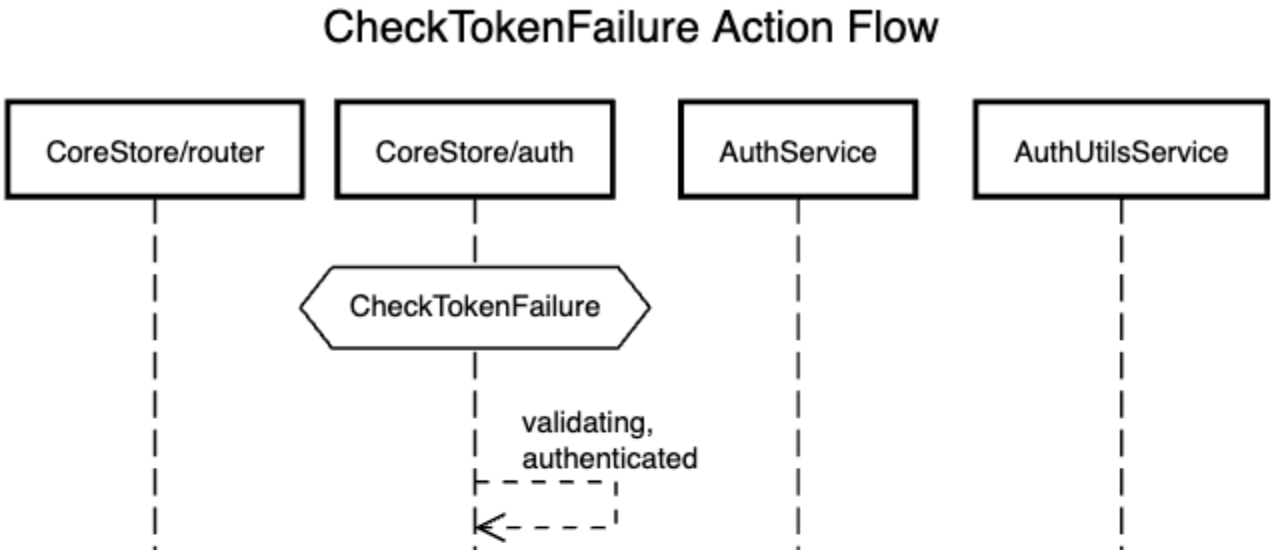
checks auth token exist, and expiry

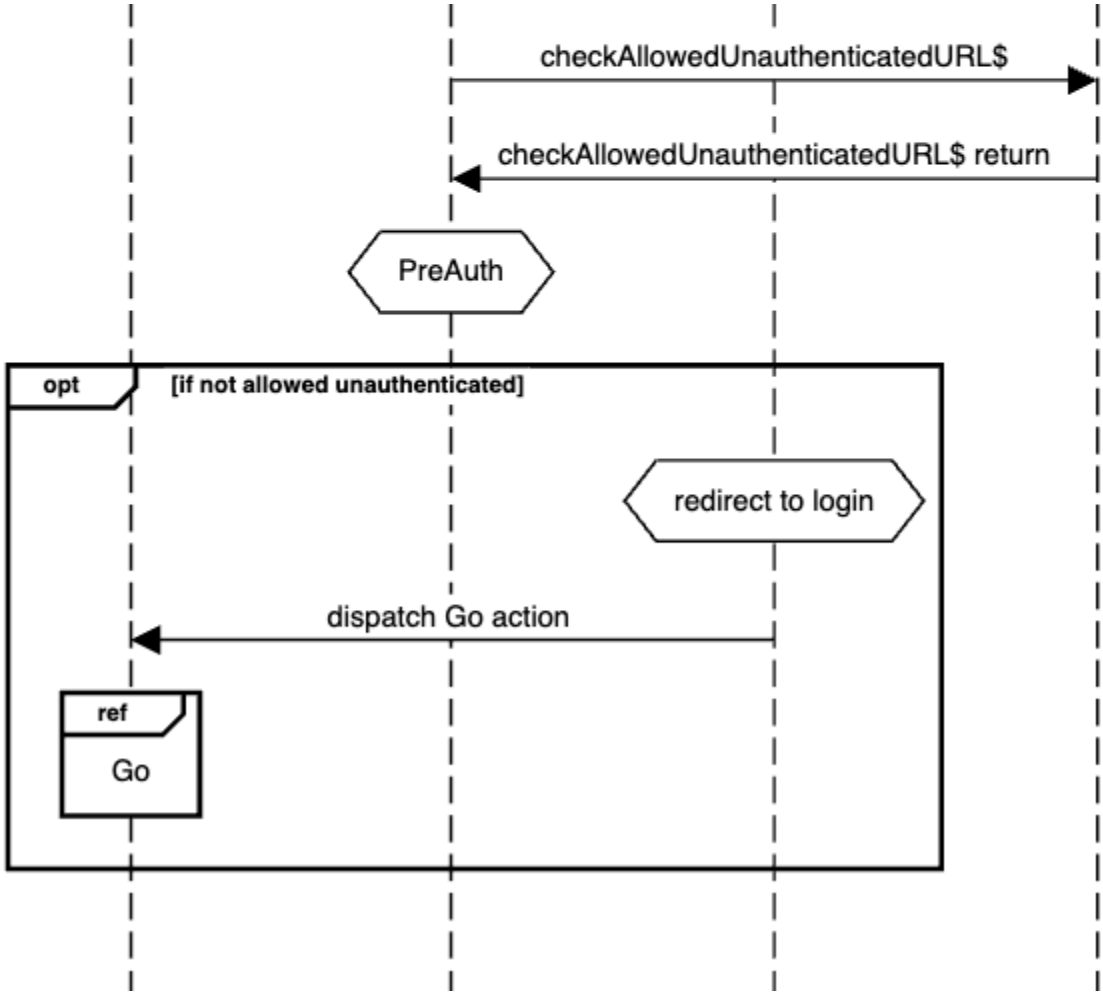




CheckTokenFailure

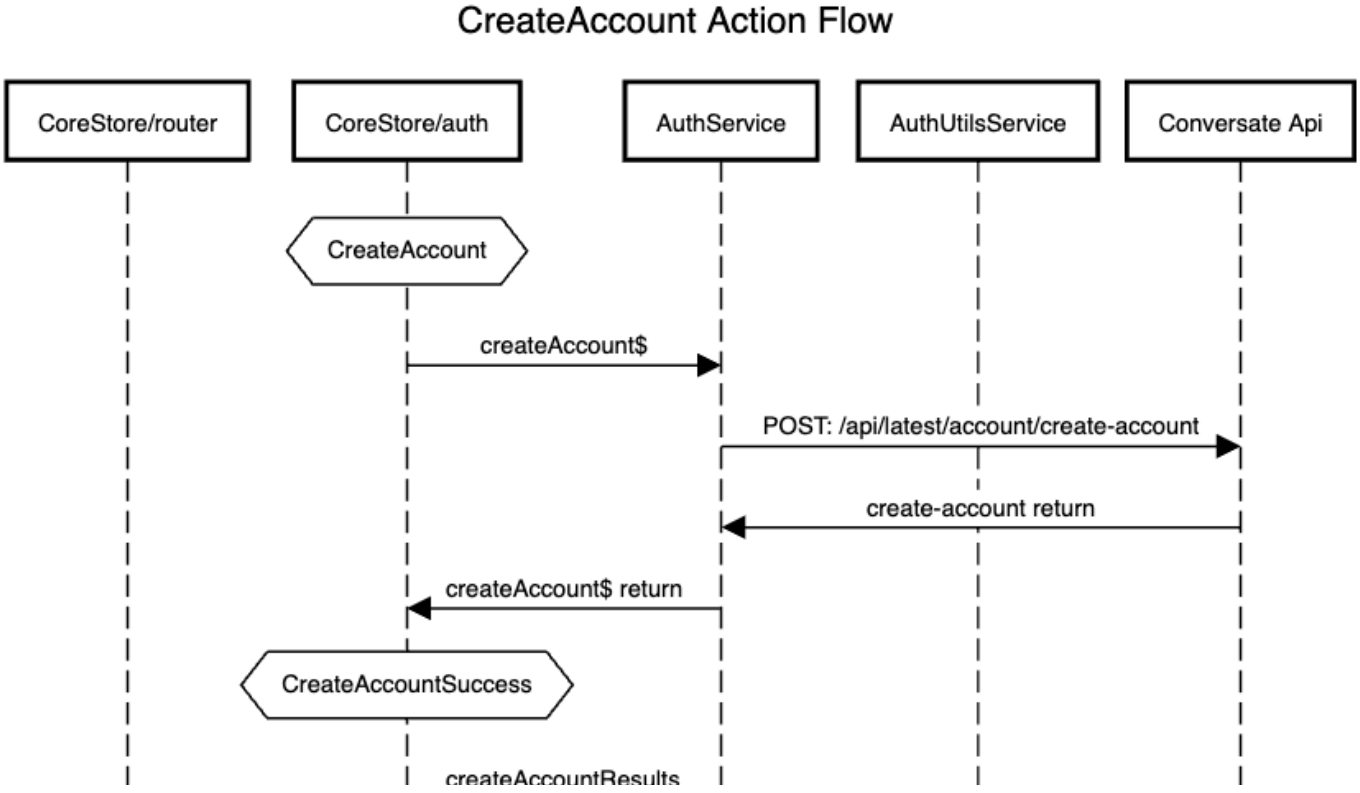
upon a failed token check, we need to redirect the user somewhere. If the current url is accessible unauthenticated we set the preauth to the window location pathname. If the path is not allowed unauthenticated, then we set the preauth url and redirect to the login page. `/account/create` is the only allowed unauthenticated route.

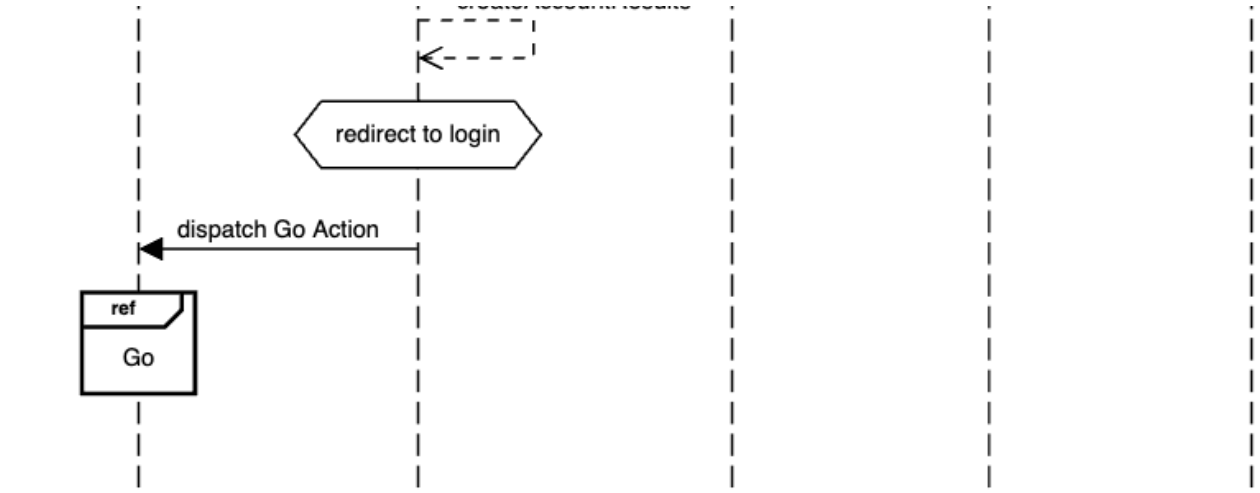




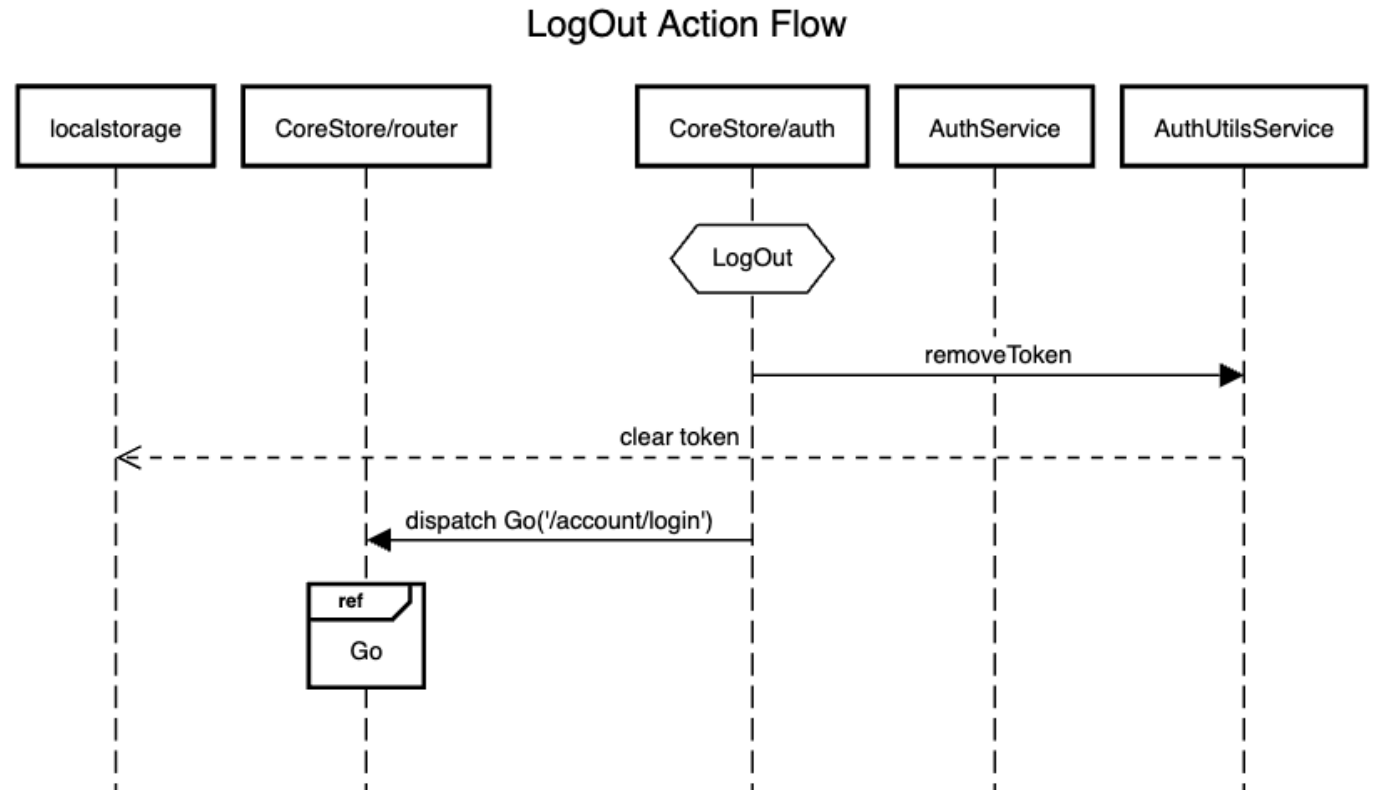
CreateAccount

Creates a new organization and user (if not existing)



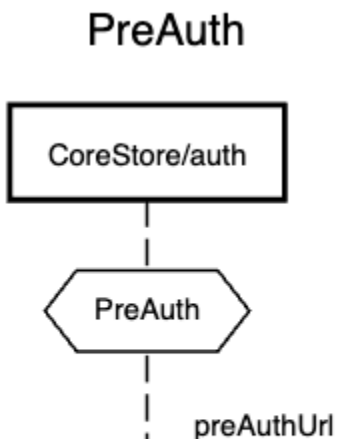


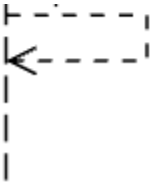
LogOut



PreAuth

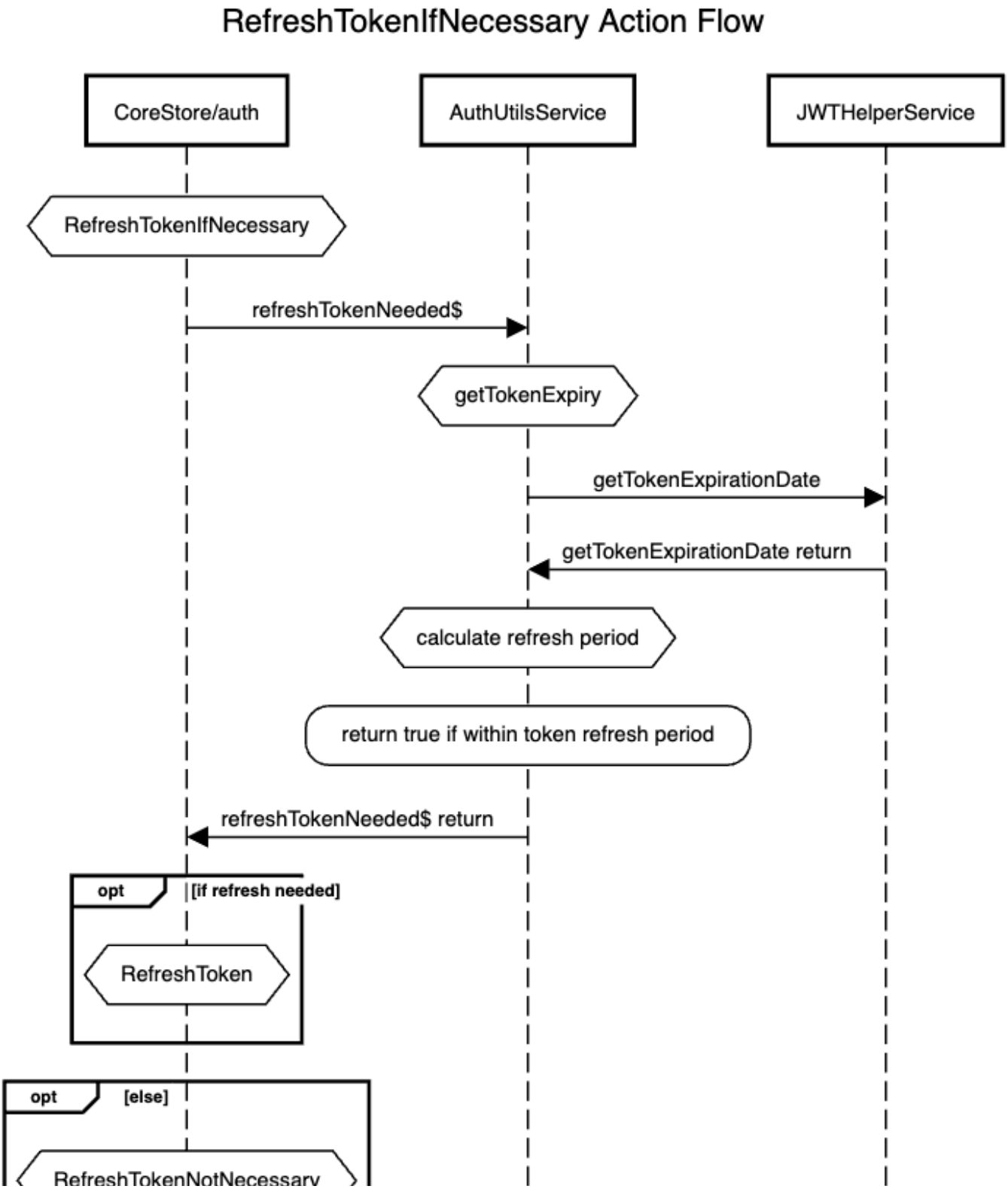
stores the current `window.location.pathname`





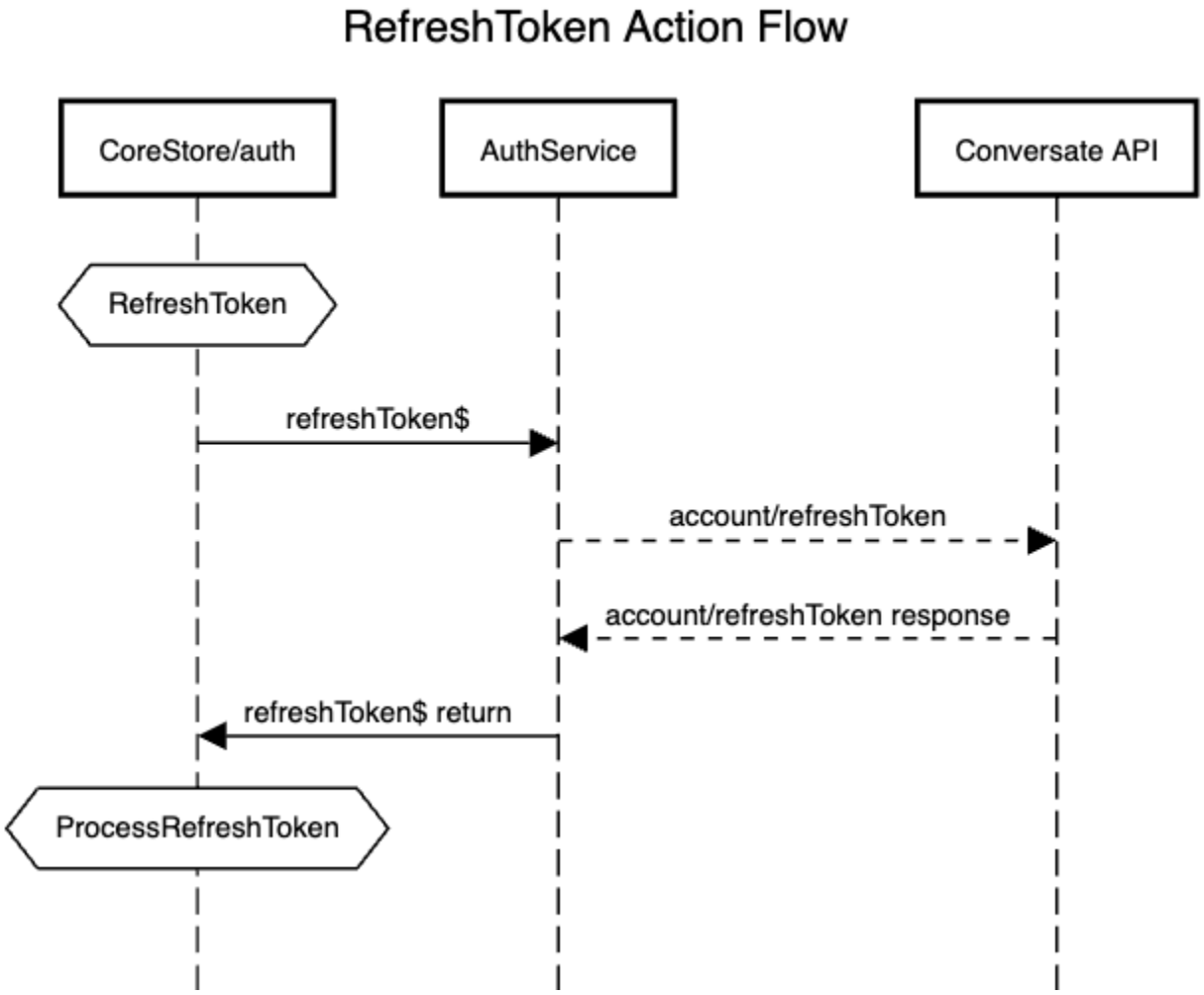
RefreshTokenIfNecessary

Determines if the current time is within 10 minutes of token expiry.

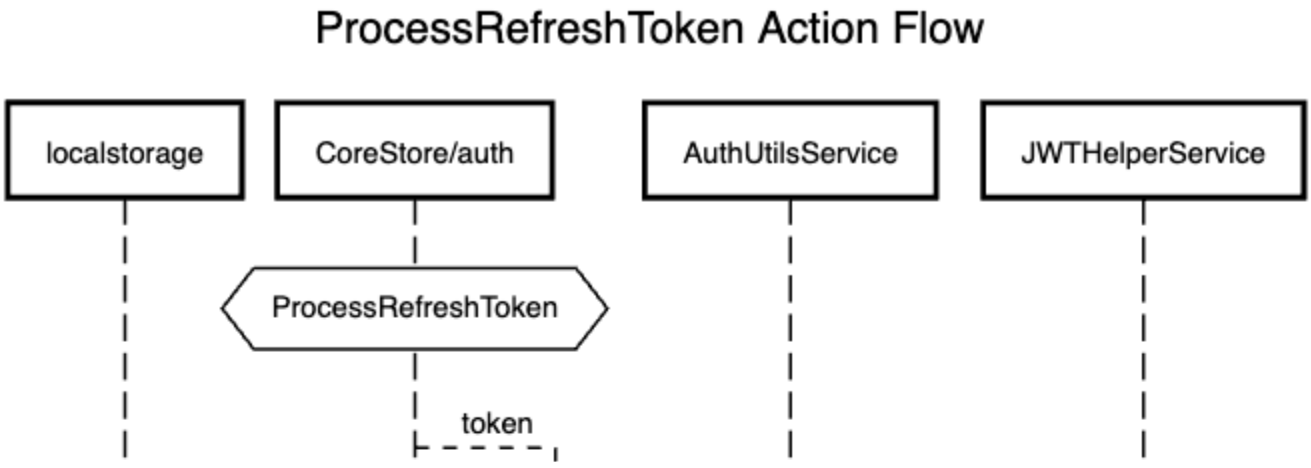


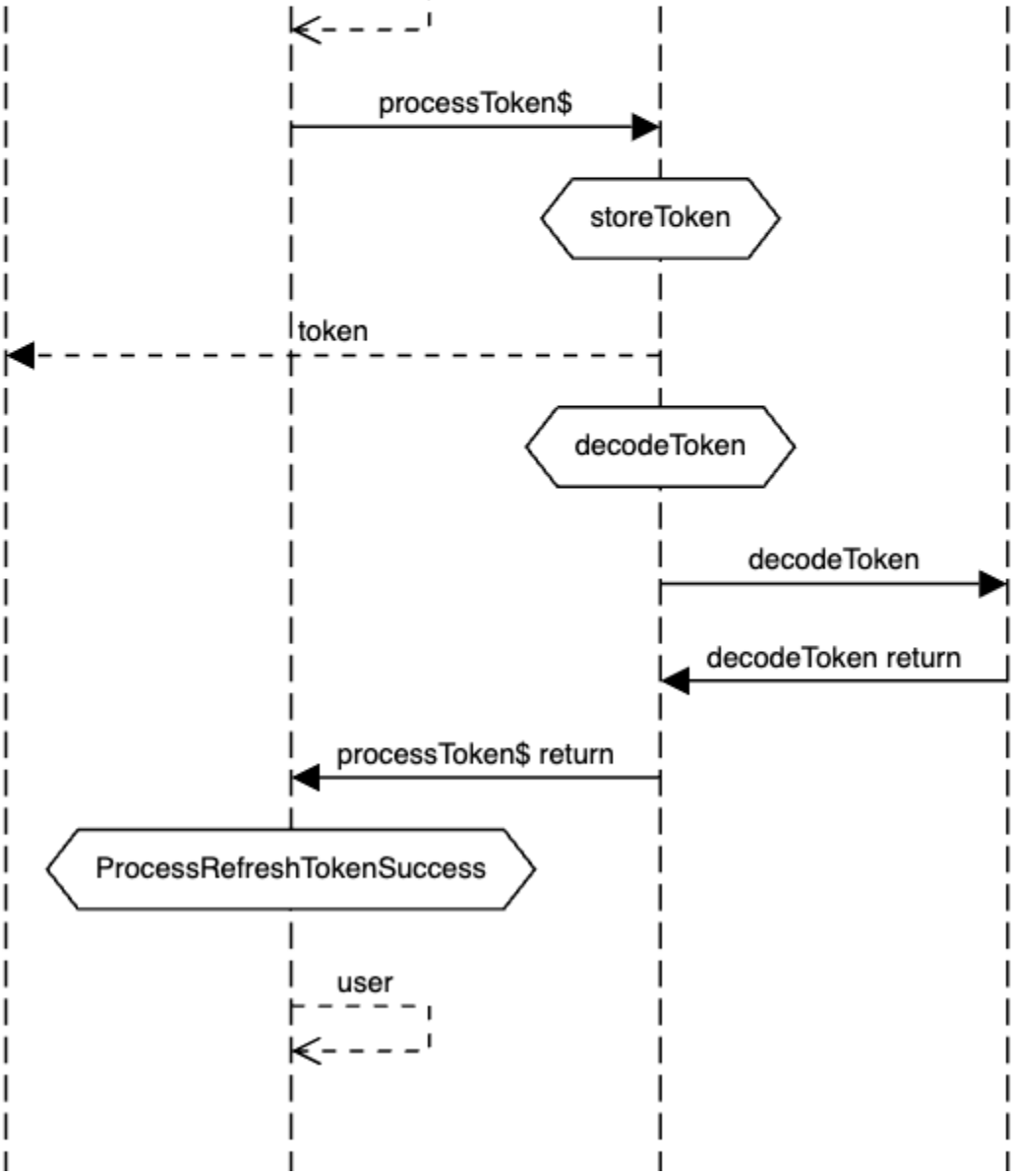
RefreshTokenNotNecessary

This action does not perform any mutations to the state, or trigger any additional effects. It exists simply to log in ngrx history that we did not refresh the token
RefreshToken



ProcessRefreshToken





Authenticated Account State

Table of Contents

- UpdateAgentTestUser
- UpdateAgentGlobalVariables
- PropmtForUserInfoIfNecessary
- PromptForPasswordChangeIfNecessary
- RedirectIfNecessary

Overview

The Authenticated Account state is used to store information about the current user and their permissions, as well as the current context for what the active Organization and assistant are. This information is referenced from many other modules.

State Structure

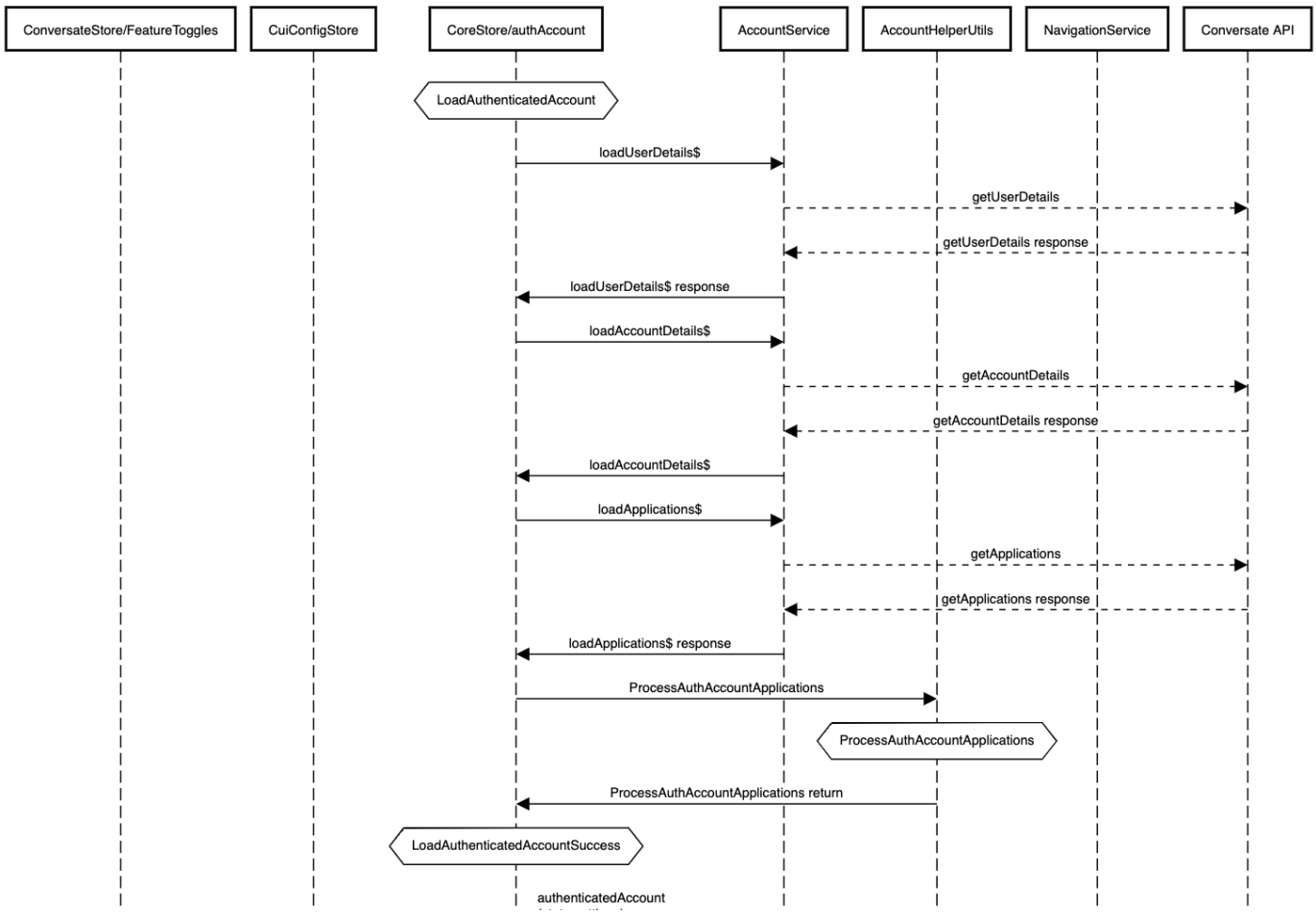
```
{
  settings: AuthenticatedAccount | null;
  createdUser: CreatedUser | null;
  loaded: boolean;
  loading: boolean;
}
```

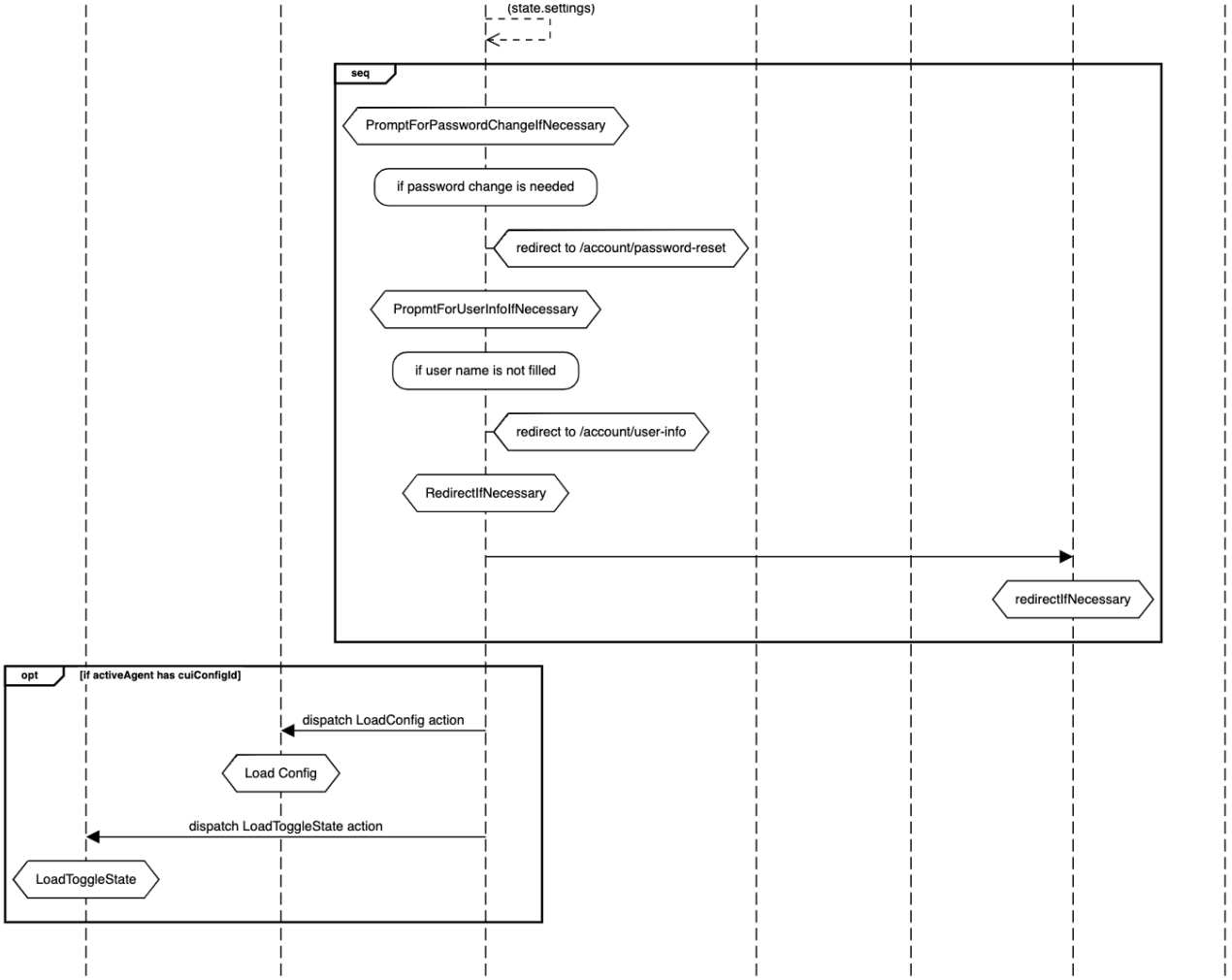
Action Sequence Flows

LoadAuthenticatedAccount

Core store, being the central store, is responsible for handling base level information gathering and storage. A very common flow in CUI is needing to be sure that the authenticatedAccount state is up to date and hydrated prior to attempting to load a cui config for a specific assistant or even to know what the current logged in user is authorized to access. This common flow is a sequential process where information is gathered piece-mail, and the store, with its `effects` handle calling additional services, or dispatching actions to other stores when needed.

LoadAuthenticatedAccount Flow

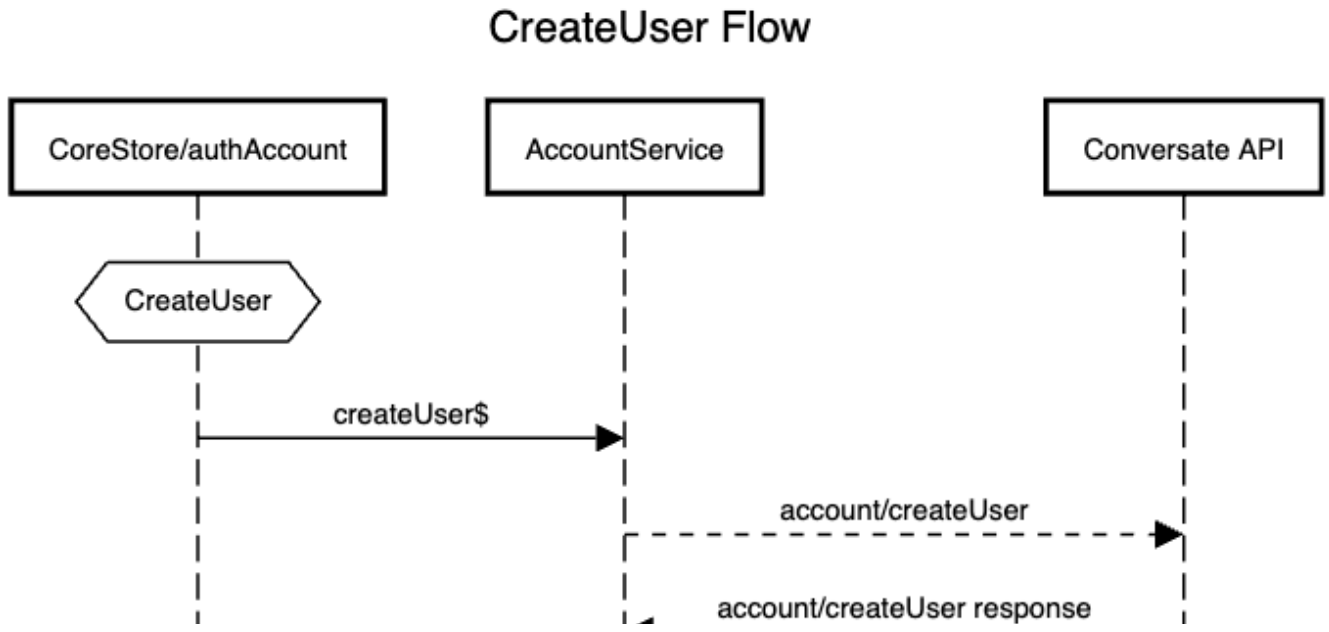


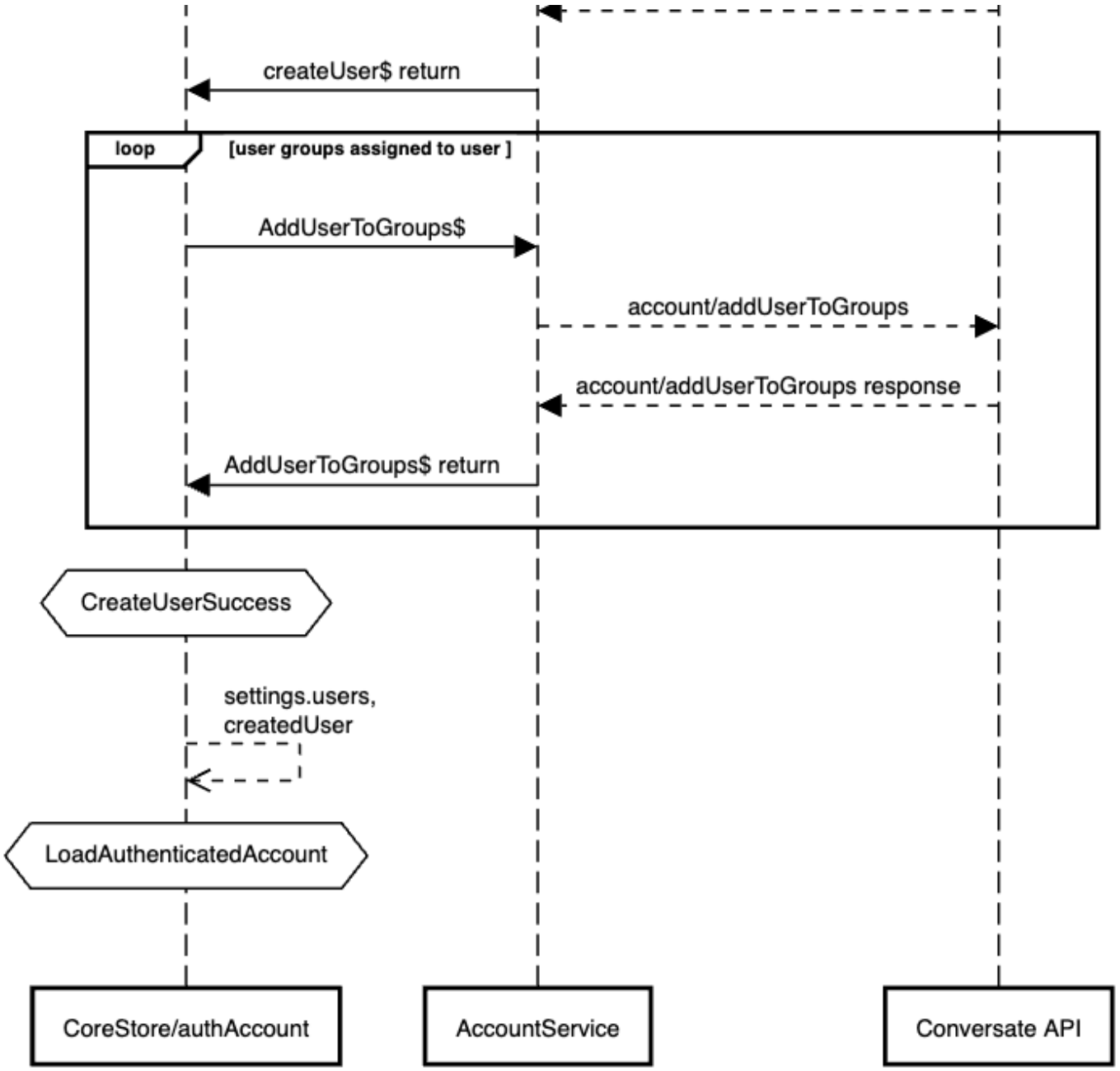


-
- LoadTogglesState

CreateUser

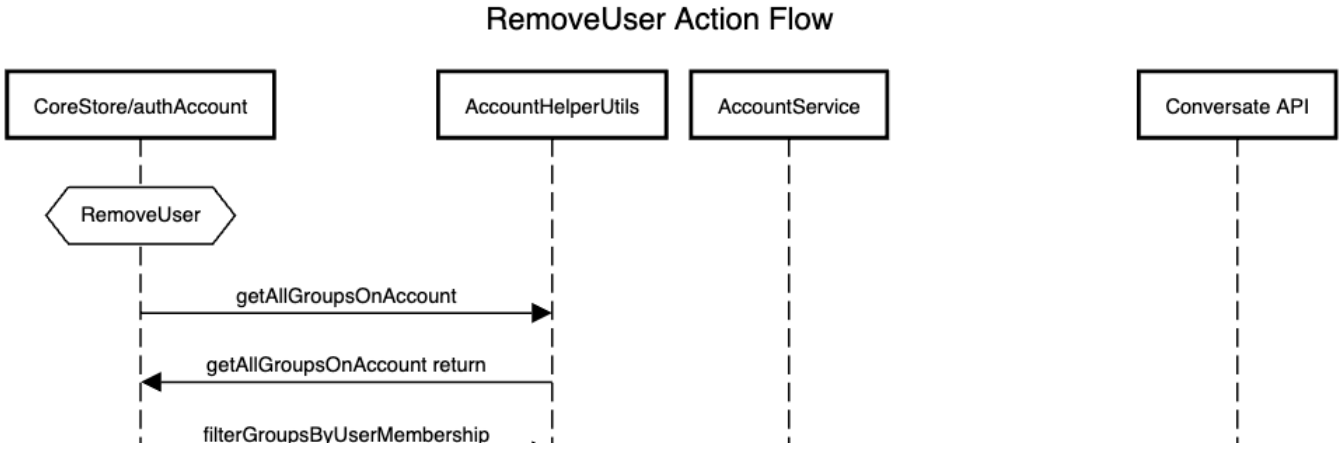
Create or assign existing user to have access to the active organization.

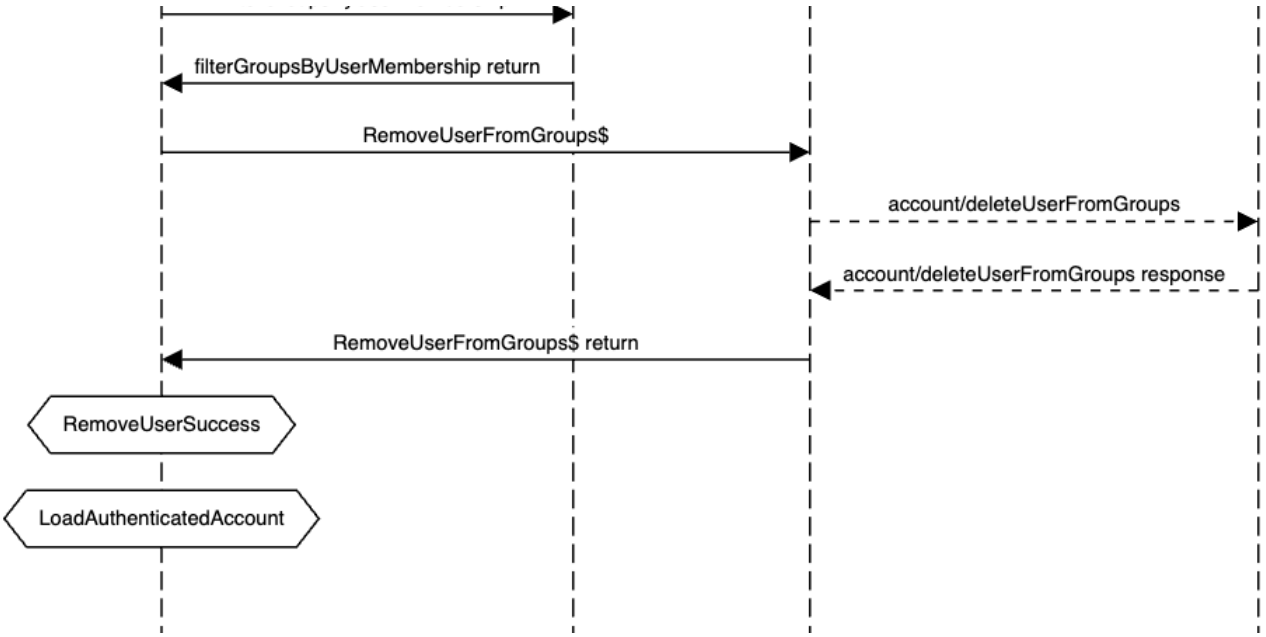




RemoveUser

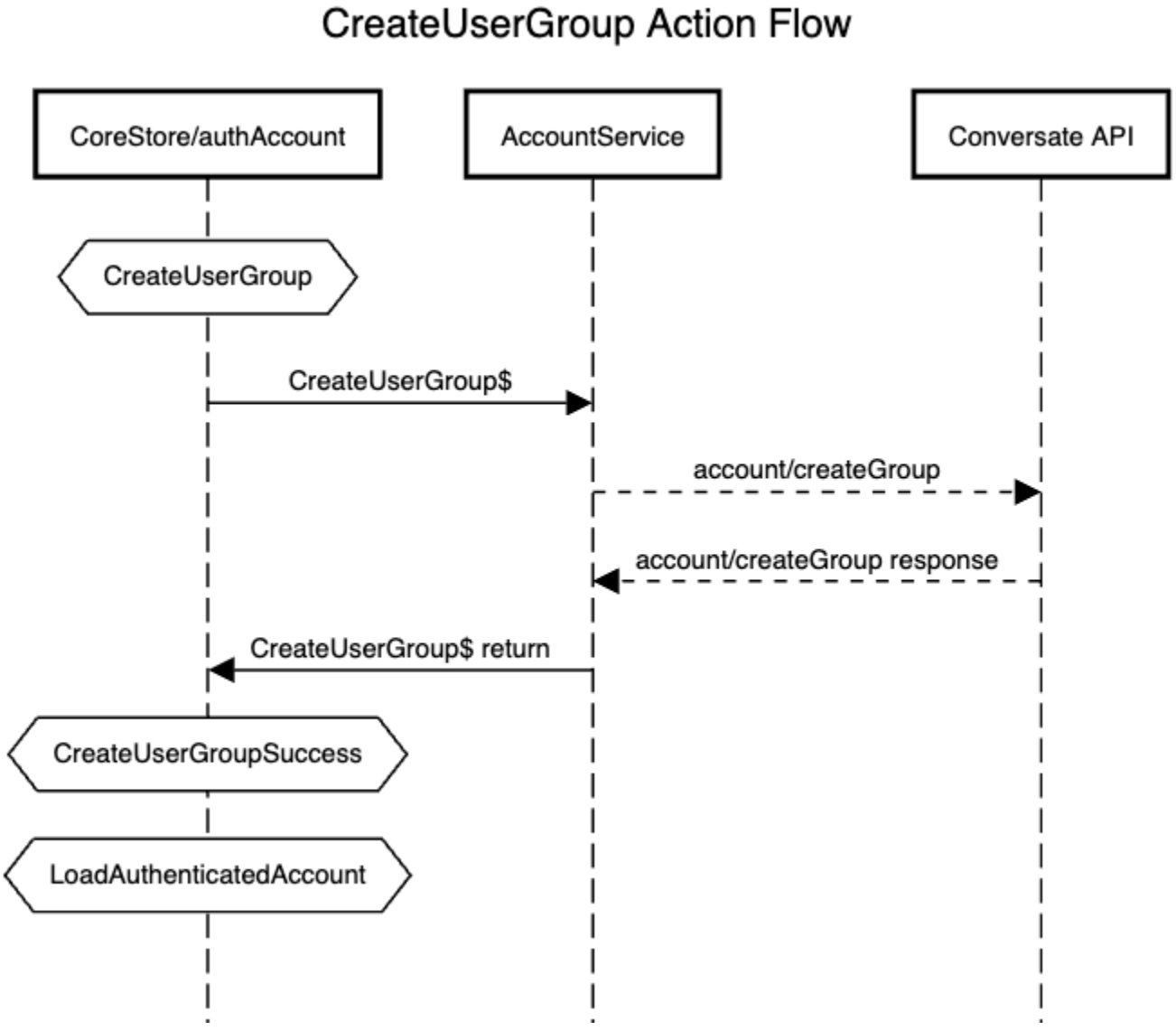
Remove a user’s access to entire organization. This does not delete the user from our DB just removes them from the organization’s user groups.





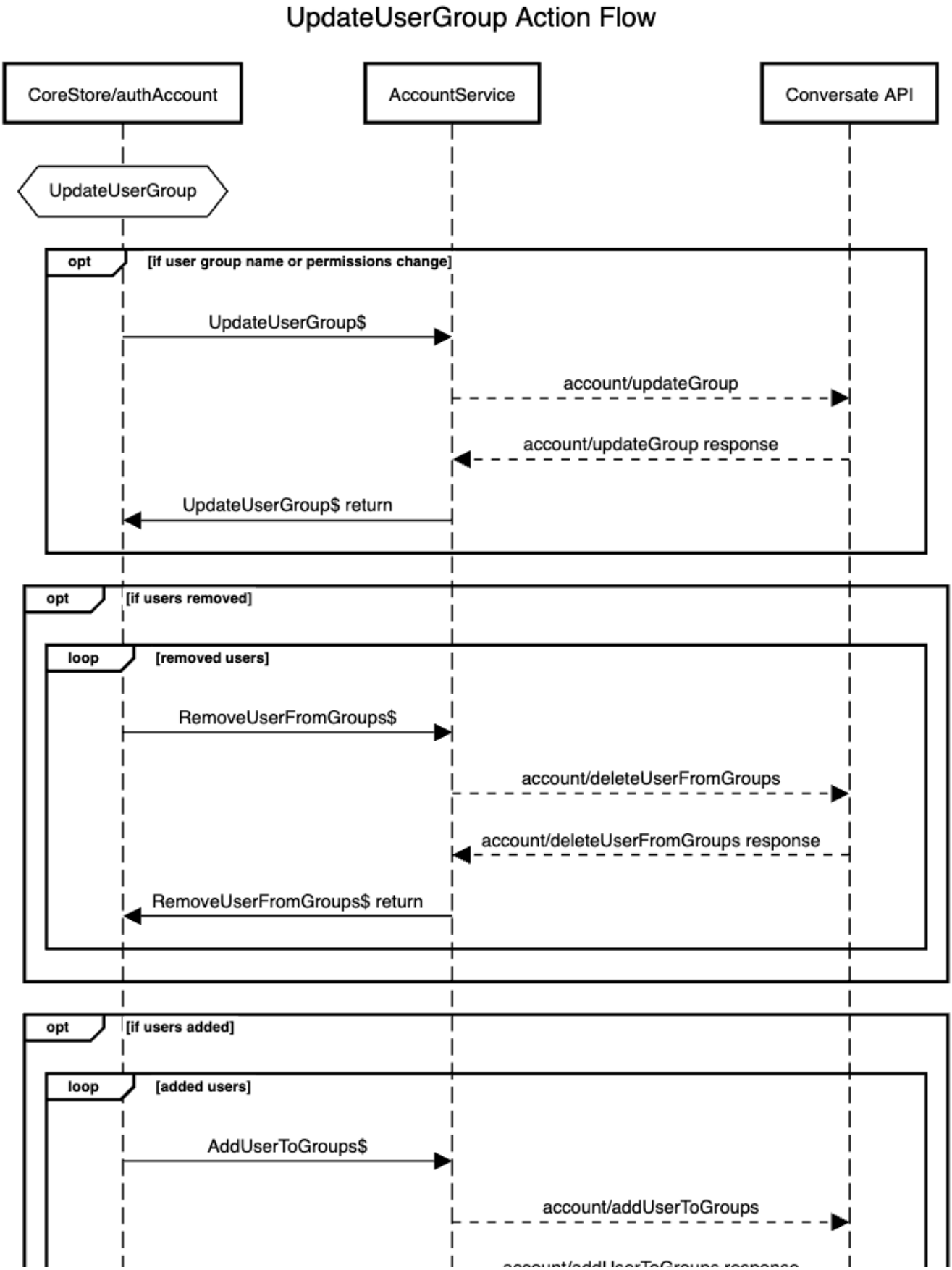
CreateUserGroup

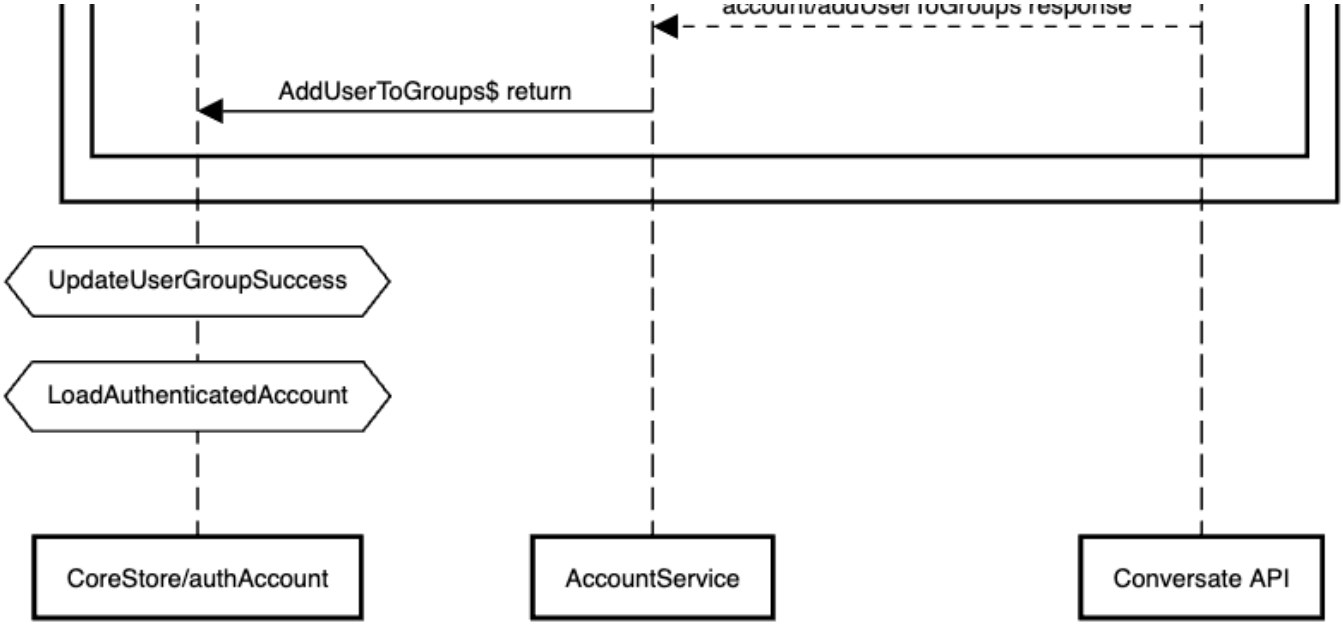
Creates a user group on provided user group owner (Organization, Assistant, Application).



UpdateUserGroup

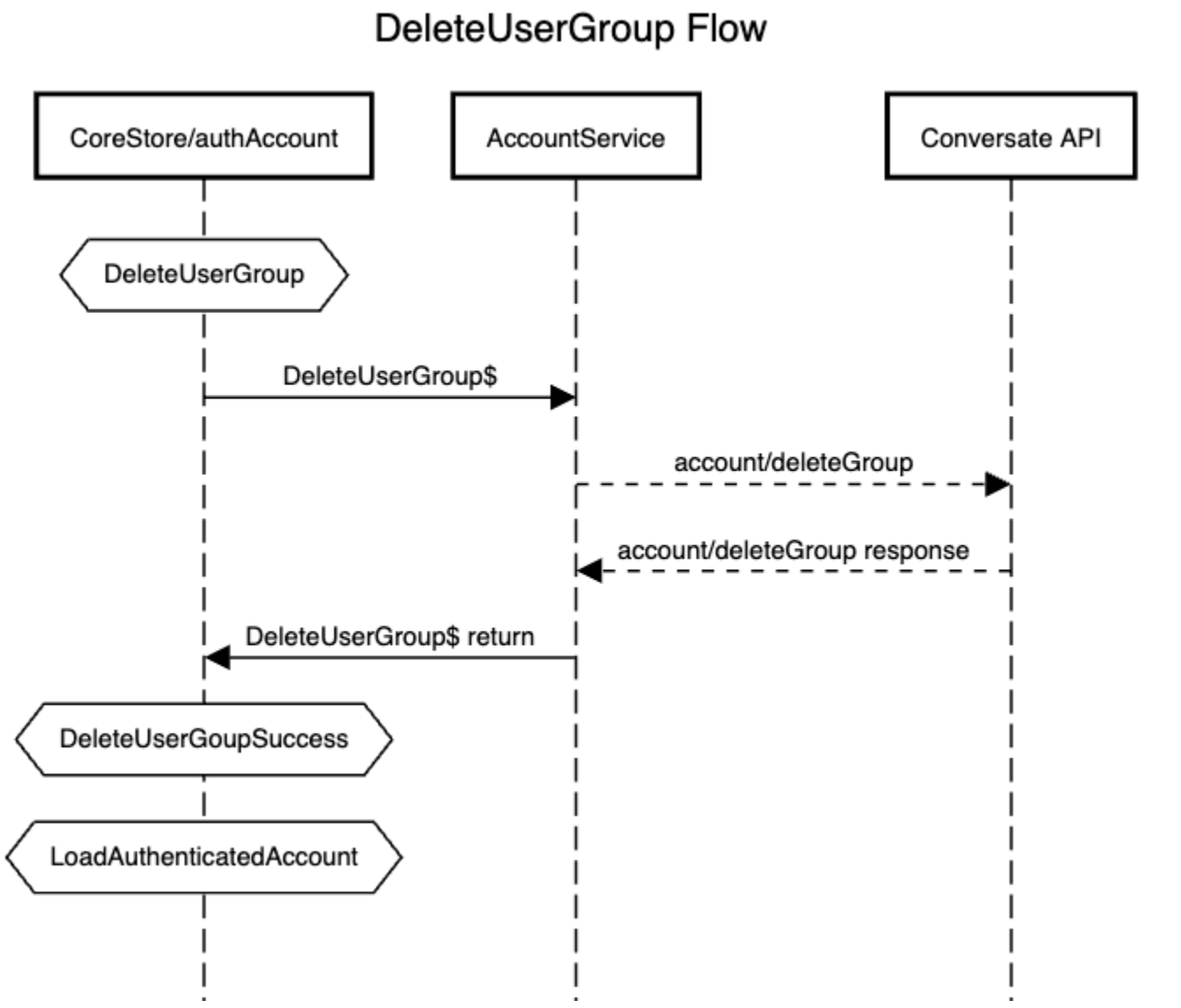
updates a user group. Changes to a user group can be either information on the user group itself, such as name or permissions list, or they can be users that need to be added/removed to the group.





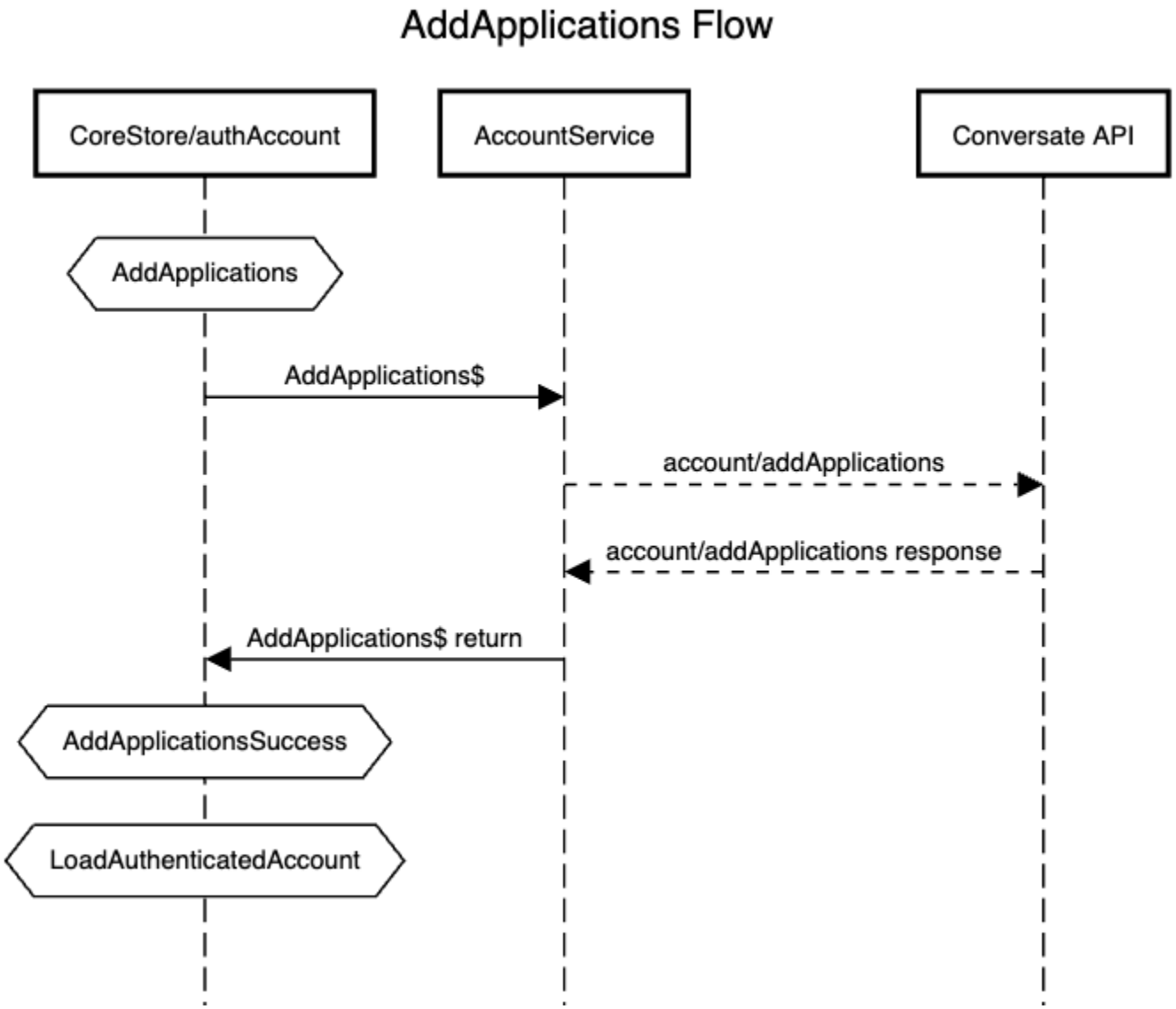
DeleteUserGroup

removes a user from the user group



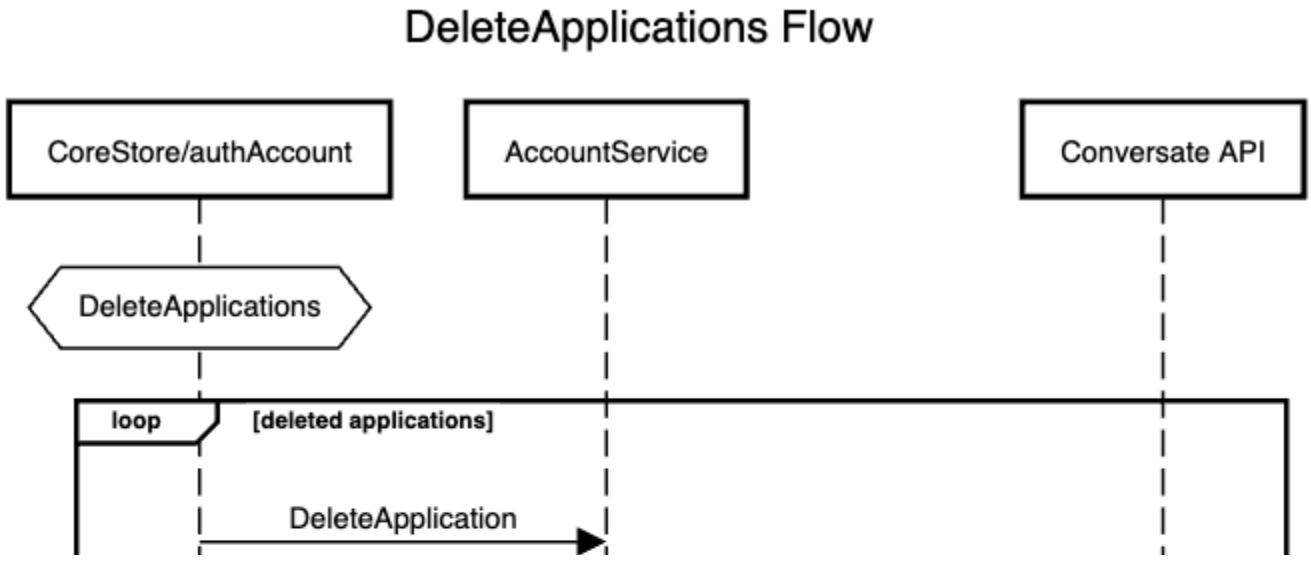
AddApplications

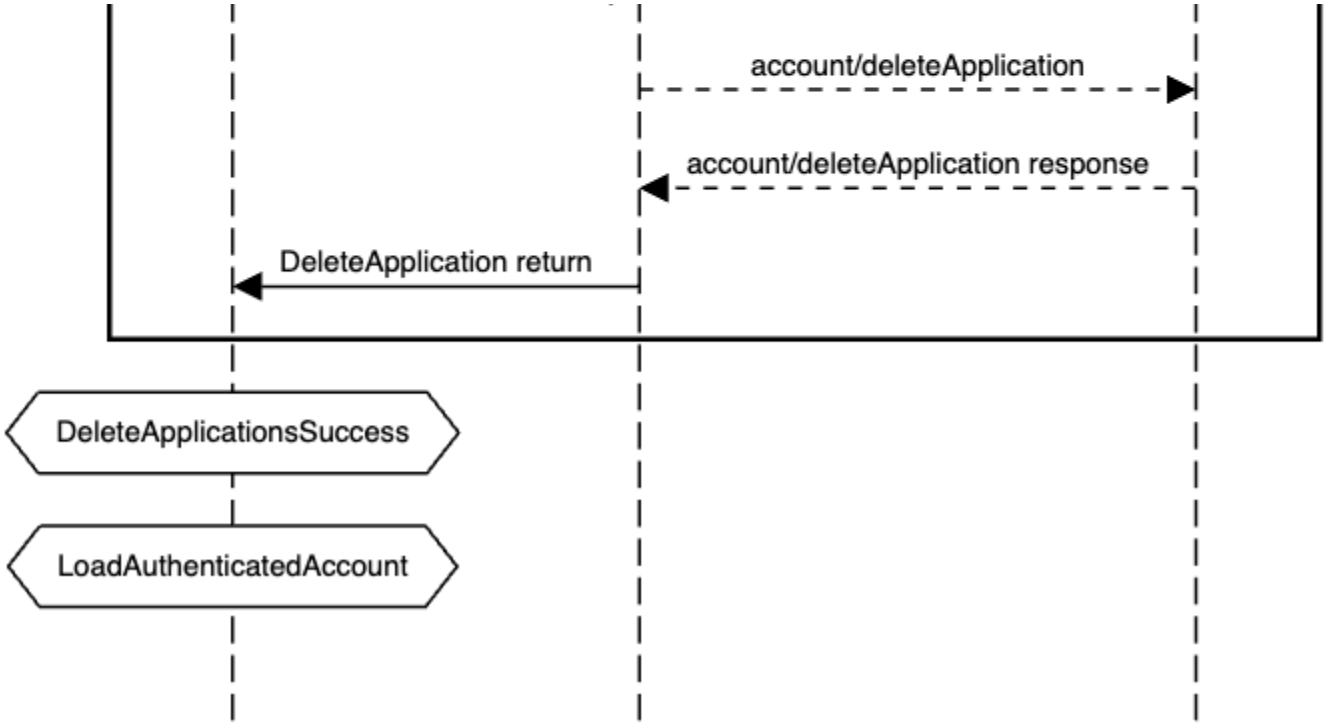
adds applications to an existing assistant



DeleteApplications

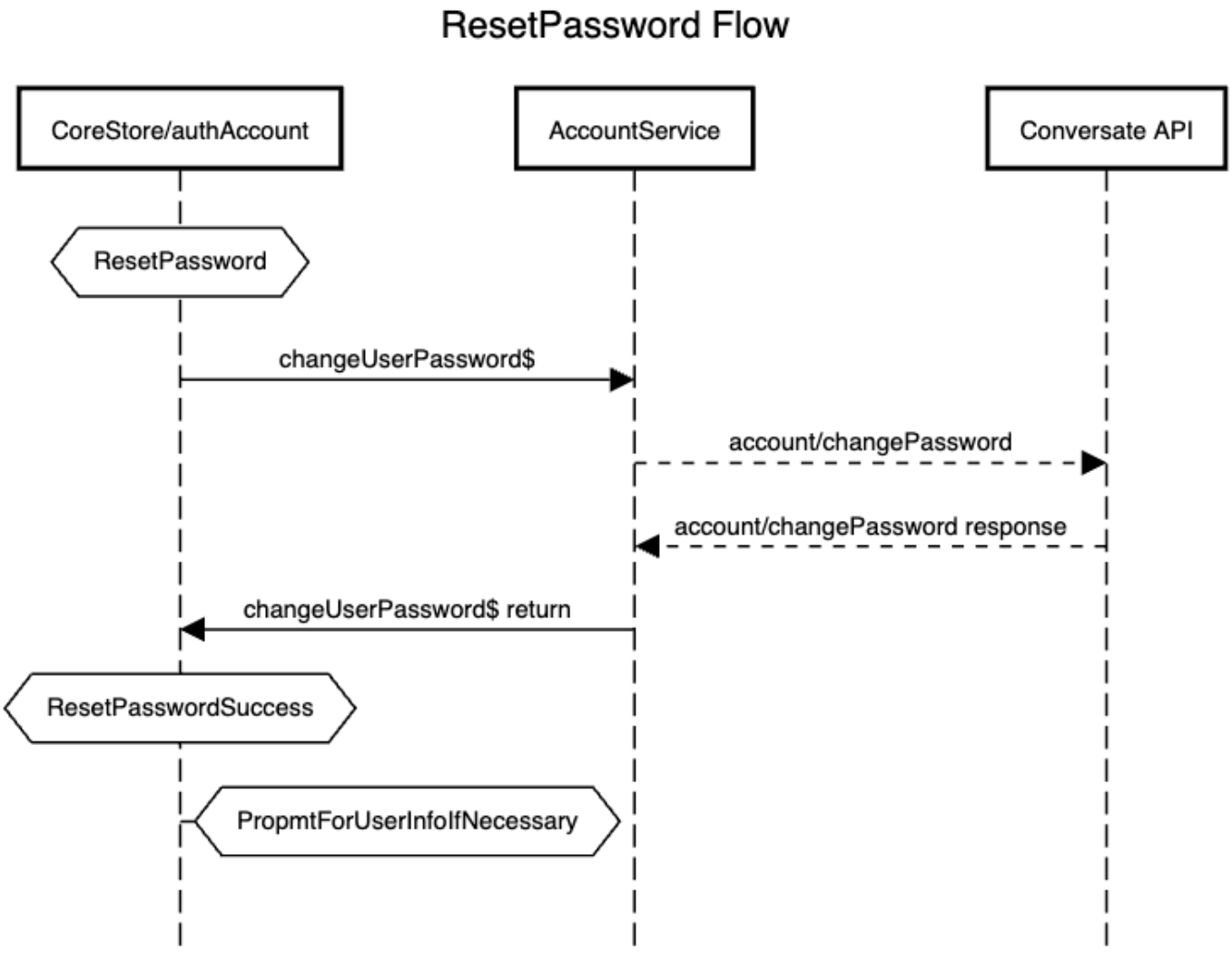
Remove applications from an assistant





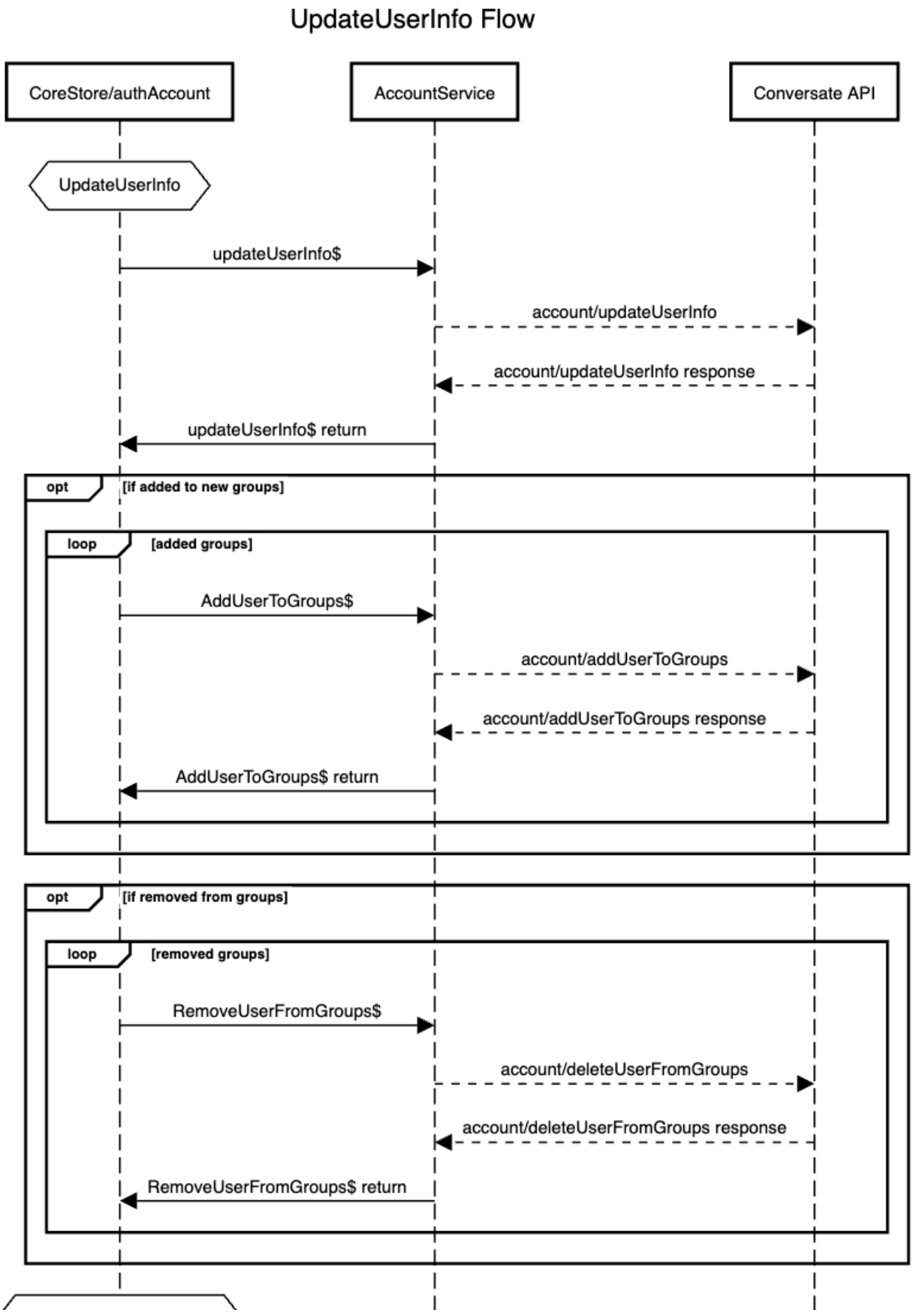
ResetPassword

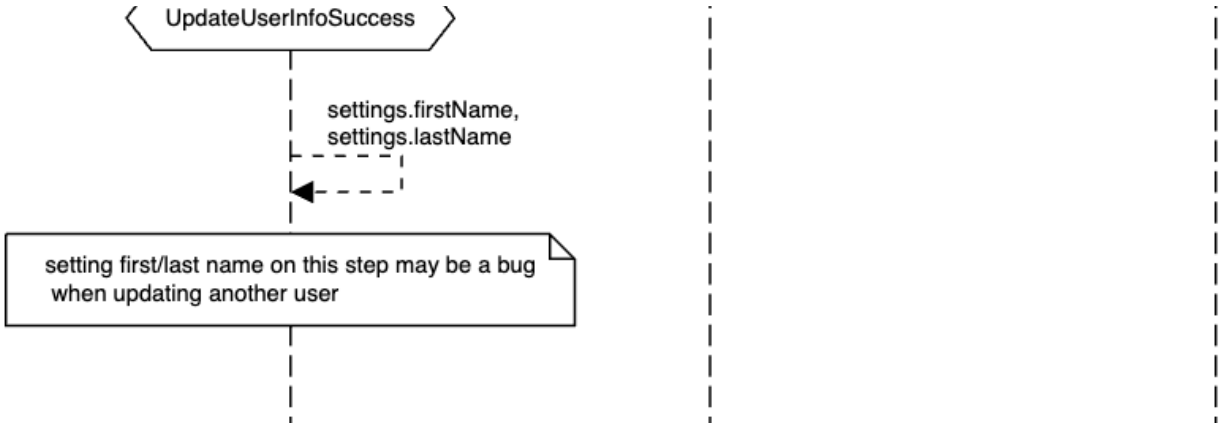
reset the current logged in user’s password



UpdateUserInfo

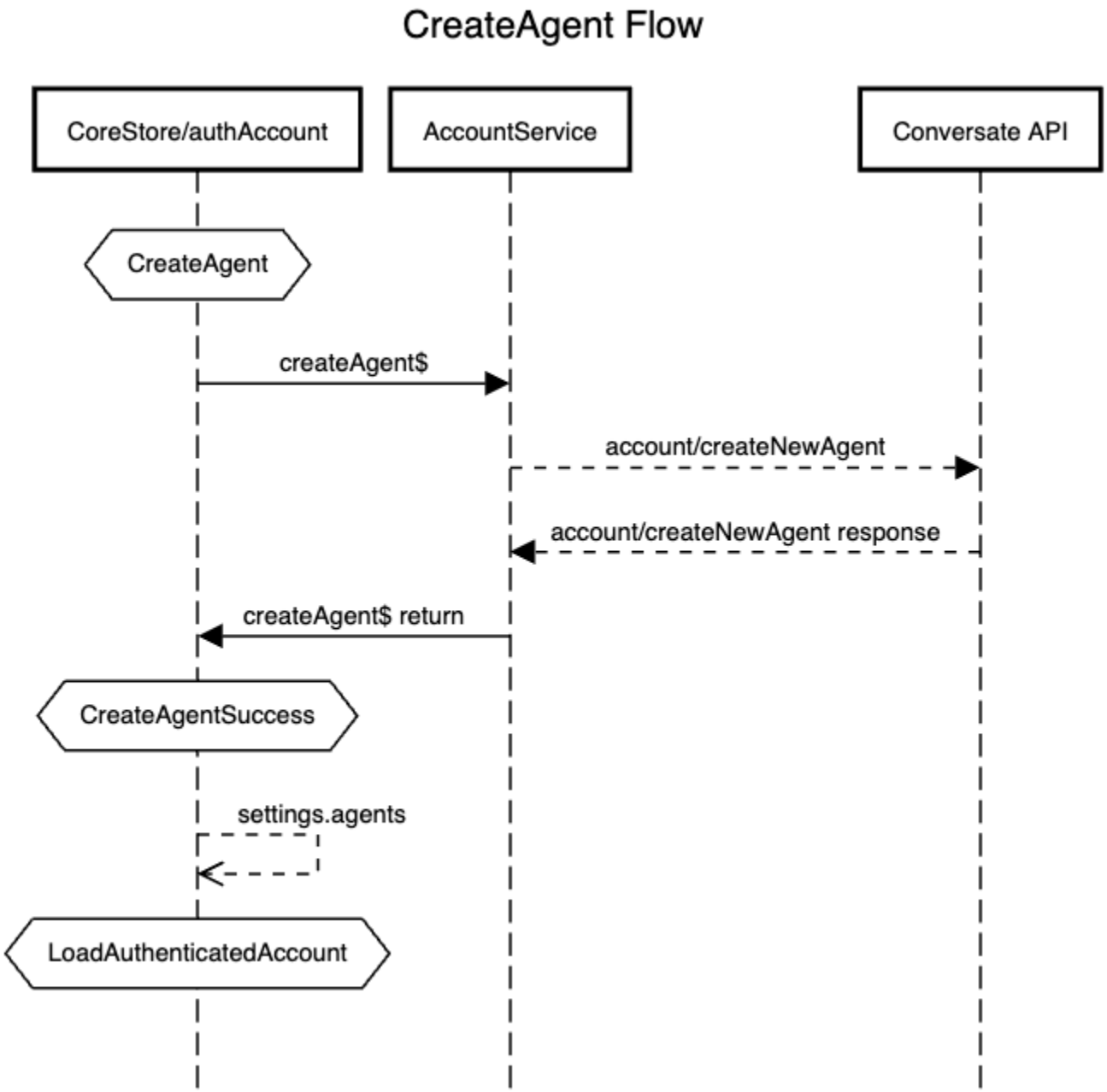
Update a users information. This can be done via the user info form component as part of the account login flow, or from the admin app's user details component





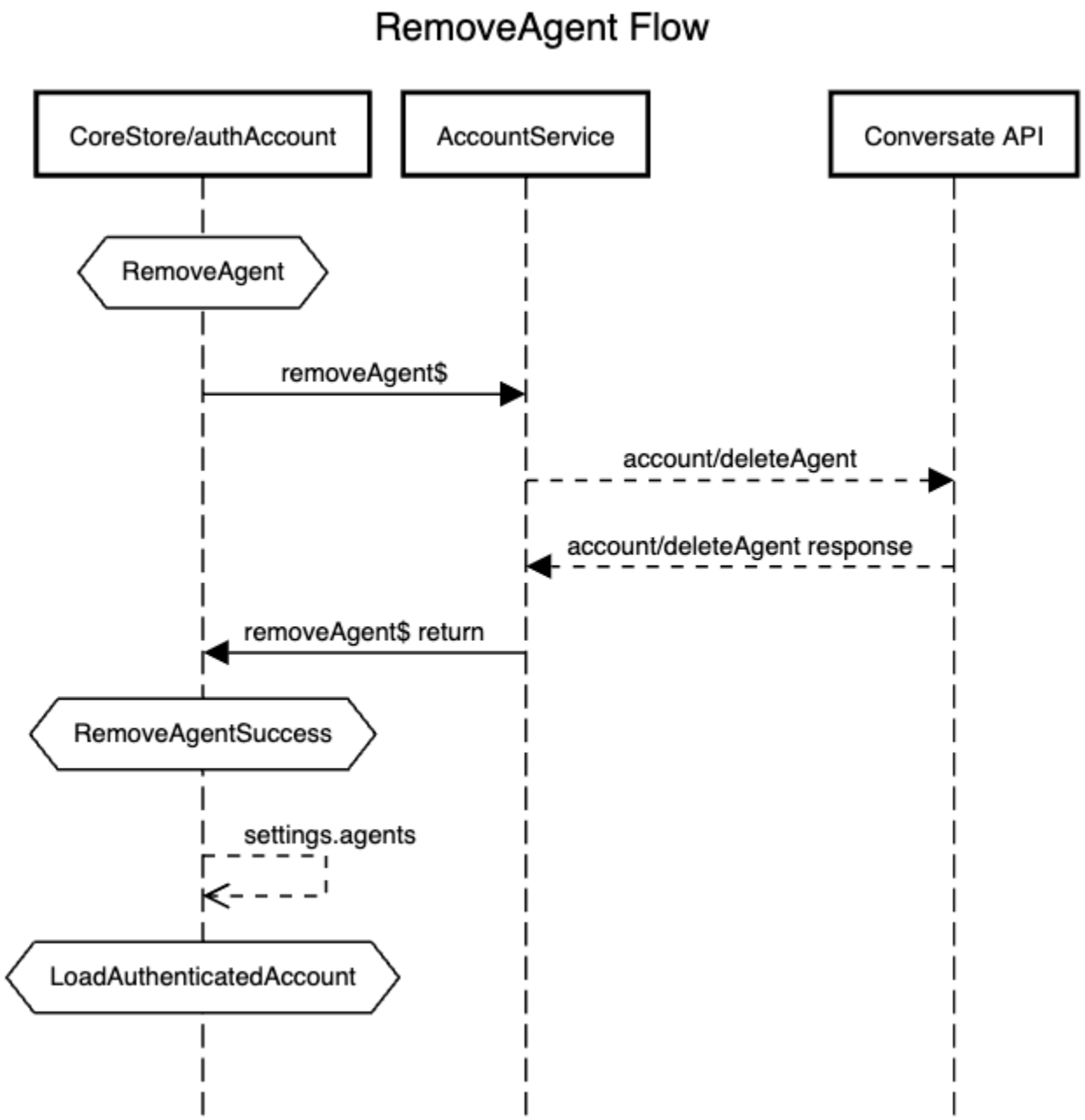
CreateAgent

creates a new agent on an existing organization.



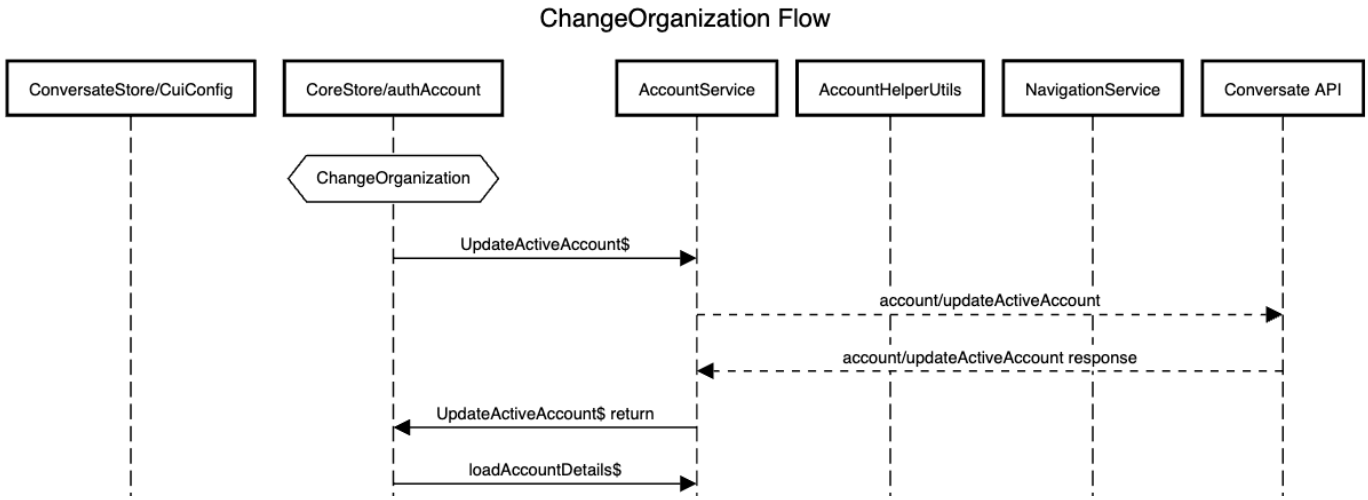
RemoveAgent

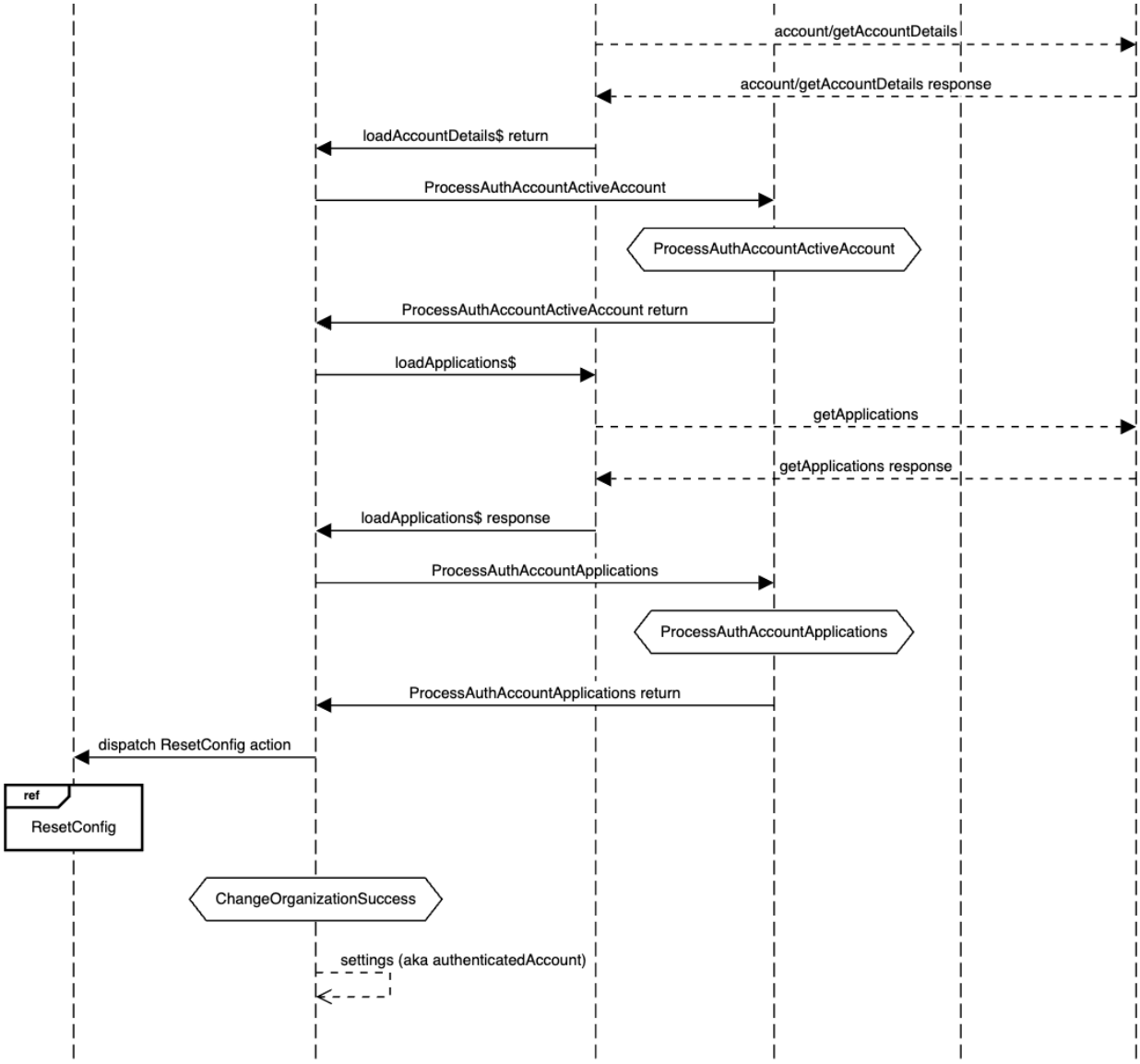
removes an assistant from an organization



ChangeOrganization

triggered when a user switches the organization, starts a flow to reinit the authenticatedAccount object.

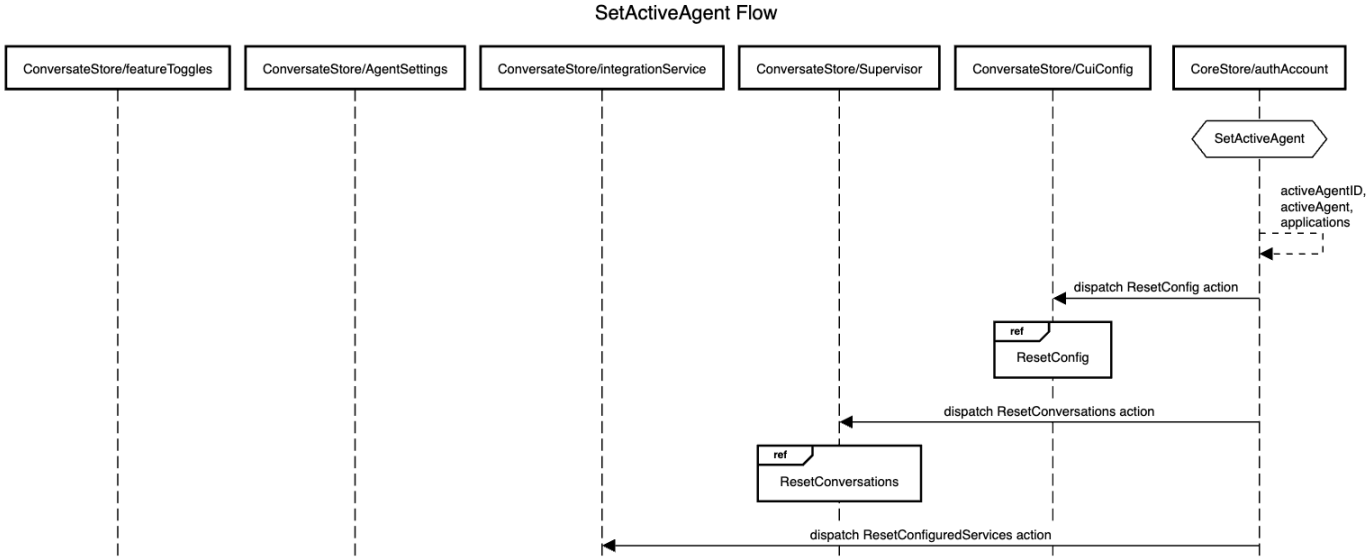


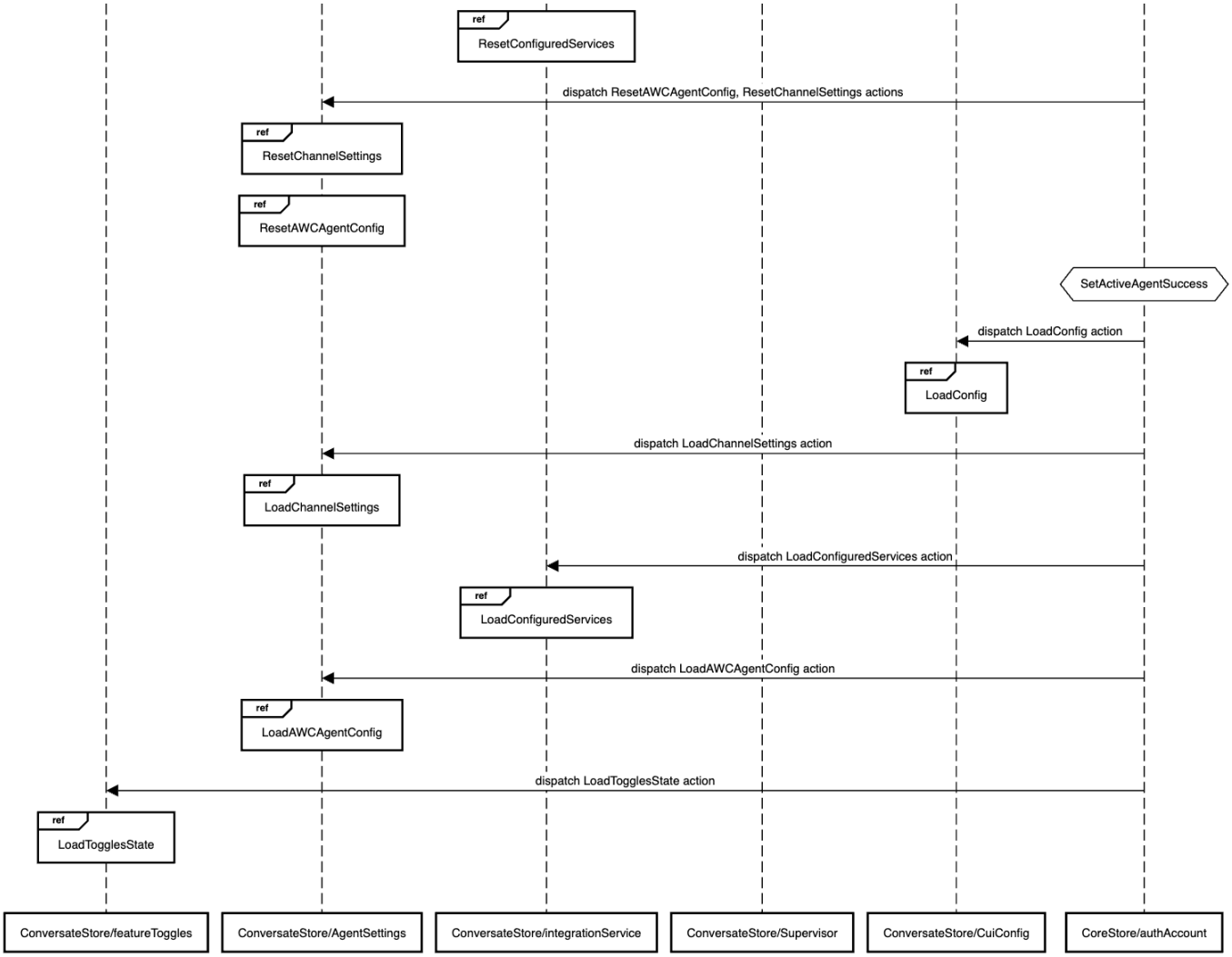


Refs

- ResetConfig

SetActiveAgent



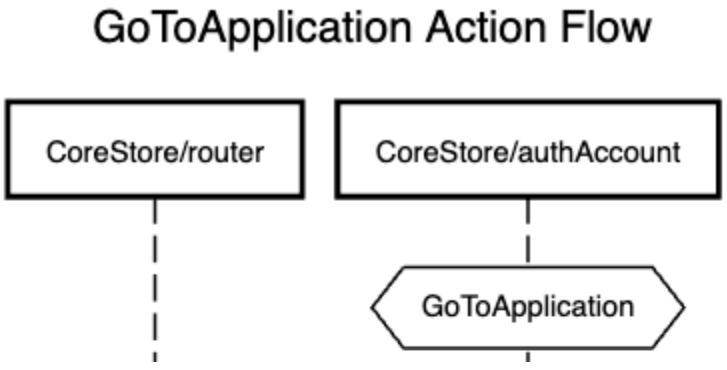


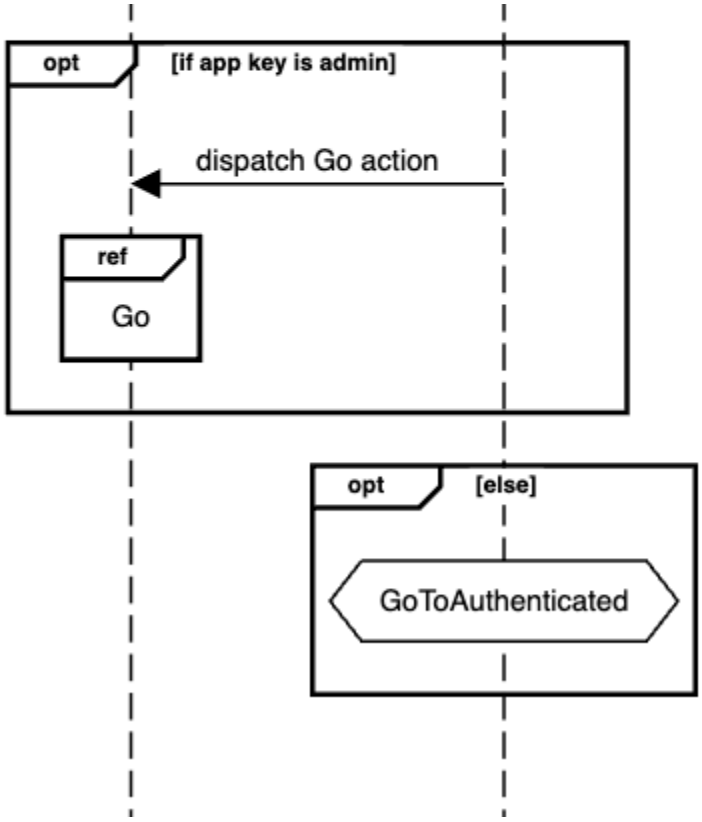
Refs

- ResetConfig
- ResetConversations
- ResetConfiguredServices
- ResetChannelSettings
- ResetAWCAgentConfig
- LoadConfig
- LoadChannelSettings
- LoadConfiguredServices
- LoadAWCAgentConfig
- LoadTogglesState

GoToApplication

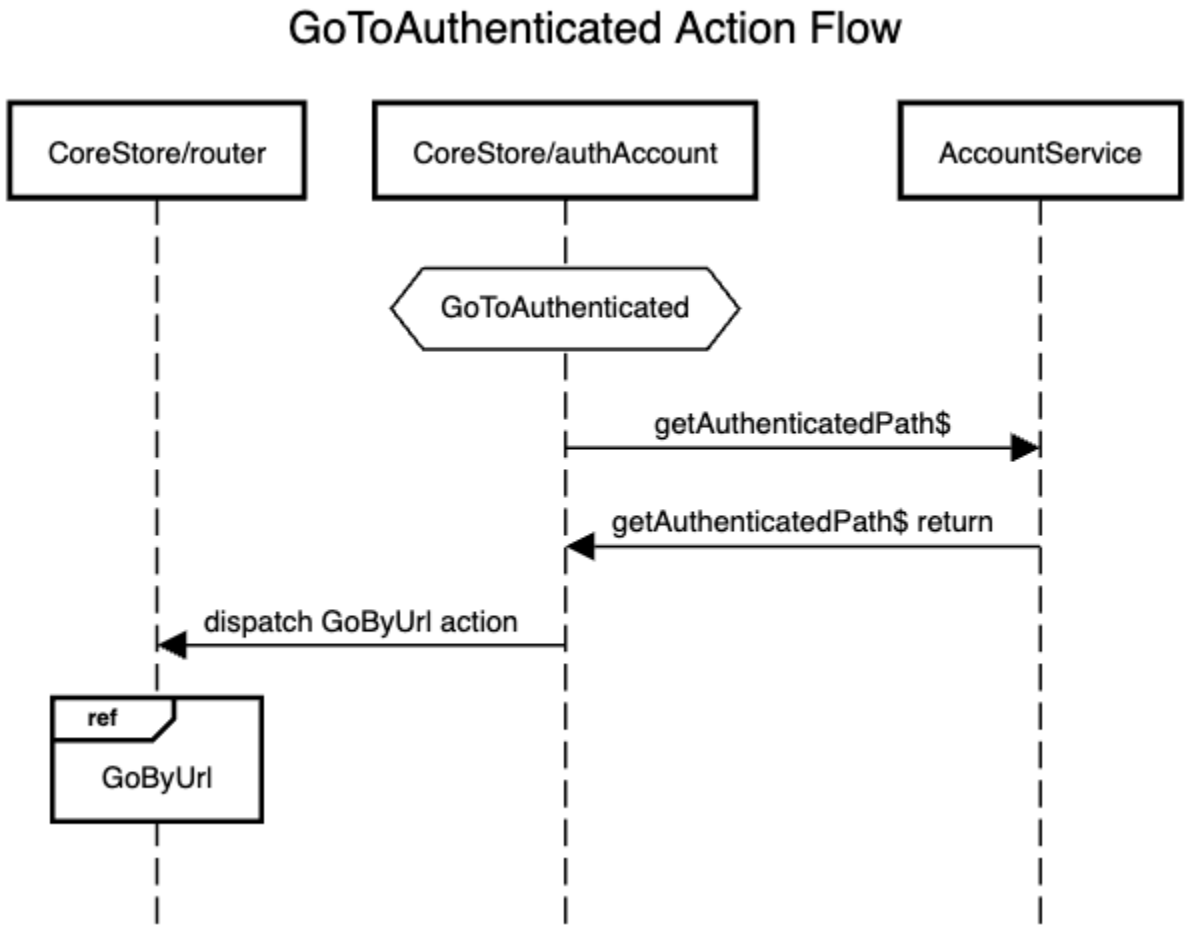
used to navigate to a sub application within CUI.





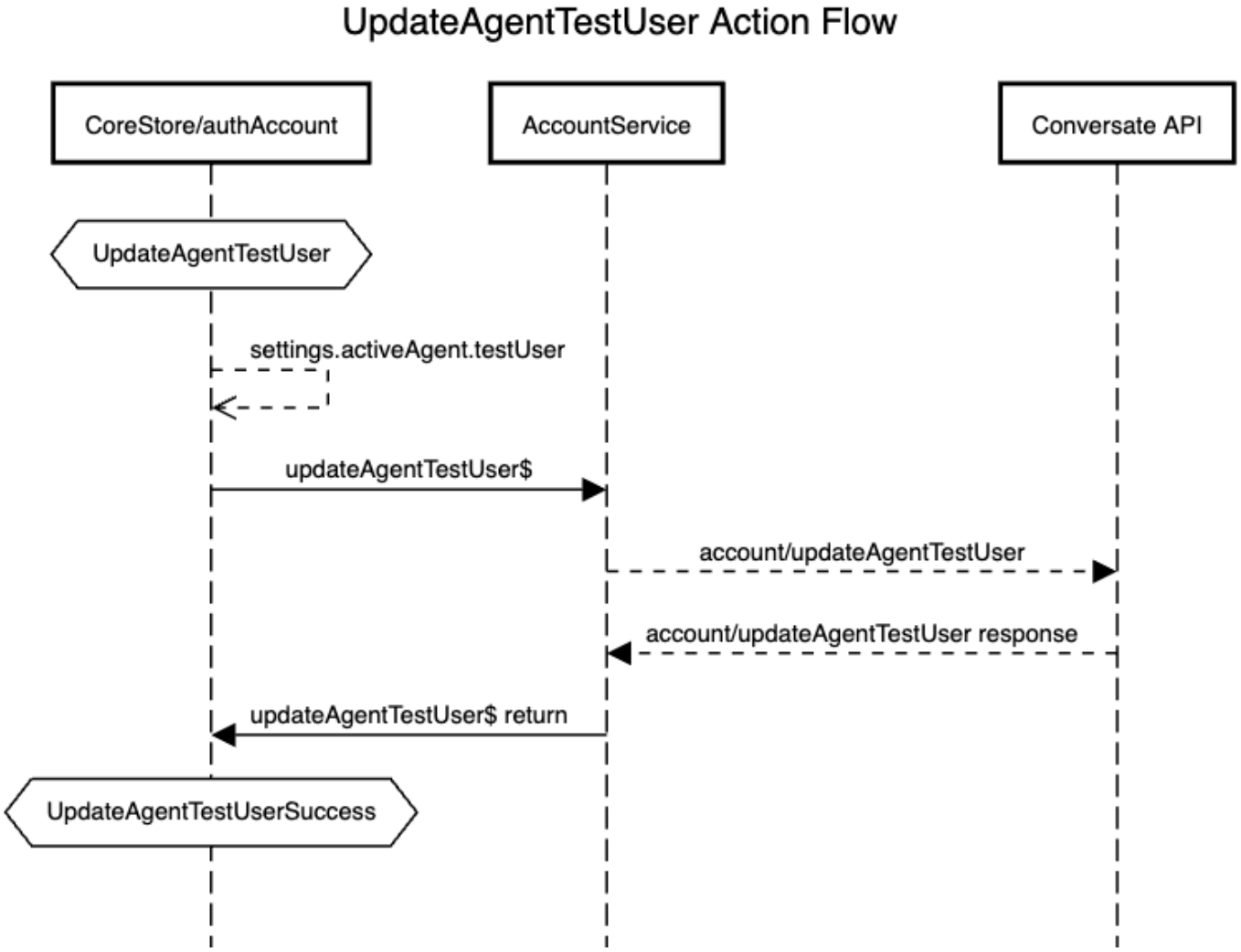
GoToAuthenticated

navigate to a url path. `getAuthenticatedPath$` will take in a path, and prepend to it the active organization's namespace, and the active assistant's slug. and example return would be `organizationName/assistantName/{pathProvided}`

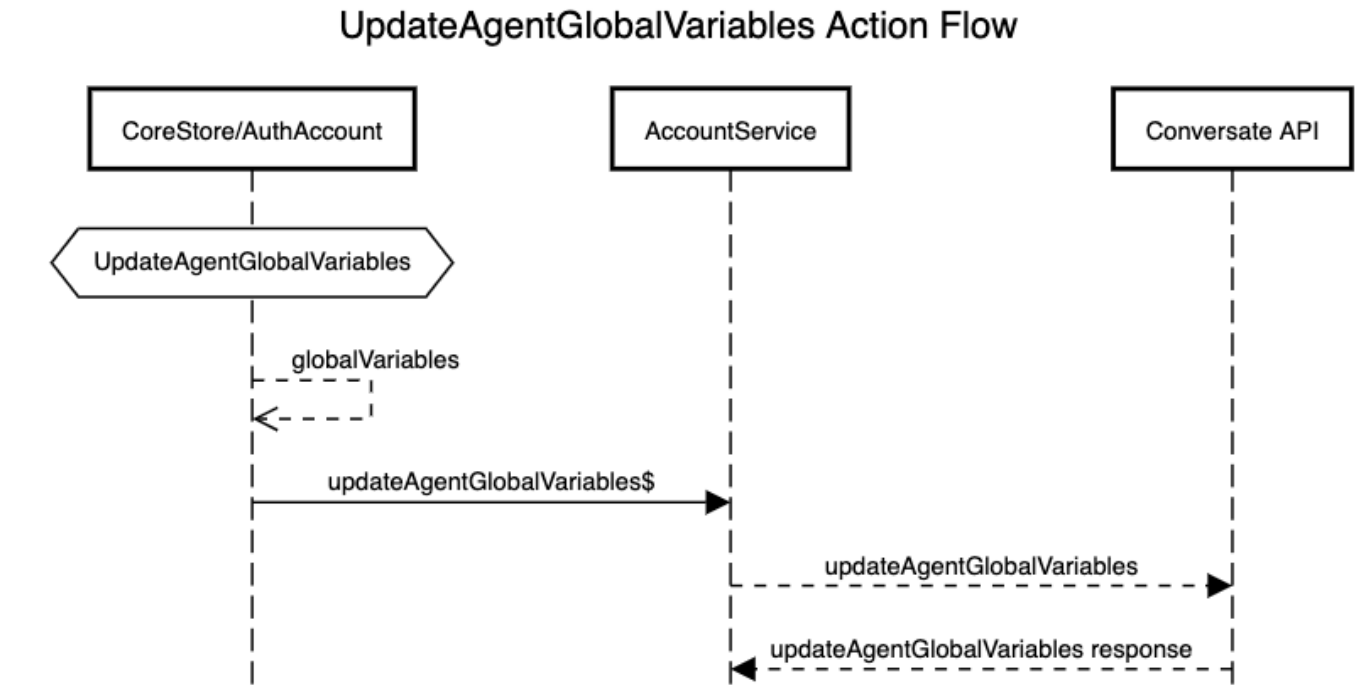


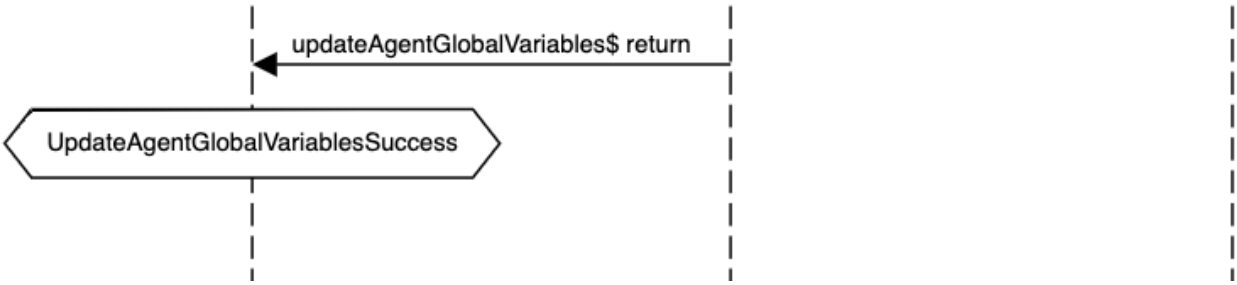
UpdateAgentTestUser

updates the test user for the active agent. The assistant's test user is the user for Dialogue Sim.



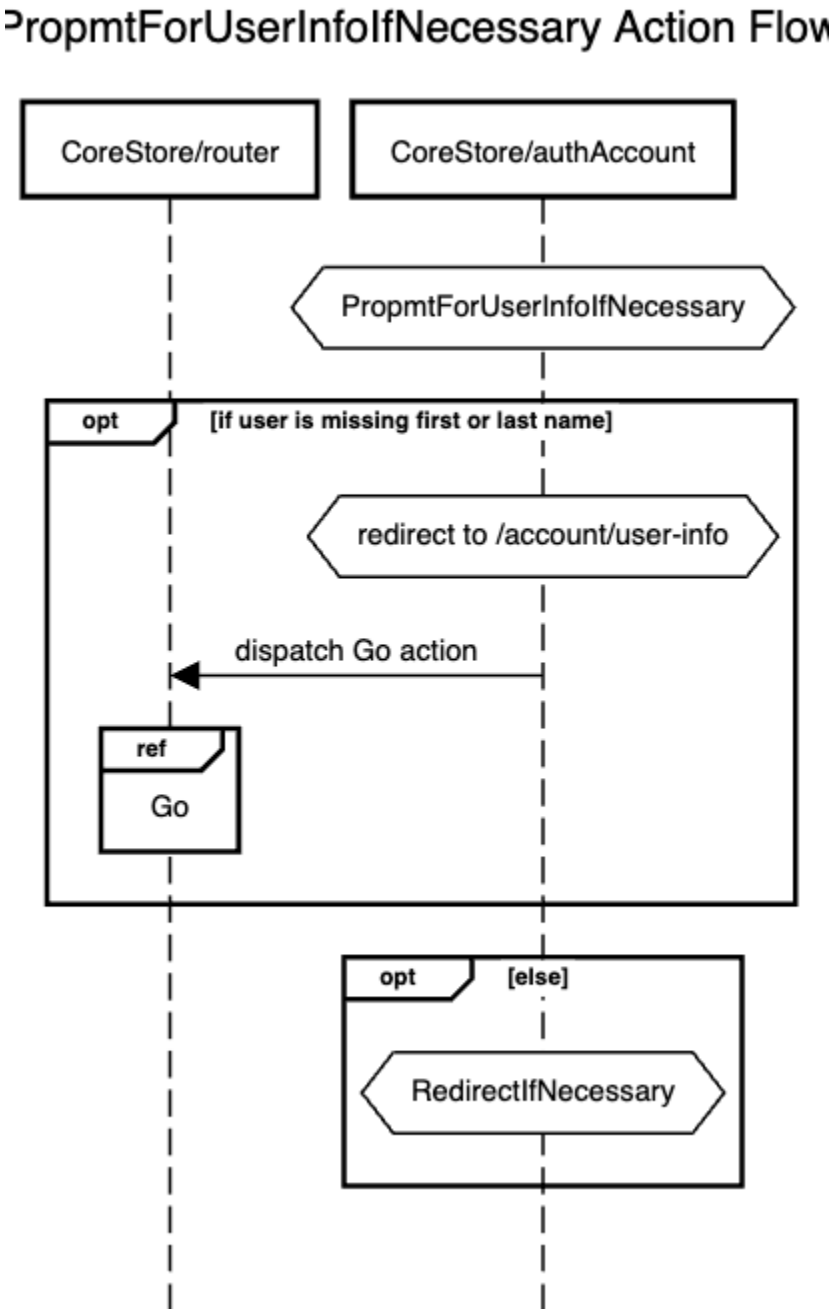
UpdateAgentGlobalVariables





PropmtForUserInfoIfNecessary

Will check if logged in user has their name populated, and if not will redirect to the user info form in account. If they are set, it will continue to RedirectIfNecessary Action

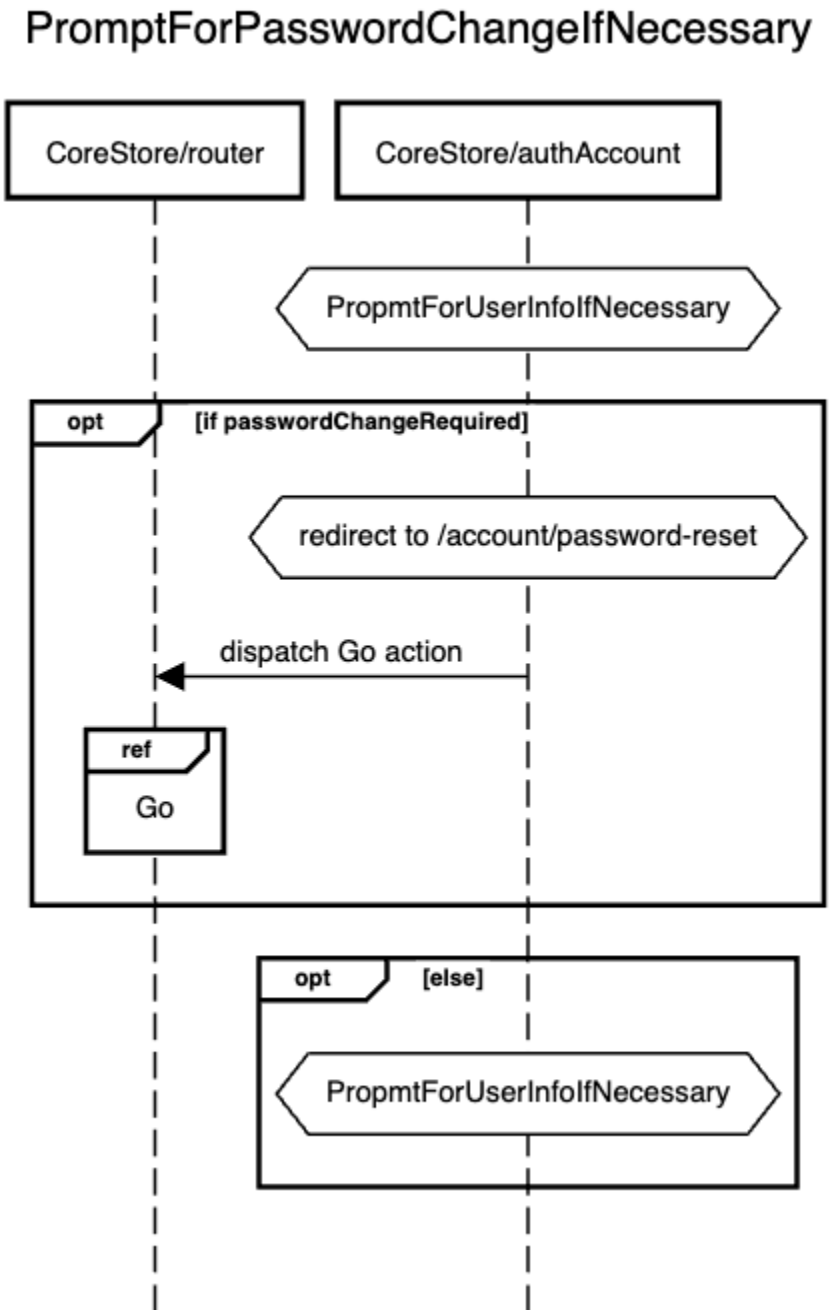


refs

- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98631418680/Navigation#RedirectIfNecessary

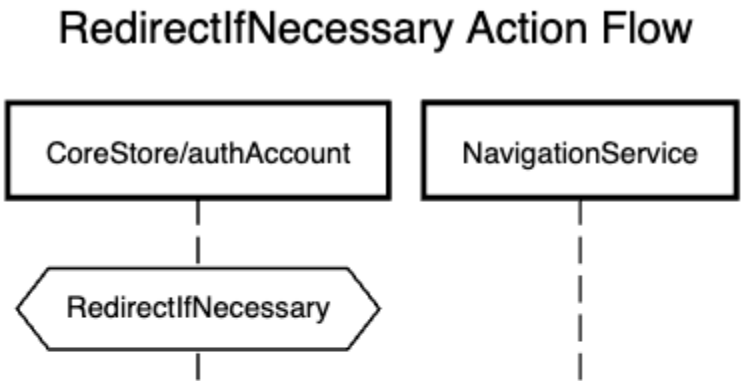
PromptForPasswordChangeIfNecessary

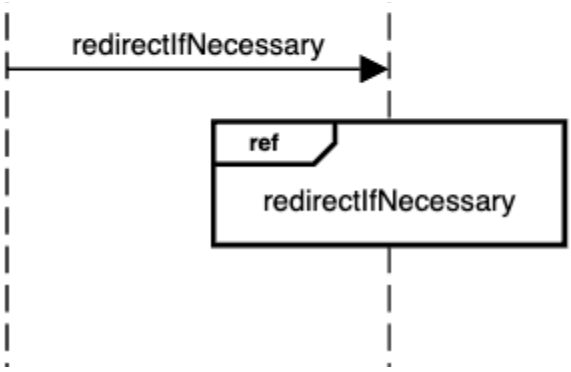
Will check if logged in user has been marked as needing a password change. If true, we redirect to the password change form in account, if false we continue to the User Info check.



RedirectIfNecessary

triggers the redirect if necessary function in Navigation Service





refs

- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98631418680/Navigation#RedirectIfNecessary

DynEnv State

Overview

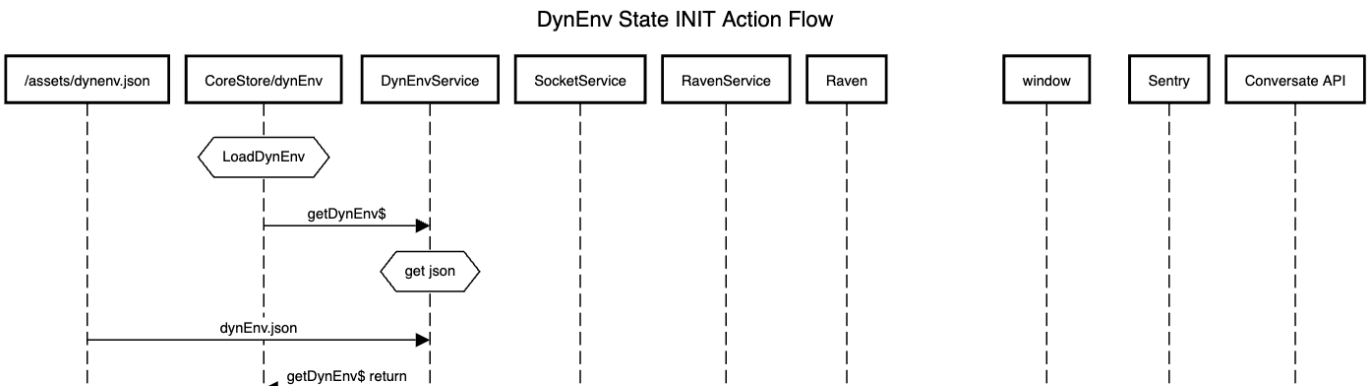
The DynEnv state is where we house the result of dynEnv loading the environment variables for CUI. The hydartion of this state is crucial for the initialization of CUI. Without it, we cannot establish our websocket connection to Conversate nor our connection to our logging service `Sentry`.

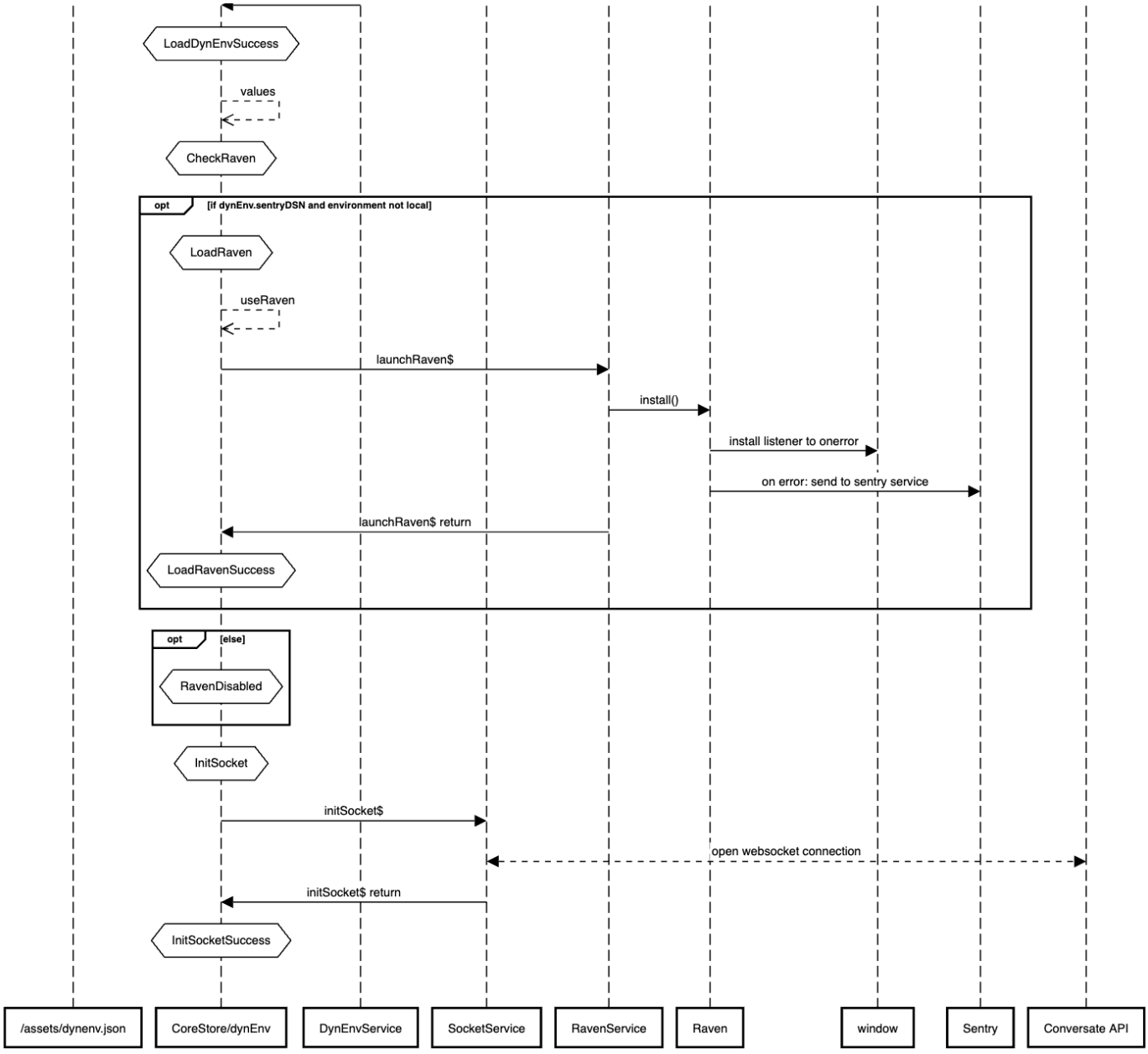
State Structure

```
{
  values: {
    applicationEnvironment?: string;
    applicationVersion?: string;
    conversate: string;
    apiBaseDomain?: string;
    apiVersion?: string;
    conversateDocker?: string;
    sentryDSN?: string;
    'lincConfig.useLinc'?: boolean;
  } | null;
  loaded: boolean;
  loading: boolean;
  useRaven: boolean;
}
```

SequenceFlows

DynEnv App Initialization Flow





Core Sub Modules

Overview

Core Module is broken up into a few sub modules in order to provide separation of concerns.

Submodules

Abe Common

Abe common module is responsible for housing logic, models, styles, and other utility logic in a place that is shared among all other modules. AbeCommonModule is imported into all other modules.

Material Custom

The Material Custom module does not have any views, instead it acts as a proxy importer. This allows us to import just the Material Custom module in other modules, instead of importing a bunch of individual modules from the material library in many modules.

Navigation

The Navigation module is responsible for housing standard logic, models, and UI components related to navigation.

Notification

The Notification Module is responsible for housing logic, models, and UI components for our system notifications, and modals

Abe Common

Overview

Abe common module is responsible for housing logic, models, styles, and other utility logic in a place that is shared among all other modules. AbeCommonModule is imported into all other modules.

Auth Utils Service

The `AuthUtilsService` is a singleton object that is used mostly to provide methods for common authentication related tasks, such as checking for a JWT token and its validity, decoding a JWT token's contents, and managing the token in local storage.
Connects to

- JWT Helper Service
- local storage

Used By

- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98607498270/Core#Socket-Service
- Auth State
- App Context Picker
- Contact Center Redirect Component

Ecma-Modules Service

The `ECMAModulesService` is a singleton object that acts as a wrapper for a few external libraries, currently FileSaver, stringify, and idb-keyval.
Used By

- Compliance
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98623095146/CUI+Config#Config-Service
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98607432614/Dialogue+Explorer#Conversation-Service

Object Utils Service

The `ObectUtilsService` is a singleton object that provides a few methods for dealing with common tasks related to Objects such as `flatten`.
Used By

- Channel Settings
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98607432614/Dialogue+Explorer#Responses

Socket Utils Service

The SocketUtilsService is a singleton object that has only one method, `responseFilter`. It is used by the SocketService to determine if a ws event is the response to a particular request we made. It does this by checking the channel and command of the incoming events, and when the channel and command match the request's, we allow that event to be passed back to the subscriber chain.

Material Custom

Overview

The Material Custom module does not have any views, instead it acts as a proxy importer. This allows us to import just the Material Custom module in other modules, instead of importing a bunch of individual modules from the material library in many modules.

Navigation

Overview

The Navigation module is responsible for housing standard logic, and models related to navigation. It also is where you will find the NavigationService and NavigationGaurd singletons. Additionally, it houses the UI components used for CUI's navigation bar.

Navigation Guard

Navigation Service

The Navigation Service singleton provides standard functions to deal with navigation tasks
URLS To Redirect

Certain urls are designated to be redirected when checked as part of the login flow. These are as follows

- /account/login
- /account/create

- /account/password-reset
- /account/user-info

RedirectIfNecessary

This process is used post token verification, typically on init of the application. It will redirect to the app picker in the following situations

- A preauth url has been set and it is a url designated to be redirected
- current url is somehow missing or in redirect list
- preauth or current url is in unauthorized org/assistant/app scope

PushRedirect

Store a url path, sans org/agent parameters in the singleton's `REDIRECT_STORE` property
PopRedirect

Attempts to navigate to `REDIRECT_STORE` using the GoToAuthenticated action

Notification

Overview

The Notification Module is responsible for housing logic, models, and UI components for our system notifications, and modals

Notification Service

The Notifcation Service is a singleton object that provides methods to open various modals in the UI.
ModalsdialogGeneric

A generic modal that will render with a Title and some text content
dialogSpinner

A modal with a Material Spinner in it
dialogPendingChanges

A simple modal notifying users of pending changes, requesting if they would like to continue
dialogDataDrivenIntentSelector

A modal used to assign a selected method from an Integration service
dialogDeveloperData

A modal that displays an Ace Editor to show developer data.
dialogTrainUtterance

A modal to add an unrecognized utterance to a selected intent.
dialogChangePasswordRequired

A modal that informs a logged in user that their password has been marked as needing changed. It also houses a form for the user to change their password.
dialogTokenExpiring

A modal with a countdown till token expiry. Upon CTA click, a token refresh process will be executed.
dialogApiTryIt

A modal that allows users to try out API calls for messages API
dialogSupervisorAddTag

A modal that allows users to add new or existing tags to a conversation
dialogLincSlotInstanceSelector

a modal that allows users to select an existing slot instance within the LINC editor

Admin App

Overview

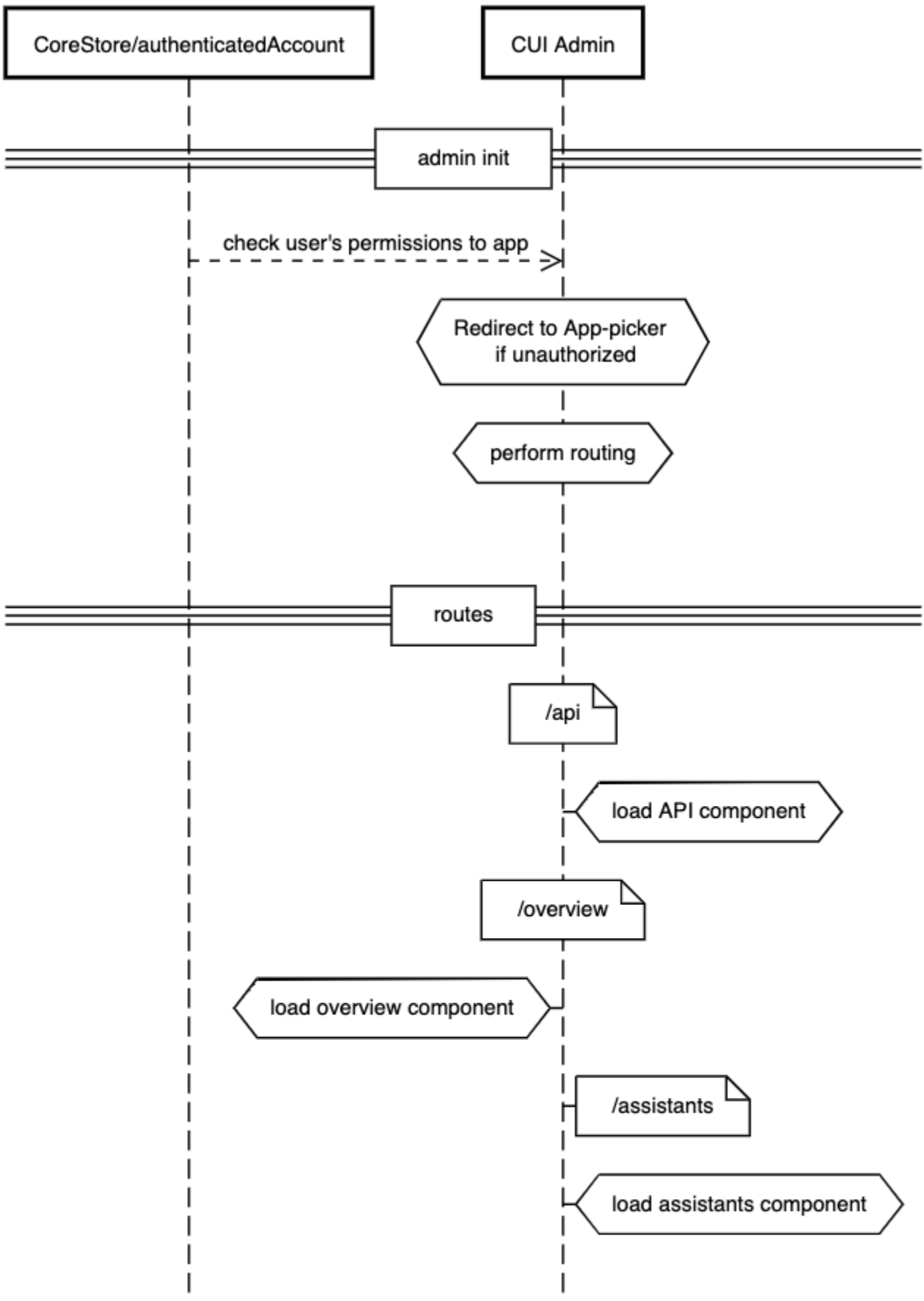
Admin module is where administrators of Organizations, or Assistants, can assign/remove access for other users, and control permissions. CUI users who are an administrator of an organization, assistant, or application will have access to the admin module. Additionally, users who belong to the super user group (internal only) will have administrator rights on all levels.

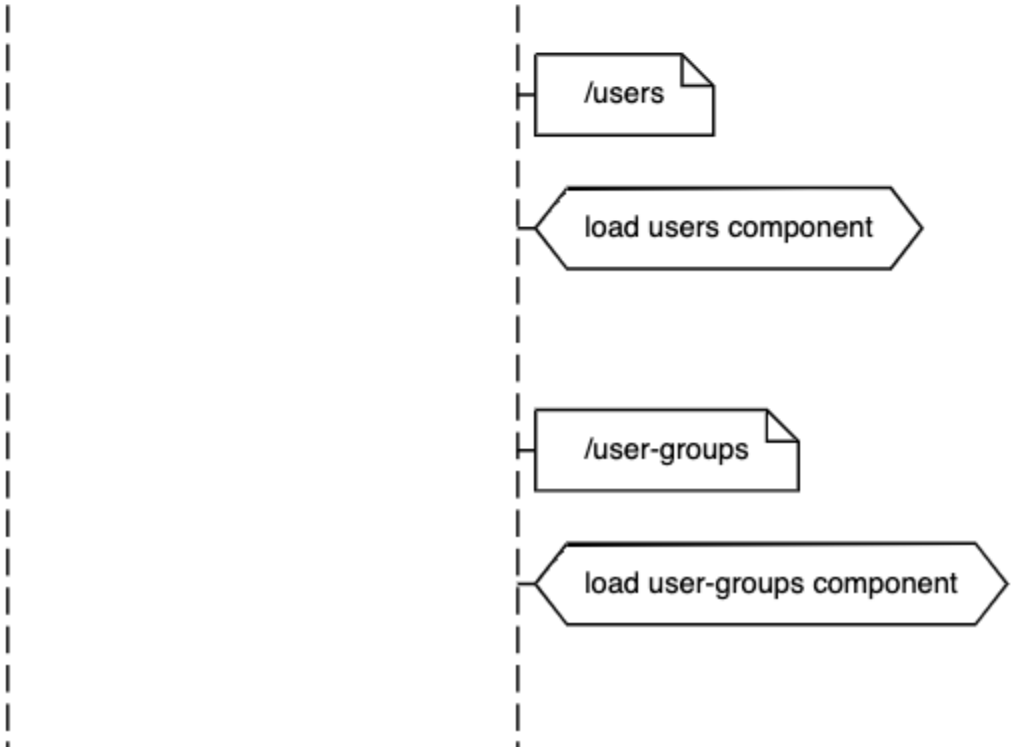
Reference

- Users, Groups & Roles

Sequence Flow

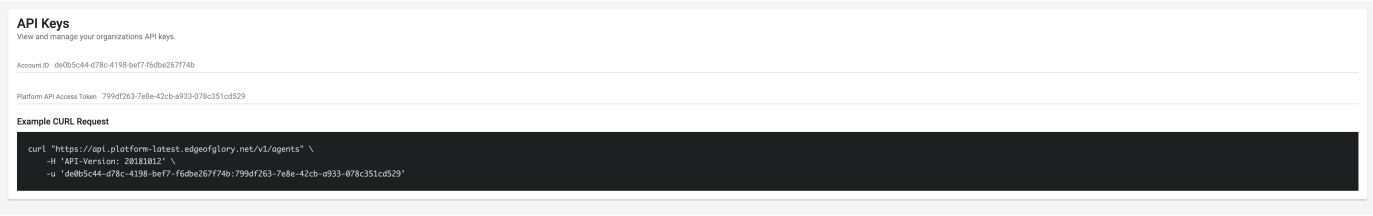
CUI Admin


CoreStore/authenticatedAccount
CUI Admin
admin init
check user's permissions to app
Redirect to App-picker
if unauthorized
perform routing
routes
/api
load API component
/overview
load overview component
/assistants
load assistants component



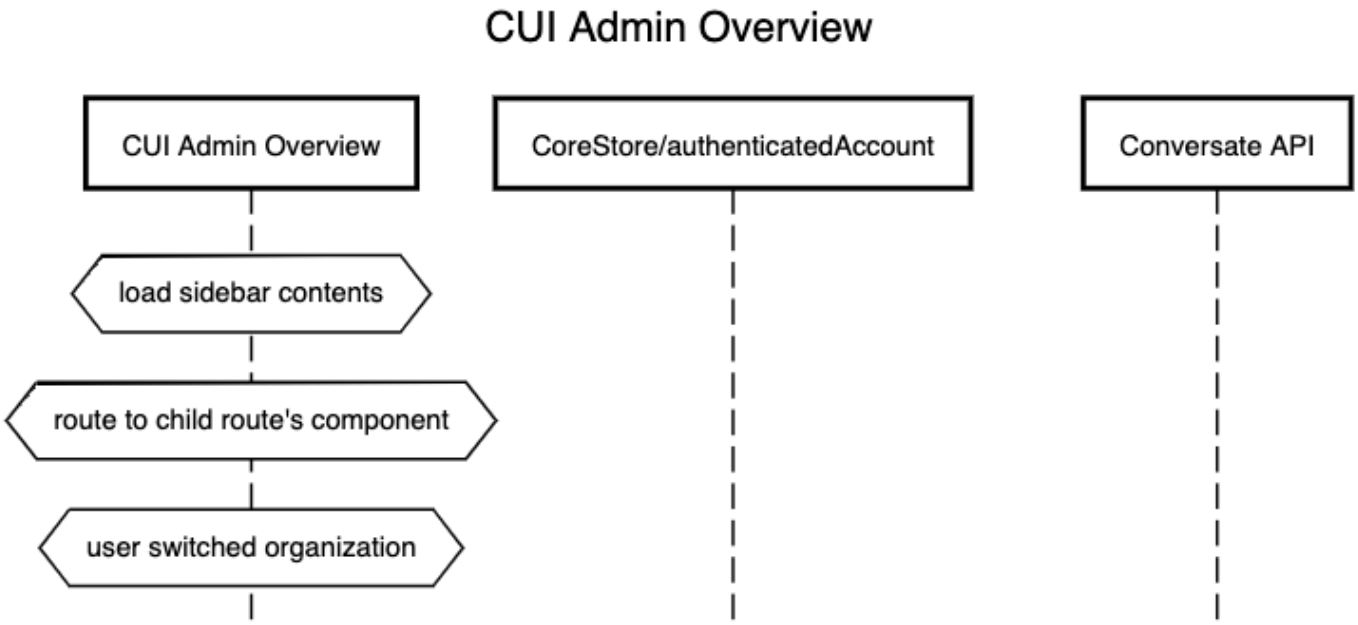
API

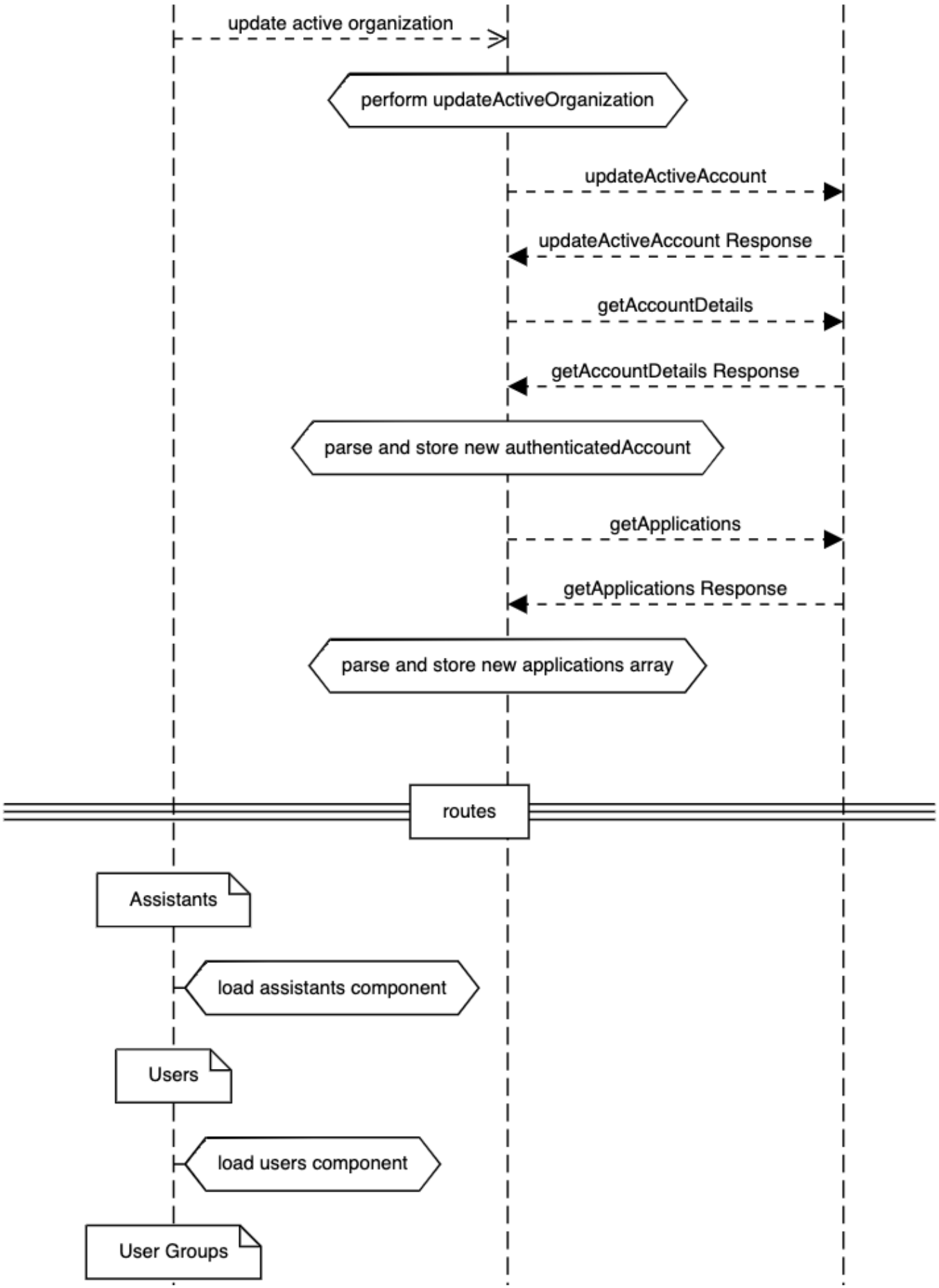
The API page in admin module contains information about how to connect to the assistant API, user with access will be able to find the account ID, access token, and an example CURL request.

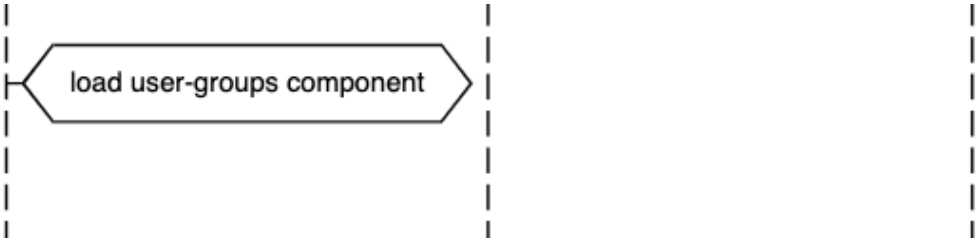


Overview

Overview is the main section of admin, it contains 3 child routes, Assistants, Users, and User Groups. The scope context for the admin overview pages is the Organization itself. Only one organization can be viewed at a time.




update active organization
perform updateActiveOrganization
updateActiveAccount
updateActiveAccount Response
getAccountDetails
getAccountDetails Response
parse and store new authenticatedAccount
getApplications
getApplications Response
parse and store new applications array
routes
Assistants
load assistants component
Users
load users component
User Groups



Assistants

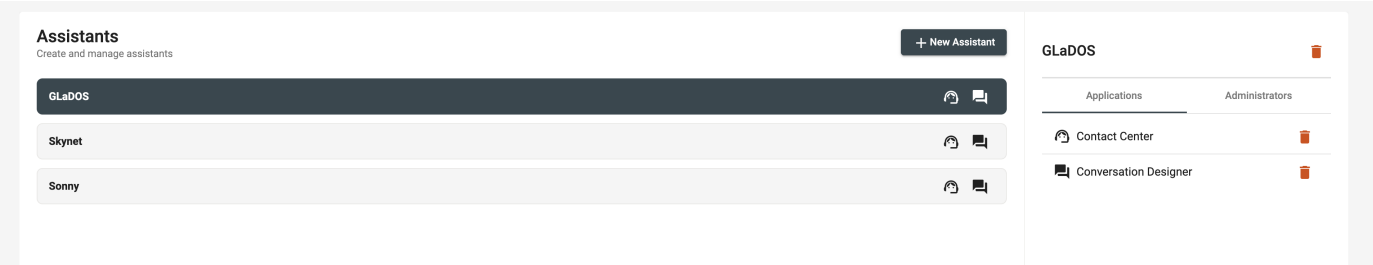
Overview

The Assistants view in Admin App is where administrators can manage assistants on an organization, their active applications, and user's permissions.

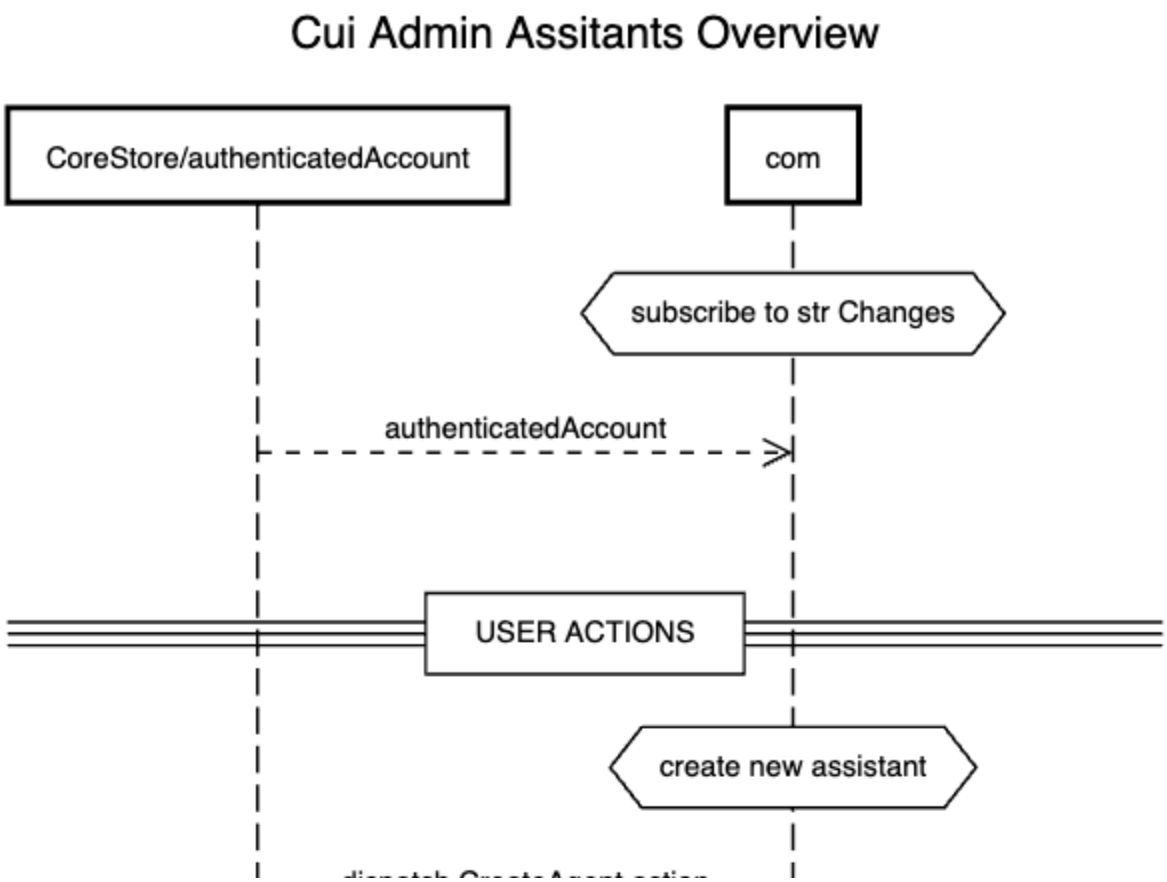
Functions

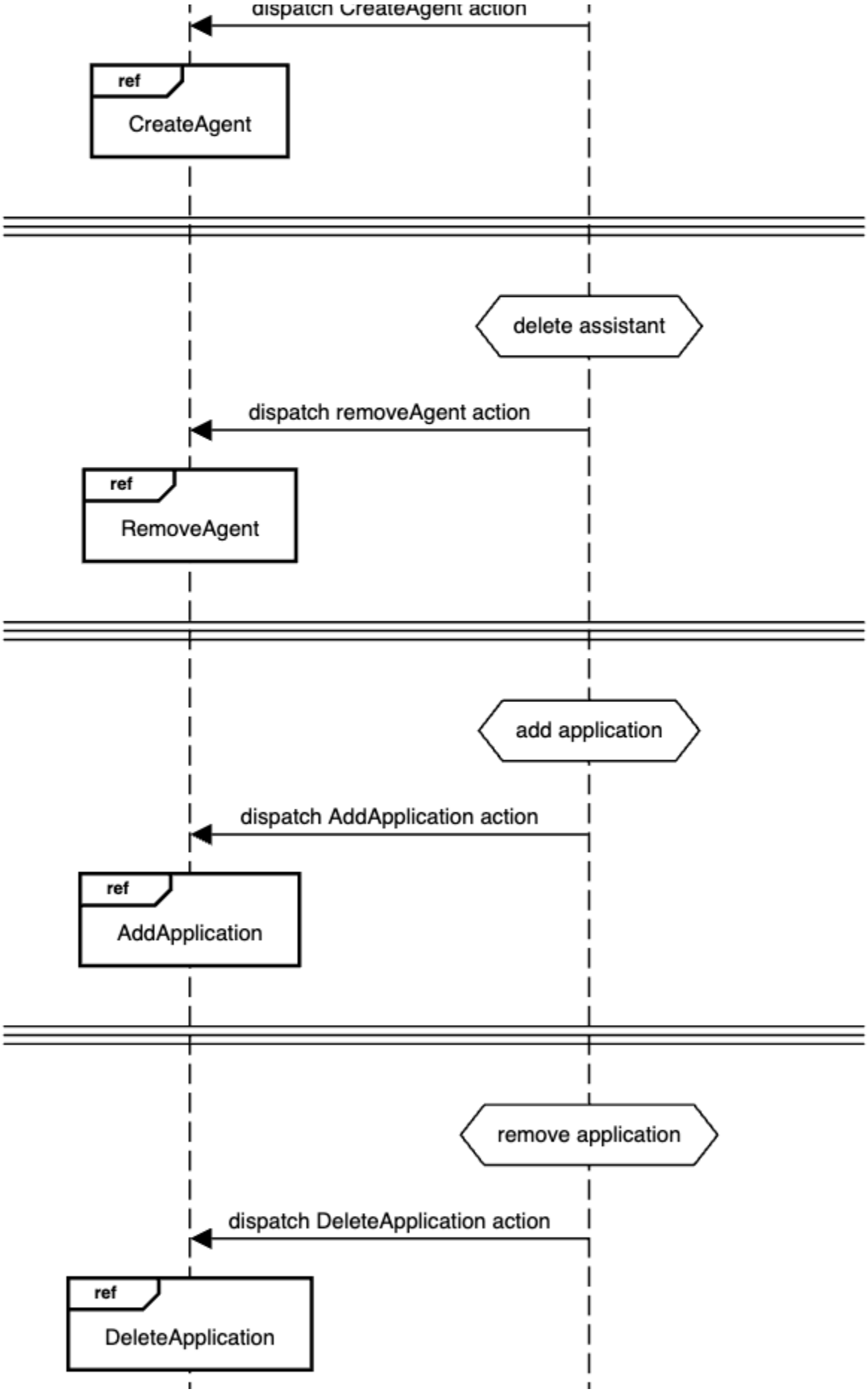
- Create/Delete Assistants
- Add/Remove Applications to an assistant
- Add/Remove Users as Admin of Assistant

UI Preview



Sequence flow




dispatch CreateAgent action
ref
CreateAgent
delete assistant
dispatch removeAgent action
ref
RemoveAgent
add application
dispatch AddApplication action
ref
AddApplication
remove application
dispatch DeleteApplication action
ref
DeleteApplication

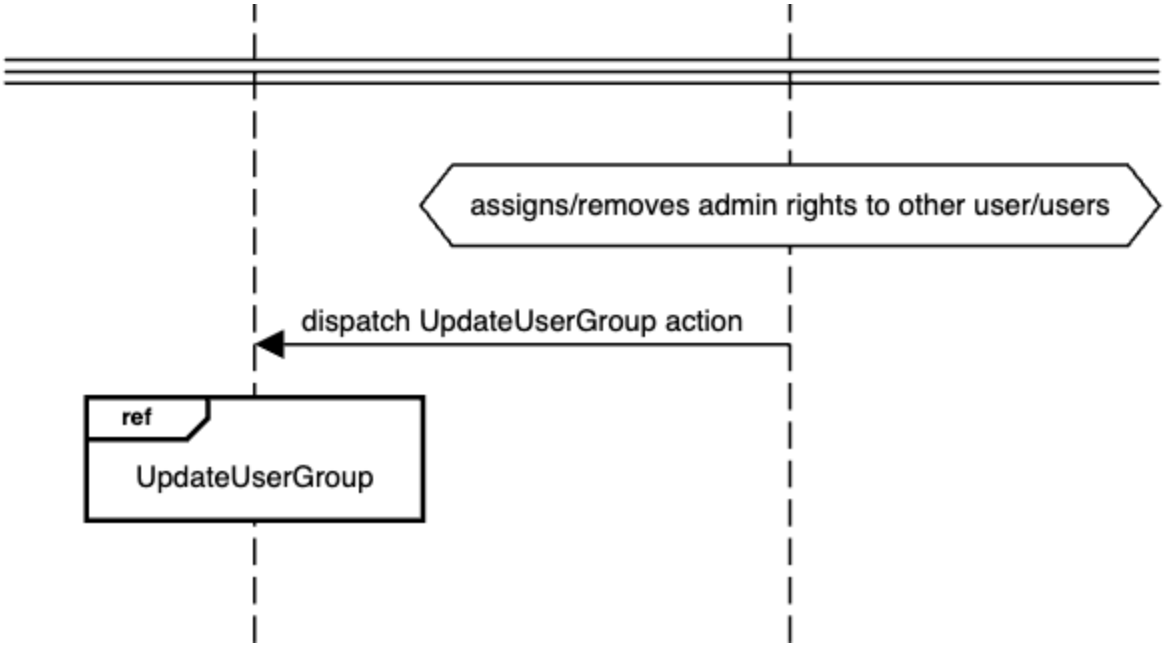


Chart References

- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637152263/Authenticated+Account+State#CreateAgent
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637152263/Authenticated+Account+State#RemoveAgent
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637152263/Authenticated+Account+State#AddApplications
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637152263/Authenticated+Account+State#DeleteApplications
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637152263/Authenticated+Account+State#UpdateUserGroup

Users

Overview

The users page is where administrators can add/remove/manage users and their permissions. The users listed are users that have access to the organization. Administrators can assign users access to assistants, or to specific user groups.

When editing a user, The assistants and user groups panes are coupled, so changes on one can impact the other.

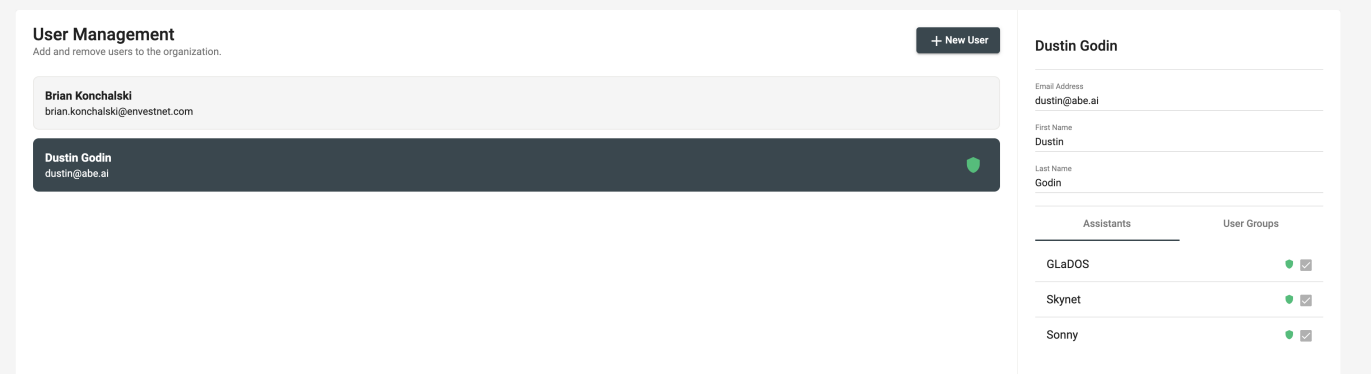
Dependencies

-
-
- Account Service

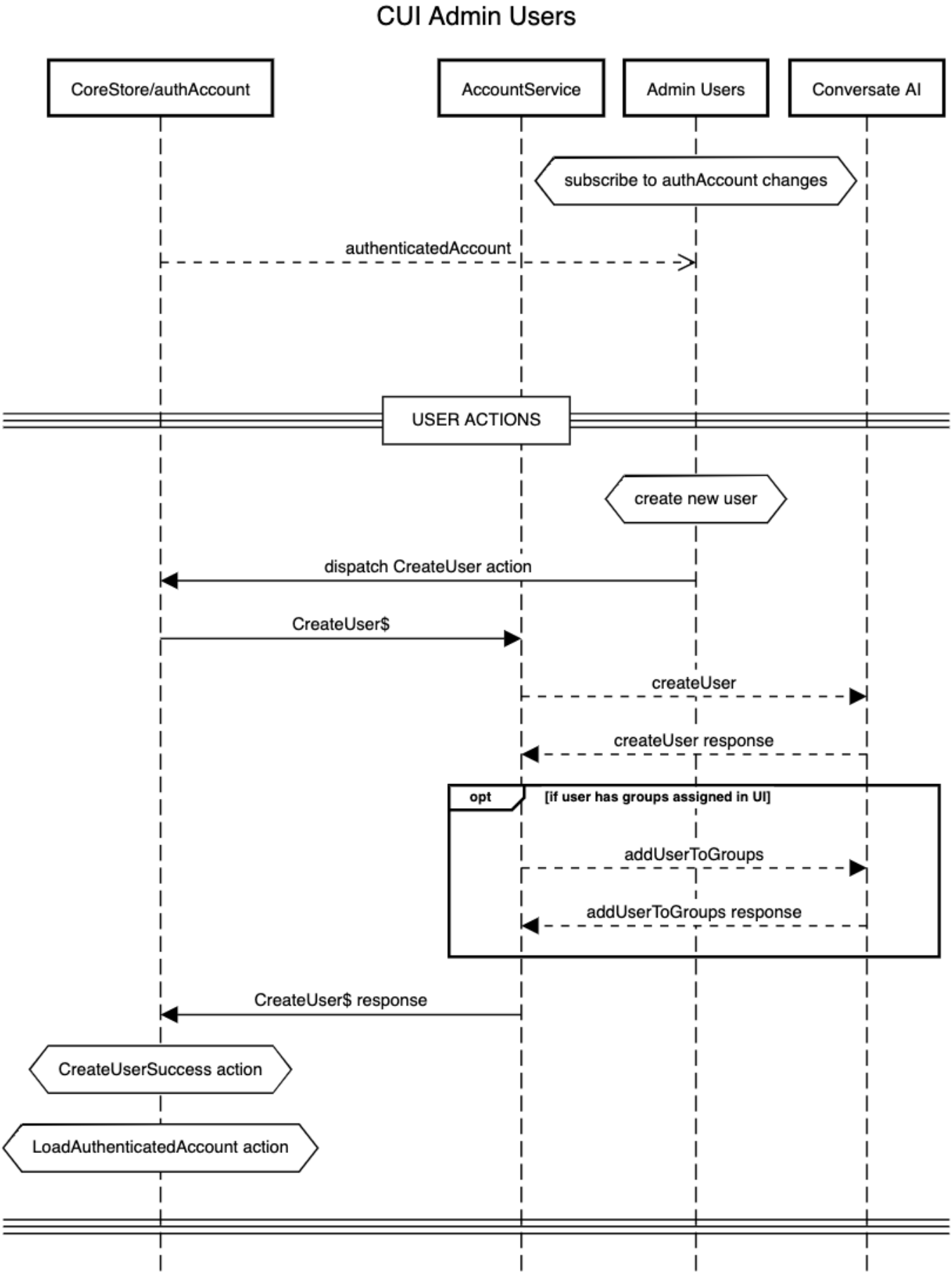
References

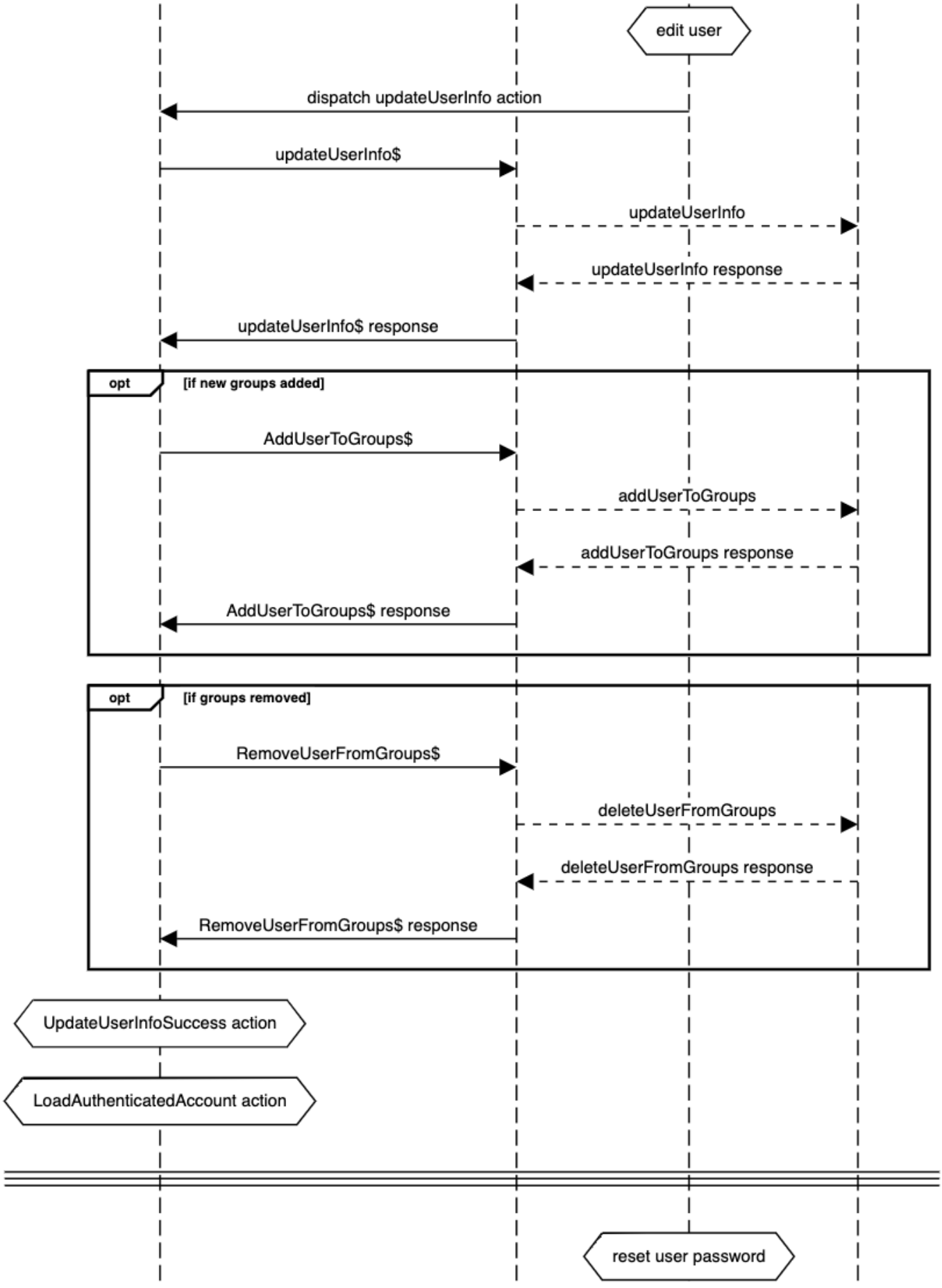
-

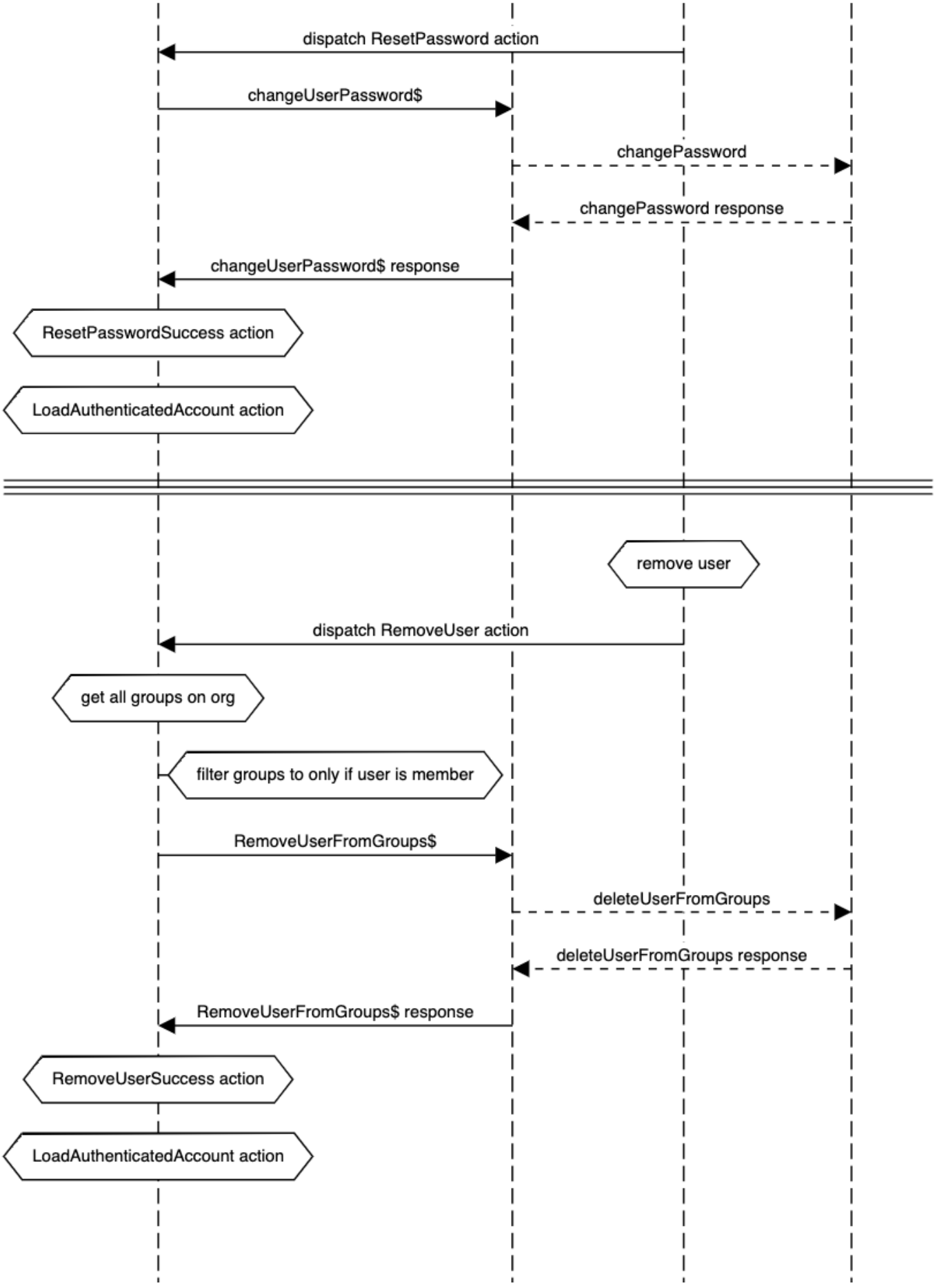
UI Preview



Sequence Flow




edit user
dispatch updateUserInfo action
updateUserInfo$
updateUserInfo
updateUserInfo response
updateUserInfo$ response
opt
[if new groups added]
AddUserToGroups$
addUserToGroups
addUserToGroups response
AddUserToGroups$ response
opt
[if groups removed]
RemoveUserFromGroups$
deleteUserFromGroups
deleteUserFromGroups response
RemoveUserFromGroups$ response
UpdateUserInfoSuccess action
LoadAuthenticatedAccount action
reset user password


dispatch ResetPassword action
changeUserPassword$
changePassword
changePassword response
changeUserPassword$ response
ResetPasswordSuccess action
LoadAuthenticatedAccount action
remove user
dispatch RemoveUser action
get all groups on org
filter groups to only if user is member
RemoveUserFromGroups$
deleteUserFromGroups
deleteUserFromGroups response
RemoveUserFromGroups$ response
RemoveUserSuccess action
LoadAuthenticatedAccount action

User Groups

Overview

This view is a user-group centric view, allowing administrators to create/delete/modify user groups. User groups belong to a parent entity, which can be the organization, an assistant, or even an application.

When editing a user group, admins can quickly assign multiple users to the group, and also modify the groups included permissions.

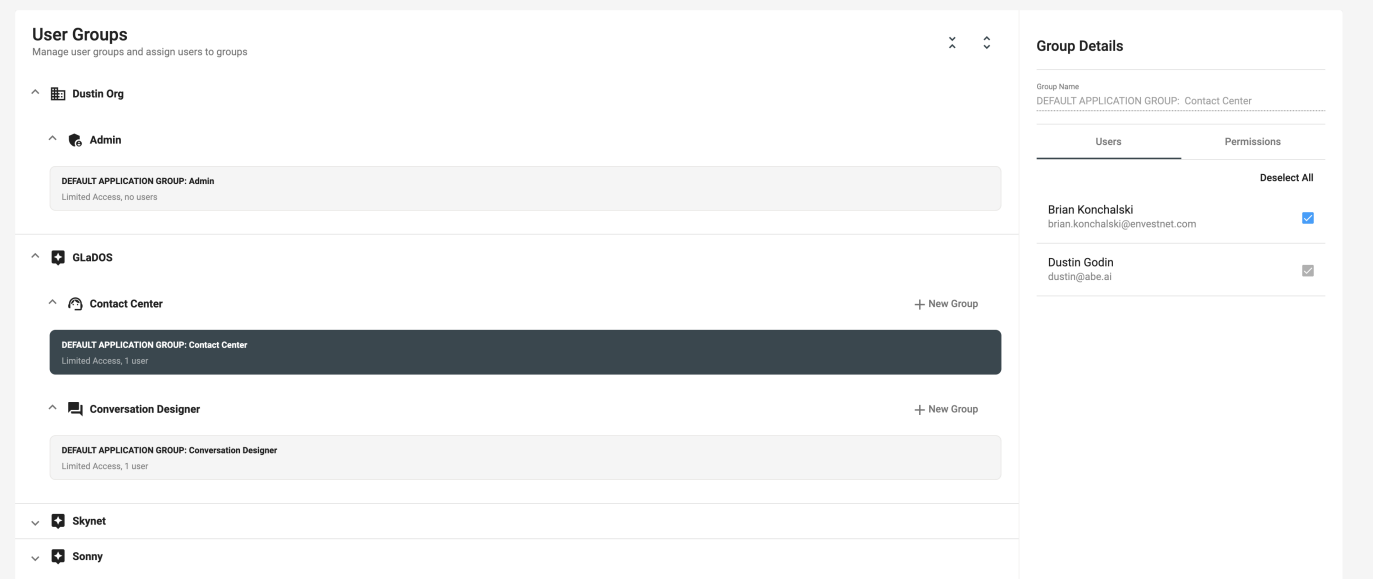
Dependencies

-
-
- Account Service

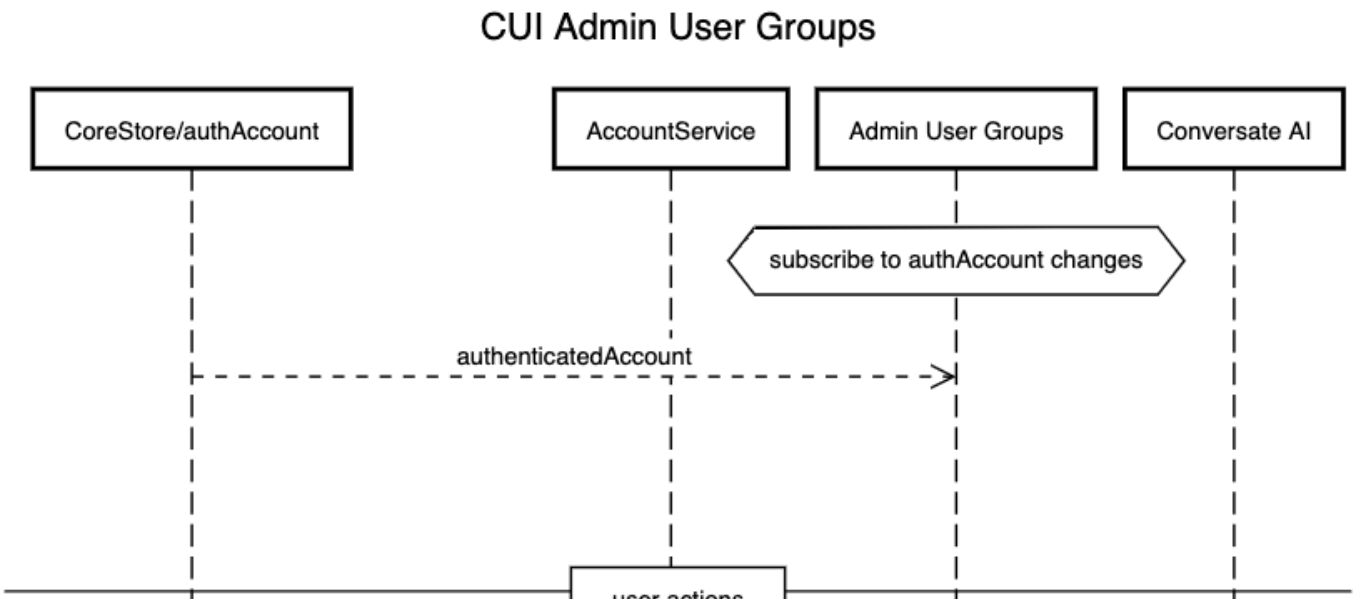
References

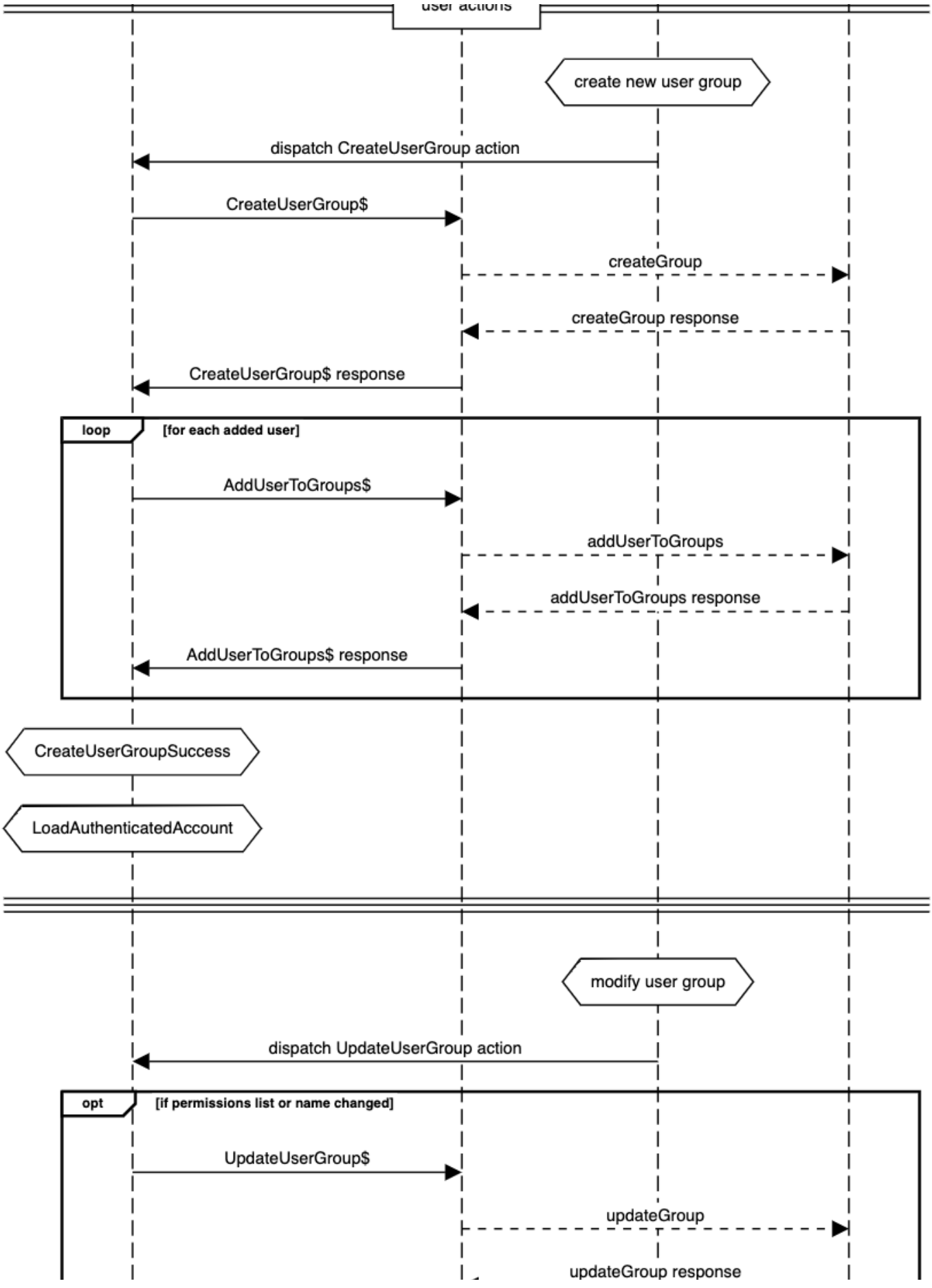
-

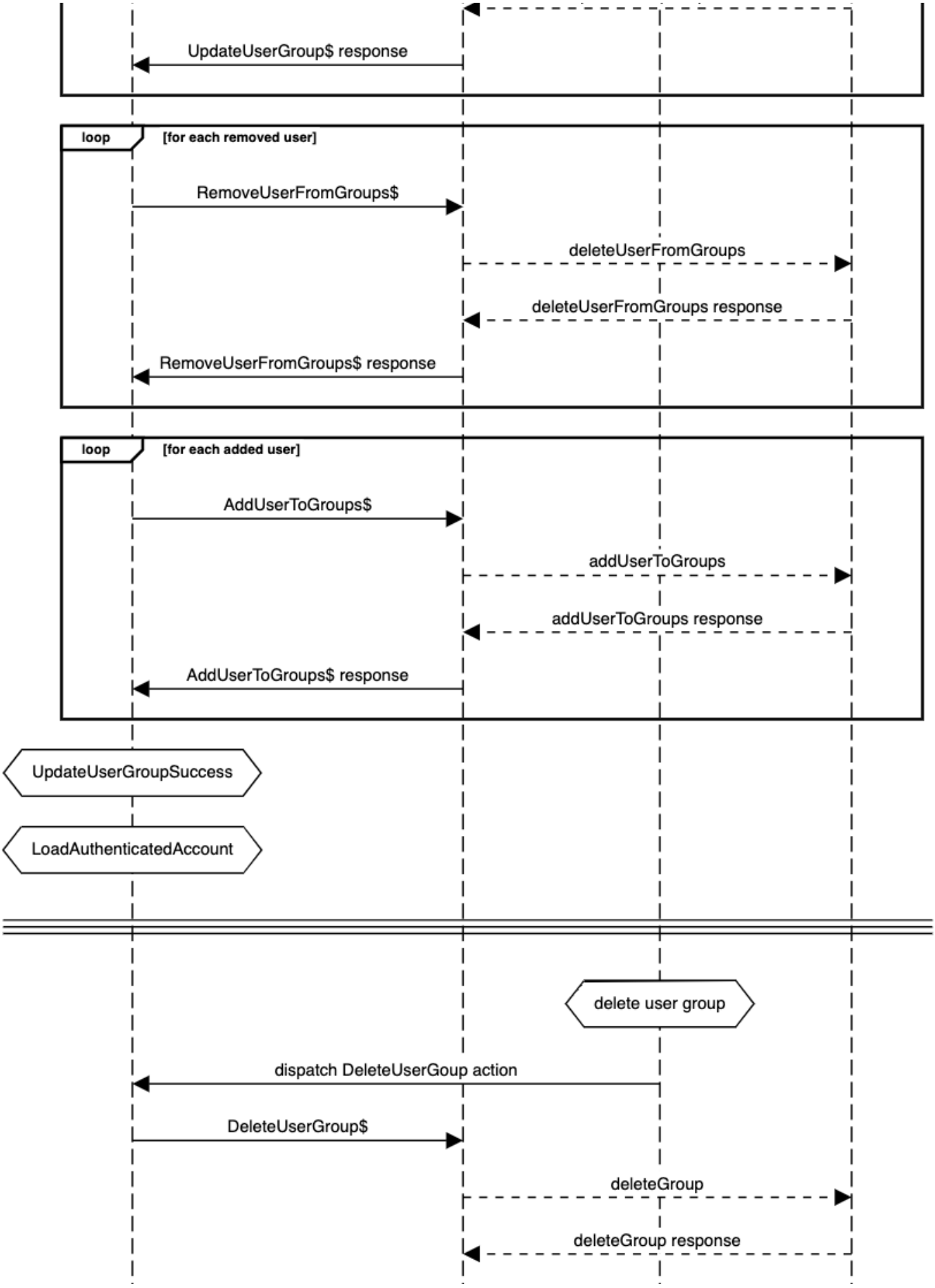
UI Preview

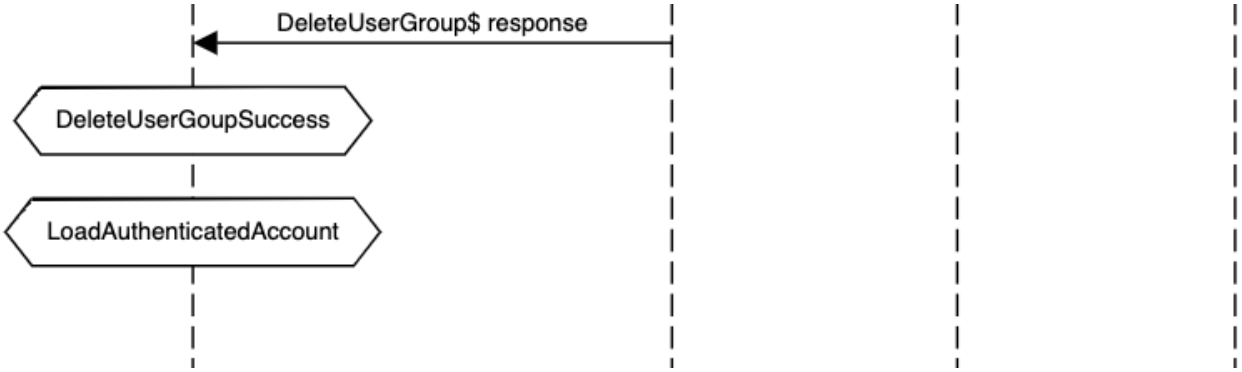


Sequence Flow




user actions
create new user group
dispatch CreateUserGroup action
CreateUserGroup$
createGroup
createGroup response
CreateUserGroup$ response
loop
[for each added user]
AddUserToGroups$
addUserToGroups
addUserToGroups response
AddUserToGroups$ response
CreateUserGroupSuccess
LoadAuthenticatedAccount
modify user group
dispatch UpdateUserGroup action
opt
[if permissions list or name changed]
UpdateUserGroup$
updateGroup
updateGroup response


UpdateUserGroup$ response
loop
[for each removed user]
RemoveUserFromGroups$
deleteUserFromGroups
deleteUserFromGroups response
RemoveUserFromGroups$ response
loop
[for each added user]
AddUserToGroups$
addUserToGroups
addUserToGroups response
AddUserToGroups$ response
UpdateUserGroupSuccess
LoadAuthenticatedAccount
delete user group
dispatch DeleteUserGoup action
DeleteUserGroup$
deleteGroup
deleteGroup response



Conversate App (Conversation Designer)

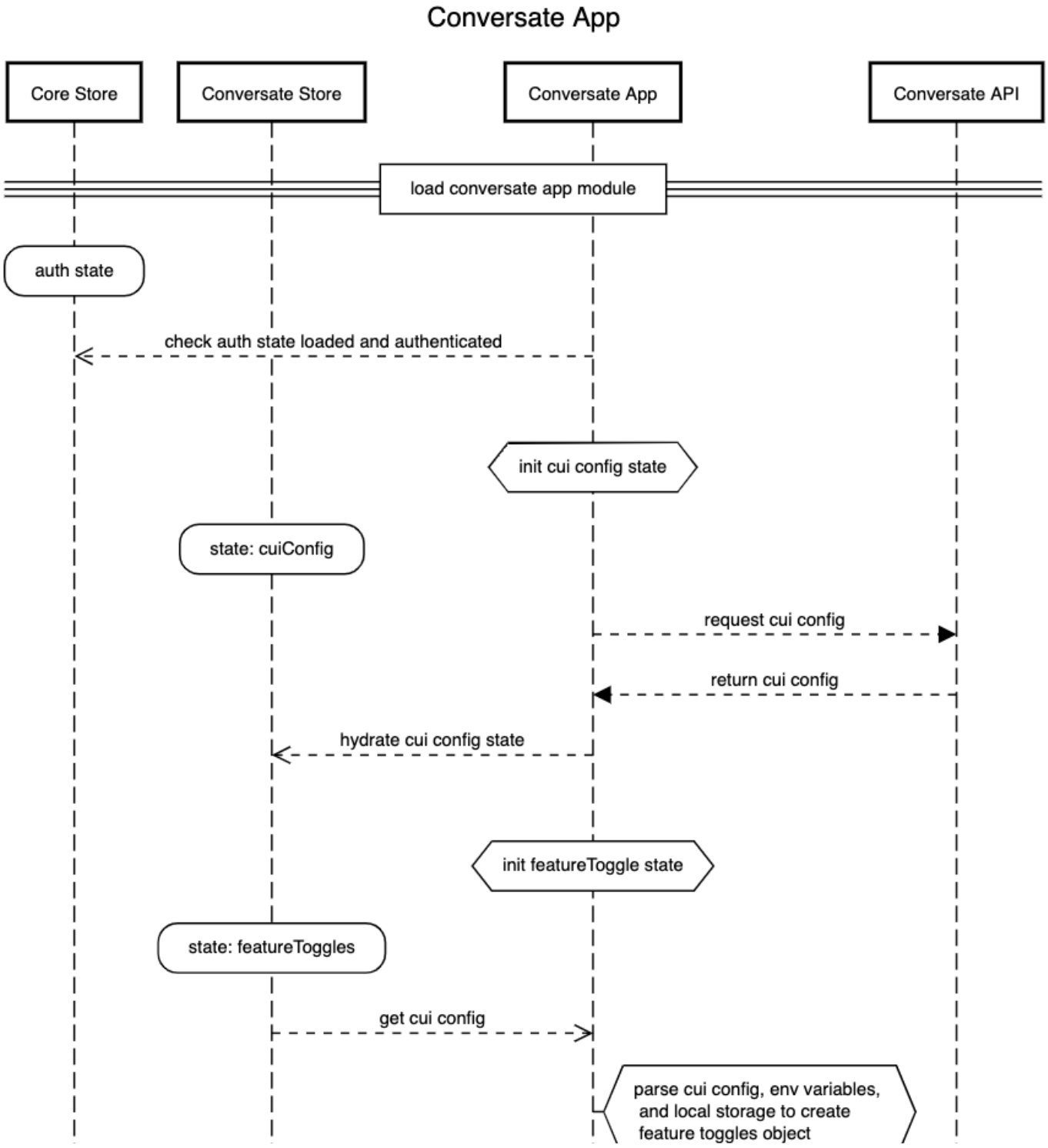
Overview

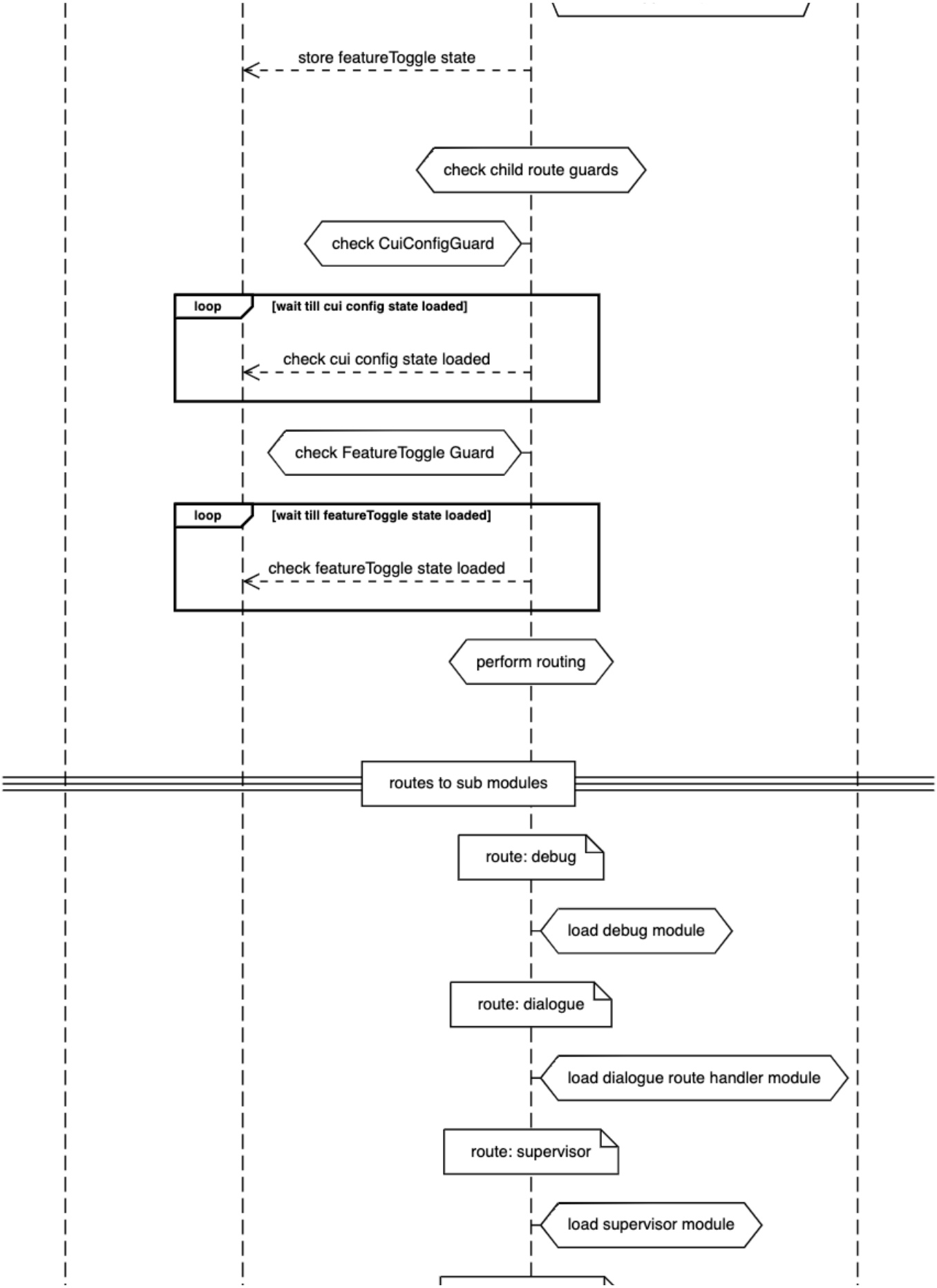
Conversate App module is a rather simple module on its own, whose main responsibility is to act as a proxy router to its sub modules. Conversate App's sub modules have many roles, and these sub modules are where users interact and configure all things related to the setting up the A.I. assistant's, their conversation flows, their enabled channels, and more.

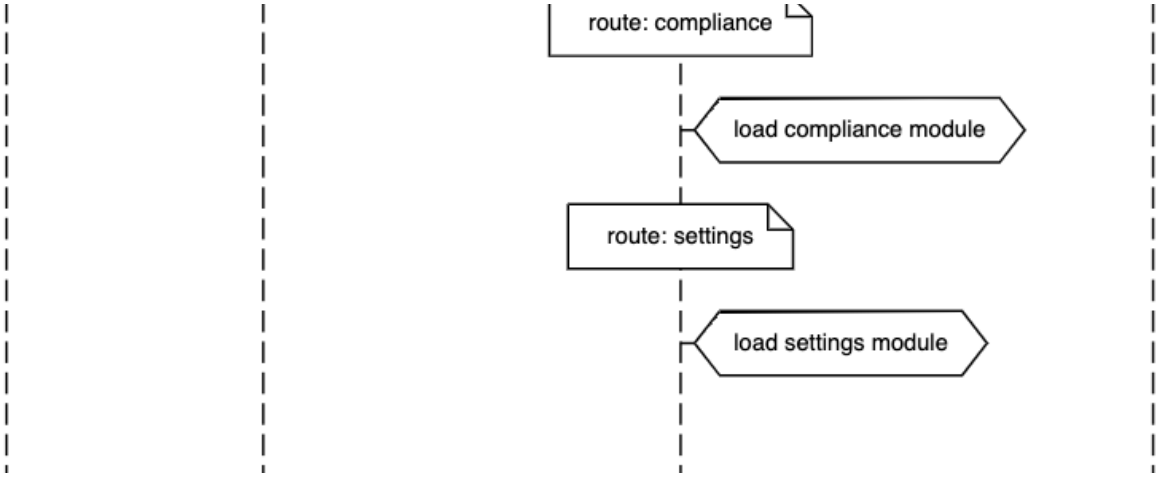
Table Of Contents

- Dialogue Explorer LINC (deprecated) — a visual node-based editor for conversation training data.
- Dialogue Route Handler — A simple module designed to handle a shared route routing to conditional modules.
- Feature Toggle — Module responsible for all things related to Feature Toggles
  - Feature Toggles Editor Component — A component to edit the current state of any registered feature toggle
  - Feature Toggles Service
- Integration Service — a viewless module containing models and singletons responsible for interacting with integration services
- Supervisor — The Supervisor module gives the ability to analyze conversations that users have had with the A.I. Agent. It houses a service, a guard, and UI views.
- Validator (Deprecated)

Initialization Flow and Routing




store featureToggle state
check child route guards
check CuiConfigGuard
loop
[wait till cui config state loaded]
check cui config state loaded
check FeatureToggle Guard
loop
[wait till featureToggle state loaded]
check featureToggle state loaded
perform routing
routes to sub modules
route: debug
load debug module
route: dialogue
load dialogue route handler module
route: supervisor
load supervisor module



Conversate Sub Modules

Conversate App is broken up into a few sub modules.

Modules

Agent Settings

Provides views to allow users to manage Assistant Settings

Compliance

Compliance module lists updates to the assistant's dialogue flow (such as intent creation/modification) and settings.

CUI Config

A viewless module responsible for housing common logic for cui config management.

Debug

Debug module is a hidden module within the conversate app. We do not link to this route from anywhere, users must know the route exists, and manually route there via url. It can be used to set/reset feature flags.

Dialogue Explorer

Dialogue Explorer is where conversation designers (CDs) manage and test an A.I. Assistants conversation flow.

Dialogue Explorer LINC (deprecated)

a visual node-based editor for conversation training data.

Dialogue Route Handler

A simple module designed to handle a shared route routing to conditional modules.

Feature Toggle

Module responsible for all things related to Feature Toggles

Integration Service

a viewless module containing models and singletons responsible for interacting with integration services

Supervisor

The Supervisor module gives the ability to analyze conversations that users have had with the A.I. Agent. It houses a service, a guard, and UI views.

Validator (Deprecated)

Agent Settings

Overview

The settings module allows users to configure assistant settings, such as enabled channels, campaigns, integration service connections, Human Handoff, API Access, Authentication, general dialogue settings, and general assistant settings.

Agent Settings Service

The Agent Settings Service is a singleton object, used by the Agent Settings State's action effects, providing methods that allow for interaction with Conversate api via the Socket Service.

Agent Settings Guard

The `AgentSettingsGuard` is a singleton object that provides a route guard to be used in routing modules. It uses the canActivate interface, and performs an async check. If this guard is triggered, and the Agent Settings State is not hydrated, or in the process of hydrating, the guard will dispatch the LoadAWCAgentConfig and LoadChannelSettings actions. It will then wait for an update to the state, and then proceed. The result is the ability to apply this guard to a route, and that route will not activate until we know that the state has been hydrated with data.

Views

Agent Settings page is broken up into multiple views, each dedicated to a certain segment of settings.
General Assistant Settings

This view is designed to hold basic settings for the assistant. At this time, it only lets users set the Assistants time zone. These settings are stored in the AWC Agent Config
Channel Settings

Channel Settings view allows users to enable/disable channels as well as manage each channels configuration. Each channel may require different set up configuration, and a form will be generated to support the individual channels needs.
Dialogue Settings

The Dialogue Settings view lets users perform actions around the Assistant's dialogue flow including setting thresholds for confidence during token matching, and setting global variables.
Integration Services Management

The Integration Services management page allows CUI users to add and edit configured Integration Services.
Handoff Settings

The handoff setting page is a view that allows user to configure or remove Human Handoff Providers.
Campaign Manager

A view to manage campaigns for the assistant.
Test User Settings

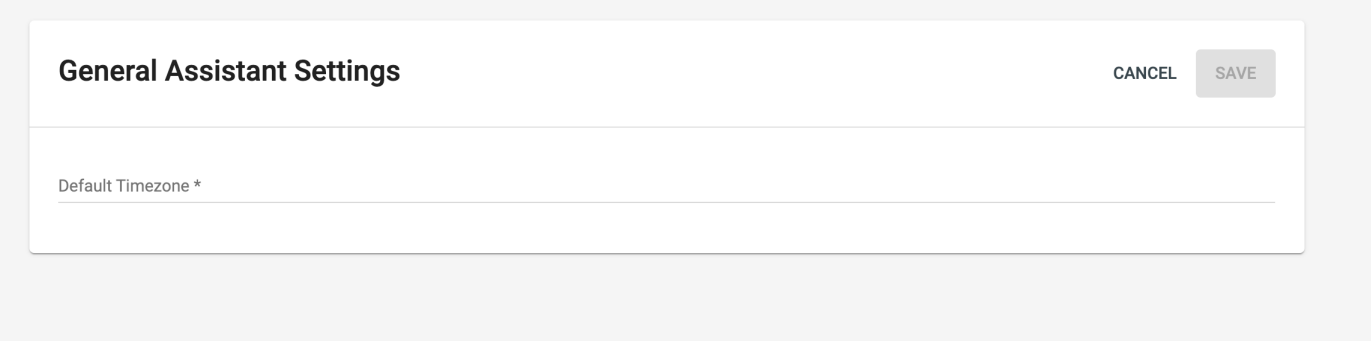
Configure the test user to be used for the Dialogue Explorer's simulator
Authentication Settings

Configure the authentication settings for user sign-in.
API Keys

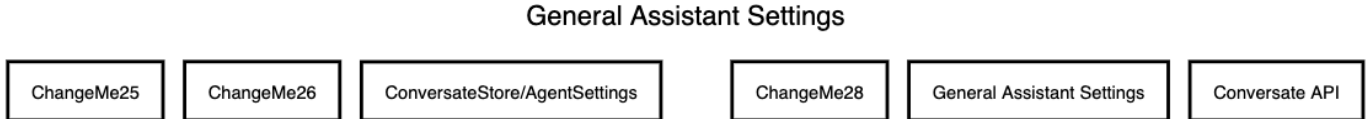
View and manage the assistant's API keys

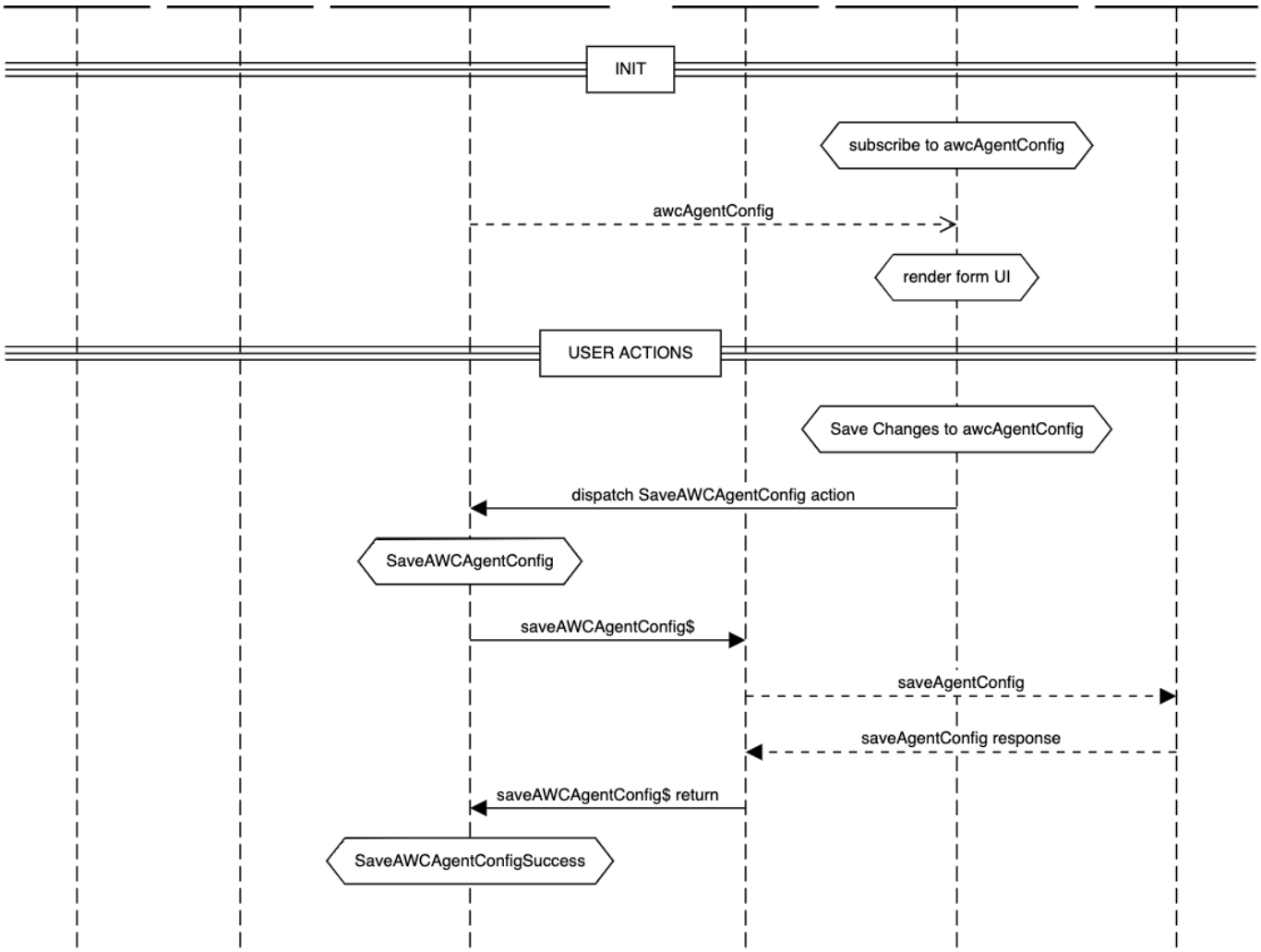
General Assistant Settings

OverviewThis view is designed to hold basic settings for the assistant. At this time, it only lets users set the Assistants time zone. These settings are stored in the AWC Agent ConfigScreenshots



Sequence Flow



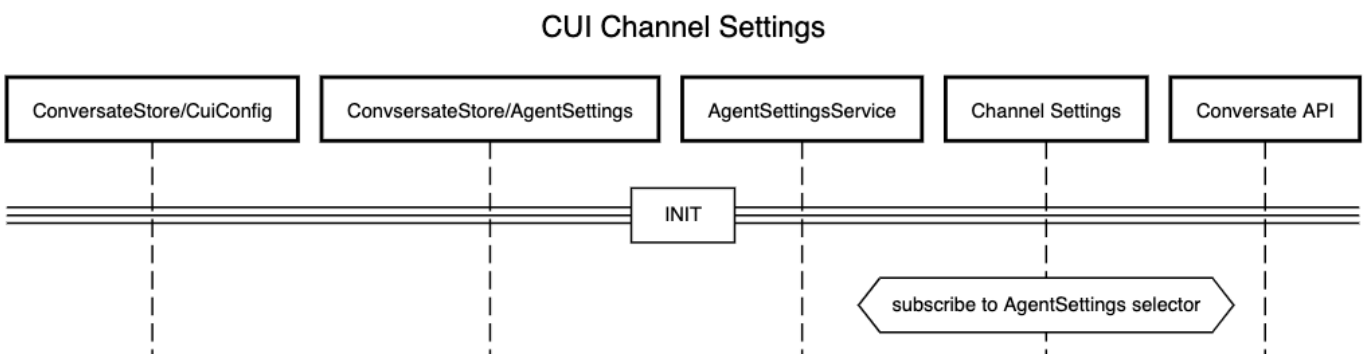


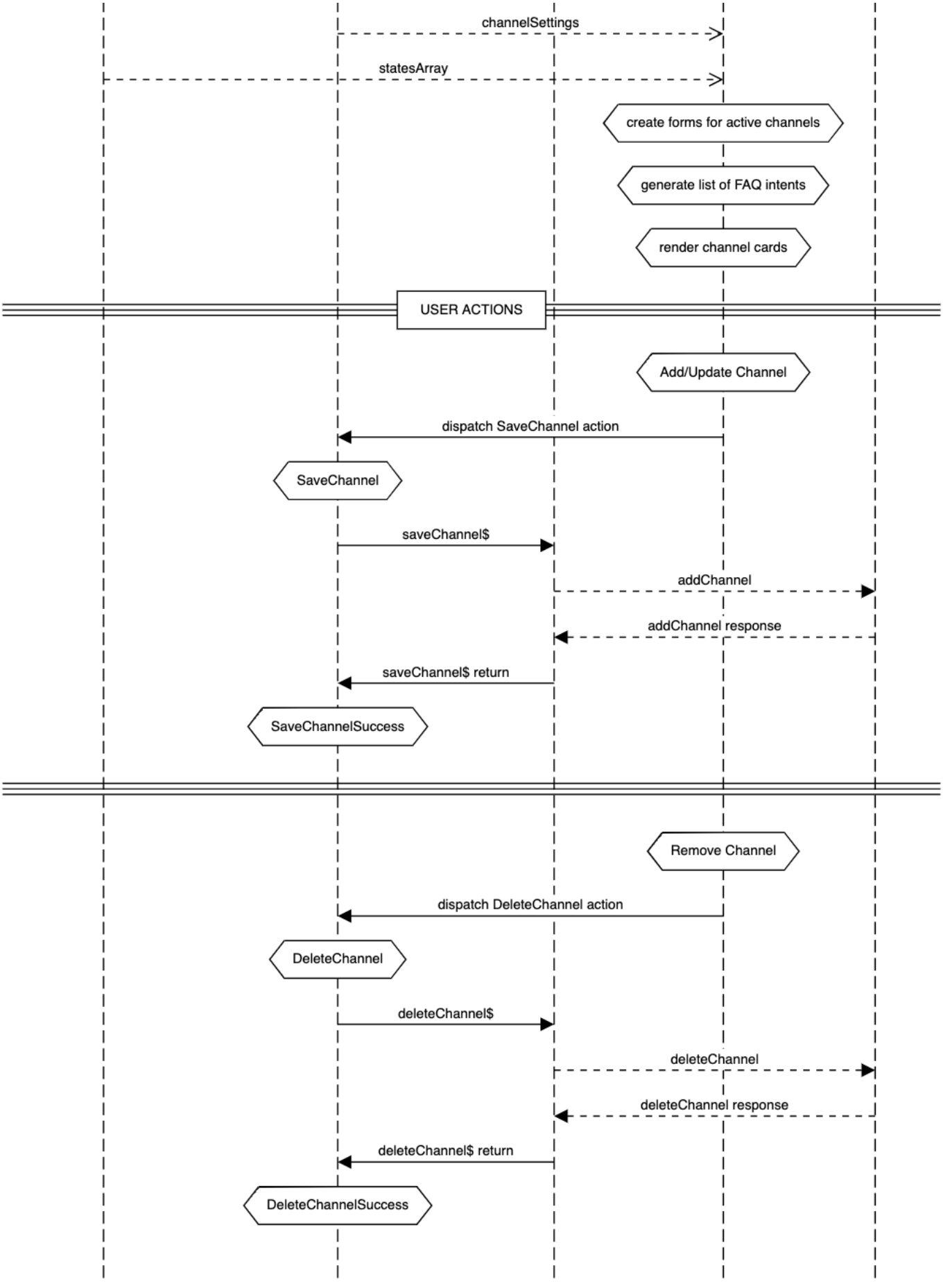
Channel Settings

OverviewChannel Settings view allows users to enable/disable channels as well as manage each channels configuration. Each channel may require different set up configuration, and a form will be generated to support the individual channels needs.Channels

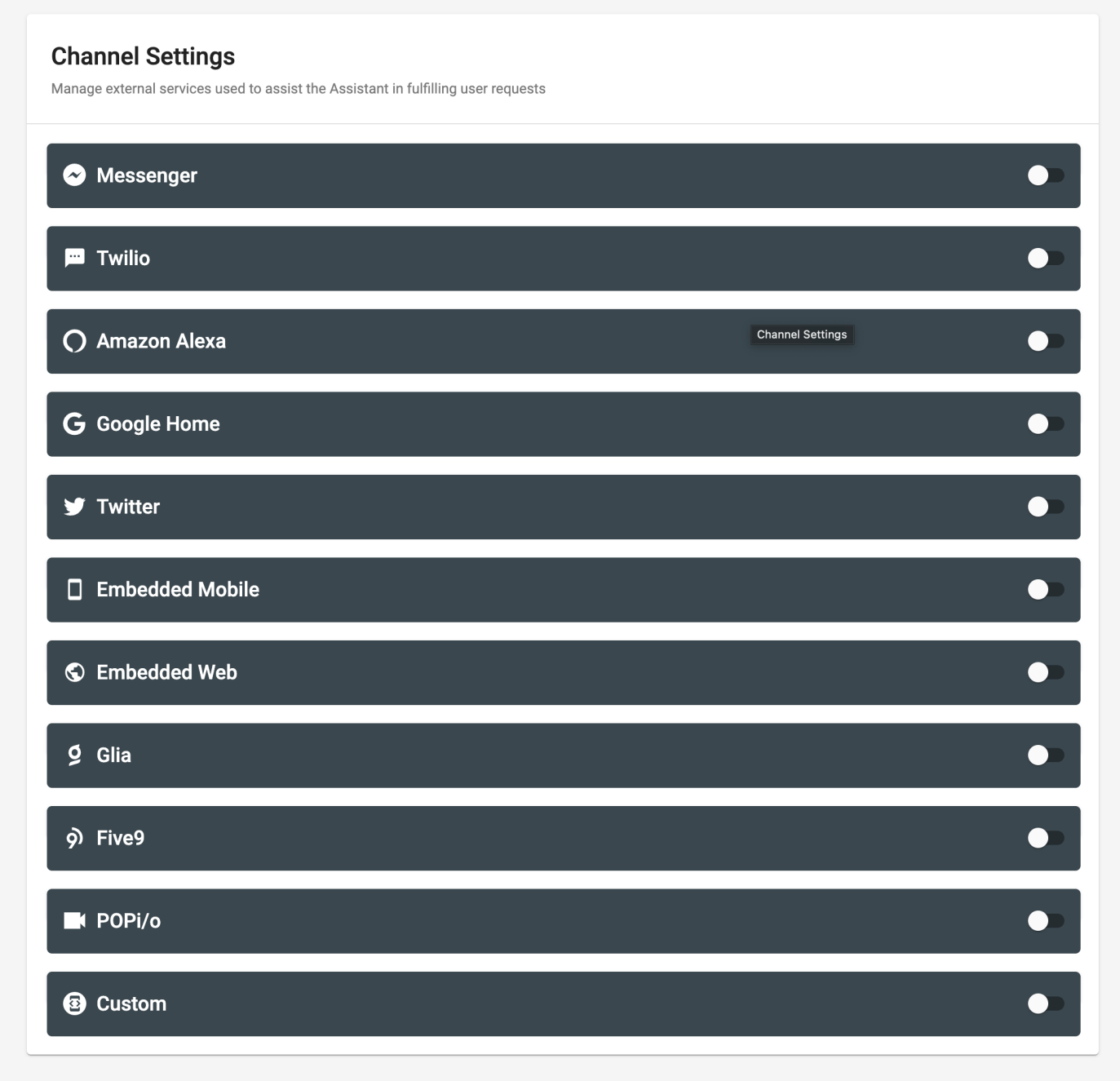
- Twilio - for SMS clients
- Facebook Messenger
- Amazon Alexa
- Google Home
- Twitter
- Embedded Mobile
- Embedded Web
- Glia
- Five9
- POPi/o
- Custom

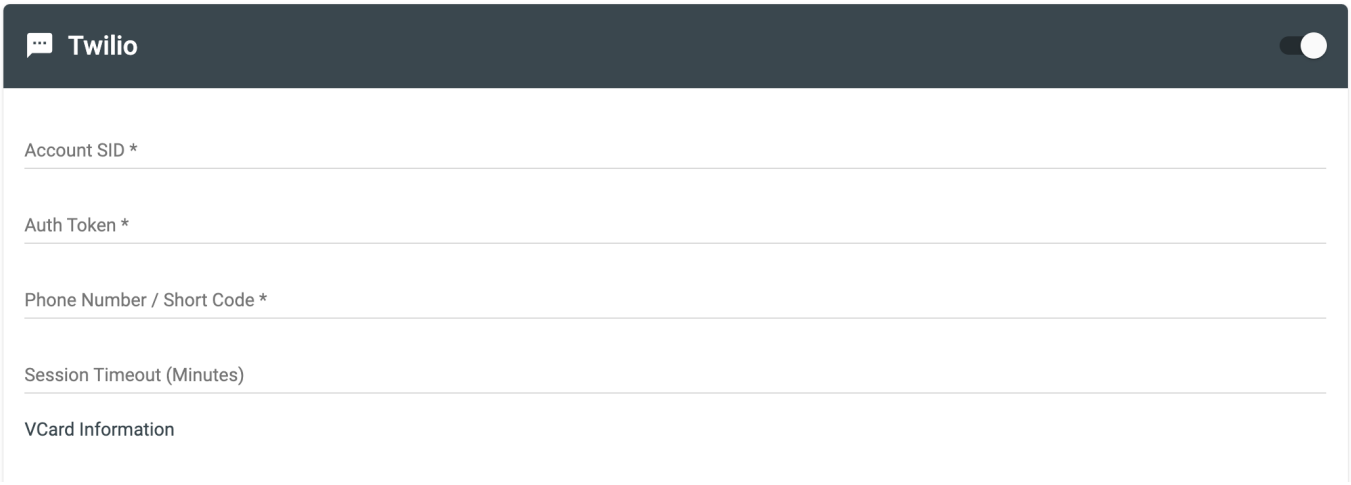
Sequence Flow




channelSettings
statesArray
create forms for active channels
generate list of FAQ intents
render channel cards
USER ACTIONS
Add/Update Channel
dispatch SaveChannel action
SaveChannel
saveChannel$
addChannel
addChannel response
saveChannel$ return
SaveChannelSuccess
Remove Channel
dispatch DeleteChannel action
DeleteChannel
deleteChannel$
deleteChannel
deleteChannel response
deleteChannel$ return
DeleteChannelSuccess

Screenshots





Name

Organization

E-Mail

Photo URL

Website URL

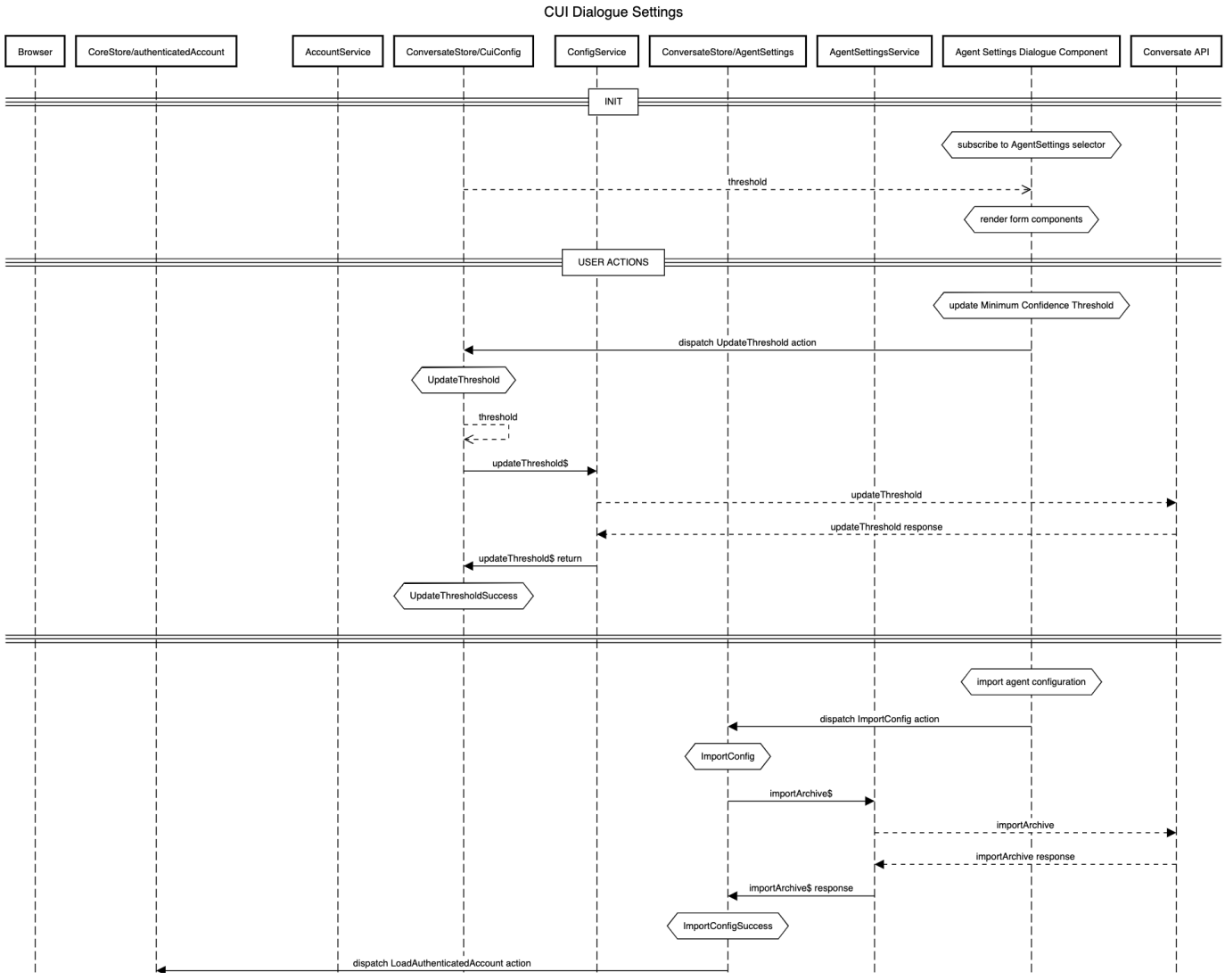
CANCEL SAVE

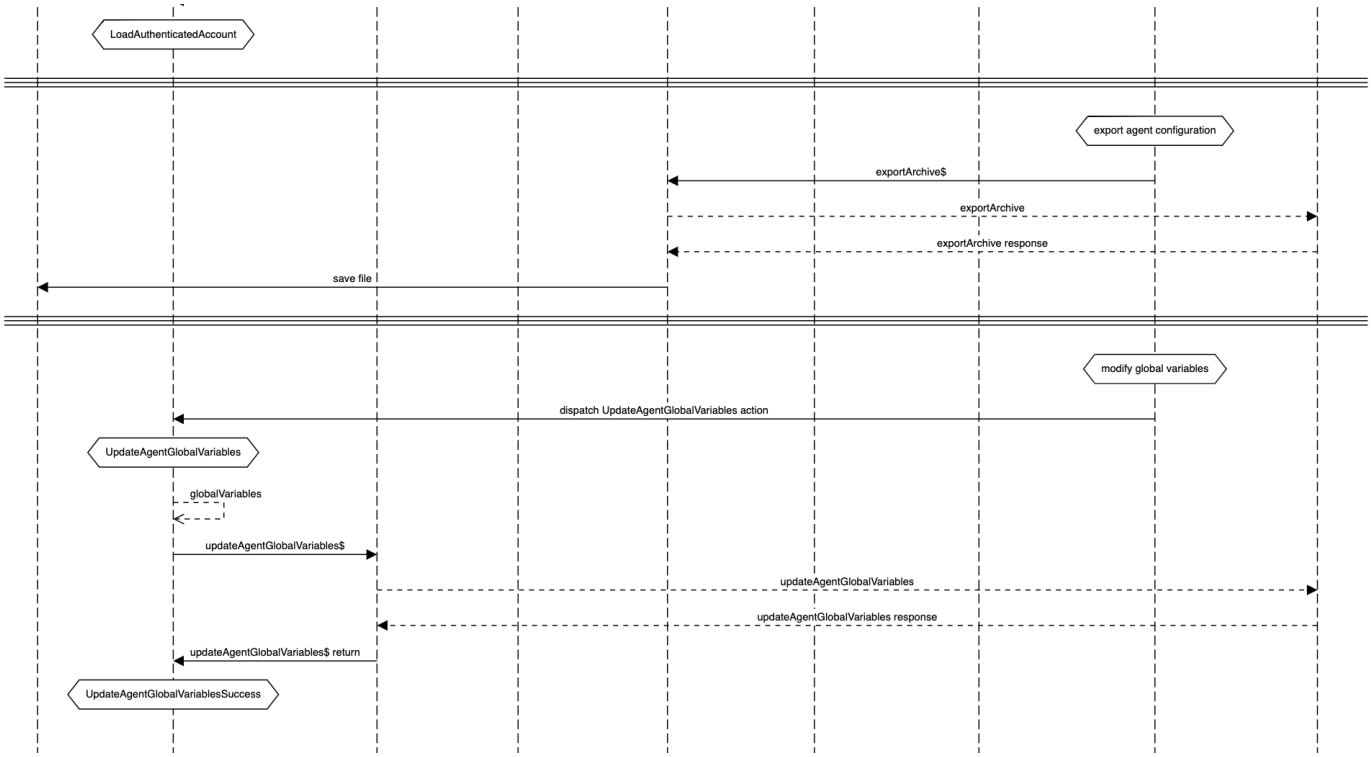
Dialogue Settings

OverviewThe Dialogue Settings view lets users perform actions around the Assistant’s dialogue flow including setting thresholds for confidence during token matching, and setting global variables.Features

- Minimum Confidence Threshold - Set the threshold for minimum intent confidence. If the prediction confidence meets or exceeds this value, the corresponding intent is used; otherwise the default NONE intent is triggered.
- Assistant Configuration Importing/Exporting
- Global Variables - special configurable variables that can be utilized in utterances and response templates

Sequence Flow

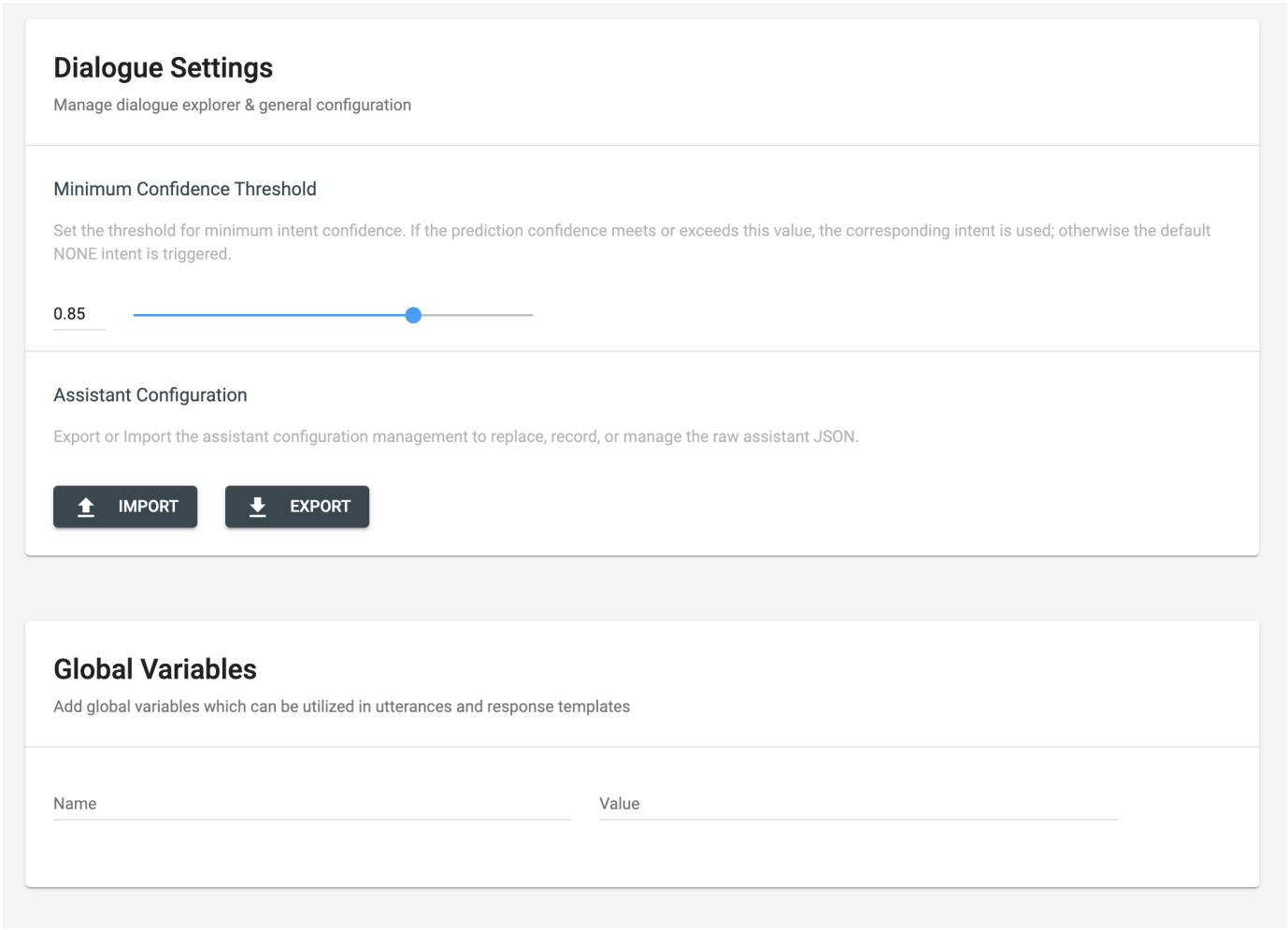




Sequence Flow References

- LoadAuthenticatedAccount

Screenshots



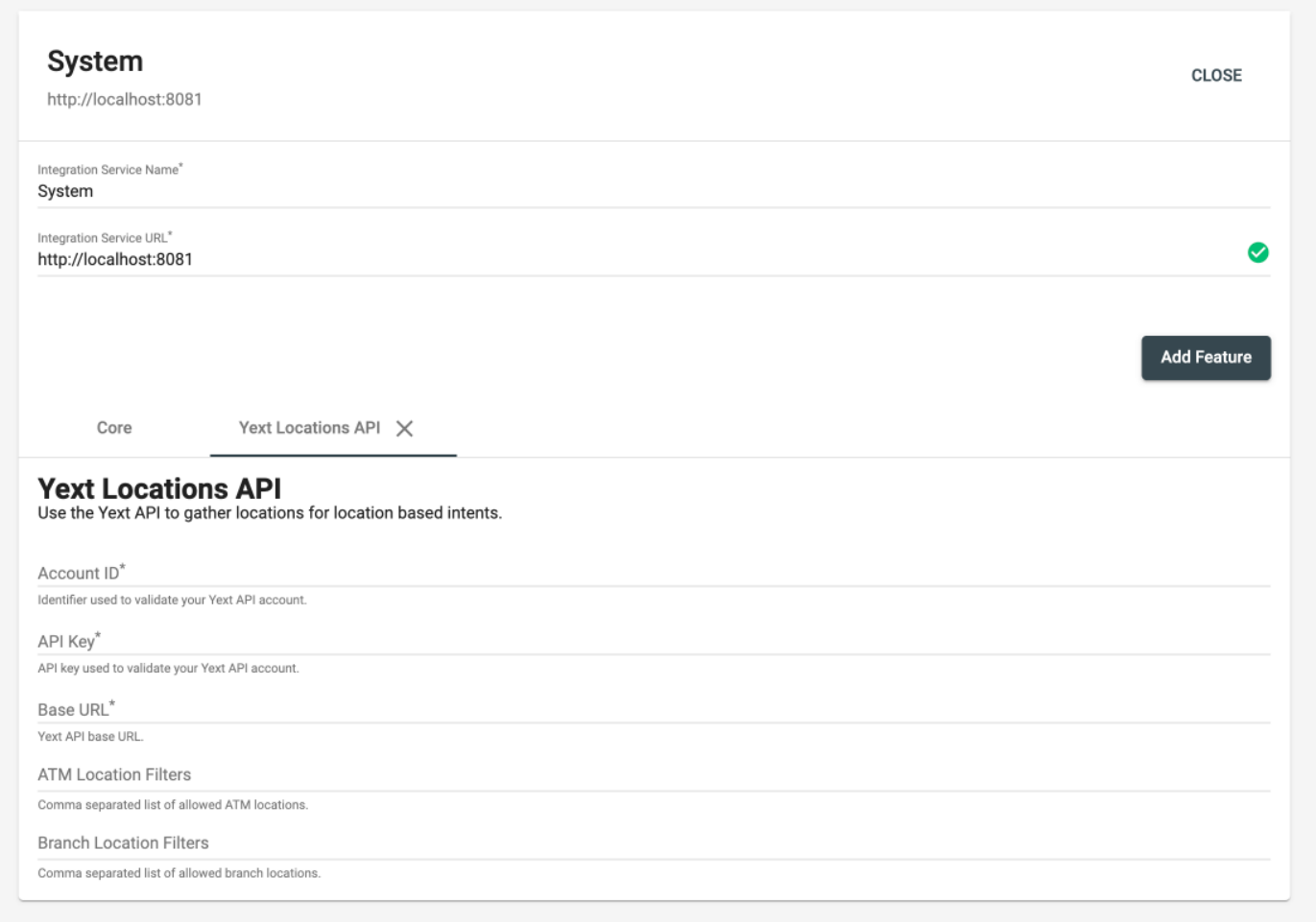
Integration Services Management

OverviewThe Integration Services management page allows CUI users to add and edit configured Integration Services. Integration Service Config Schema

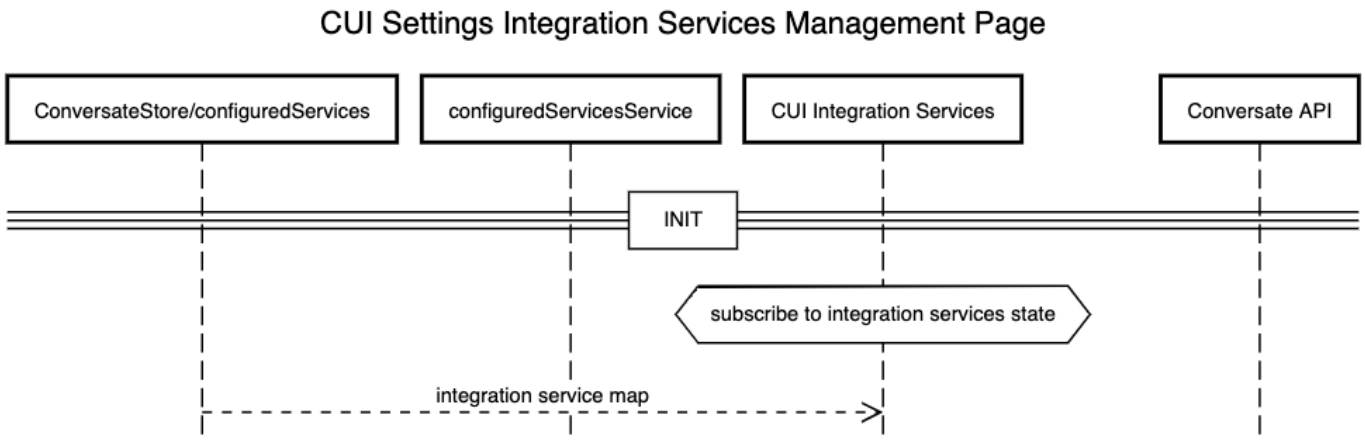
Integration services can be connected to multiple Assistants, but may need configuration in order to work properly. In order to move away from hardcoded environment variables, we have developed a system to allow the service to define its own form that needs to be filled when being registered for use via it's own `Config Schema`. A config schema is a collection of JSONSchema objects. Generally, services will put required configuration values on the `Core` form. A integration service form, may have additional `feature` configurations that are able to be added on, these are also defined in the config schema. These dynamic forms are generated real time in CUI through use of a custom-built JSON schema recursive parser. The parser takes the schema, and converts it into `FormGroups` from Angular's built in form management tools.
Dependencies

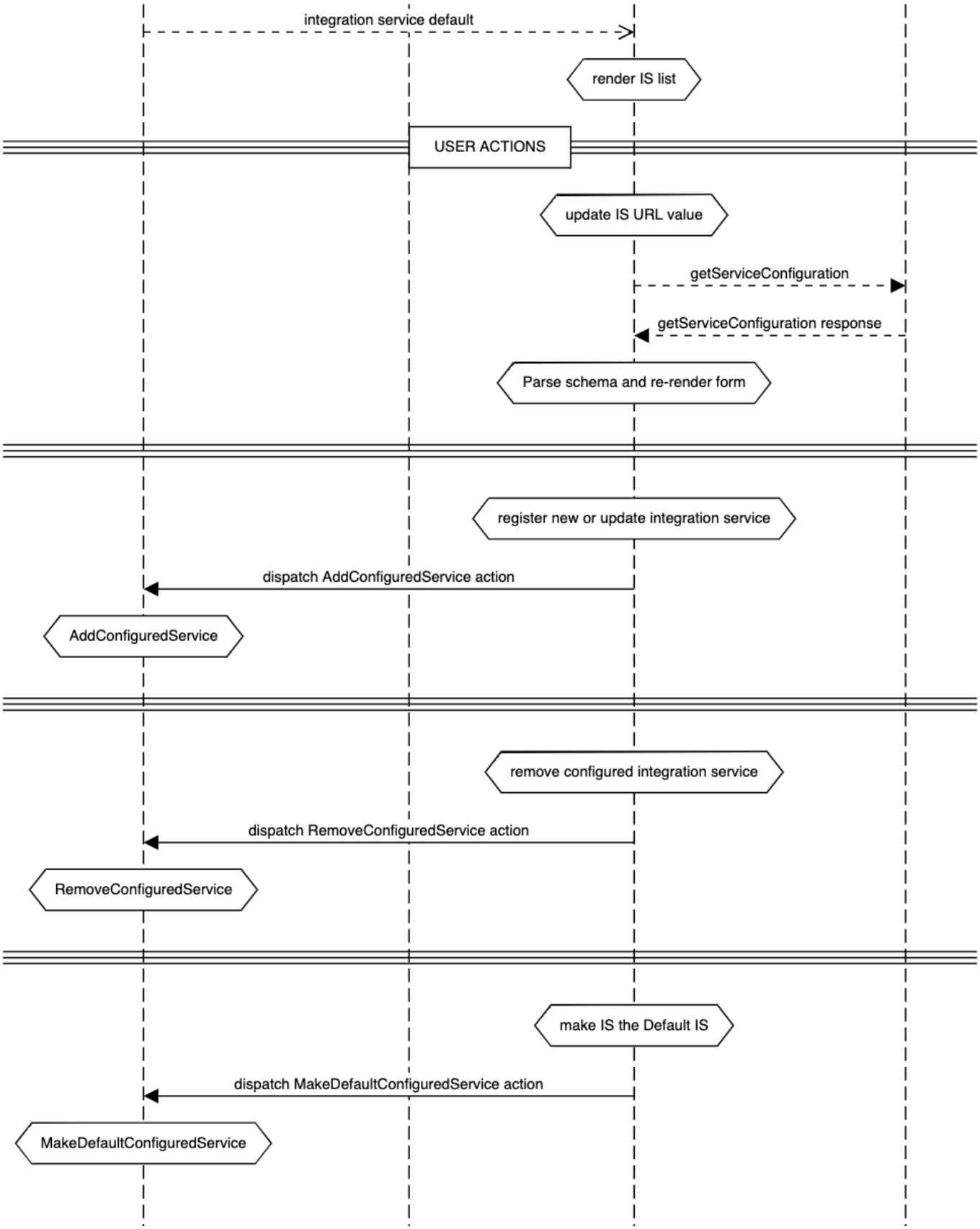
- Integration Service State
- ConfiguredServicesService

UI Preview



Sequence Flow





Handoff Settings

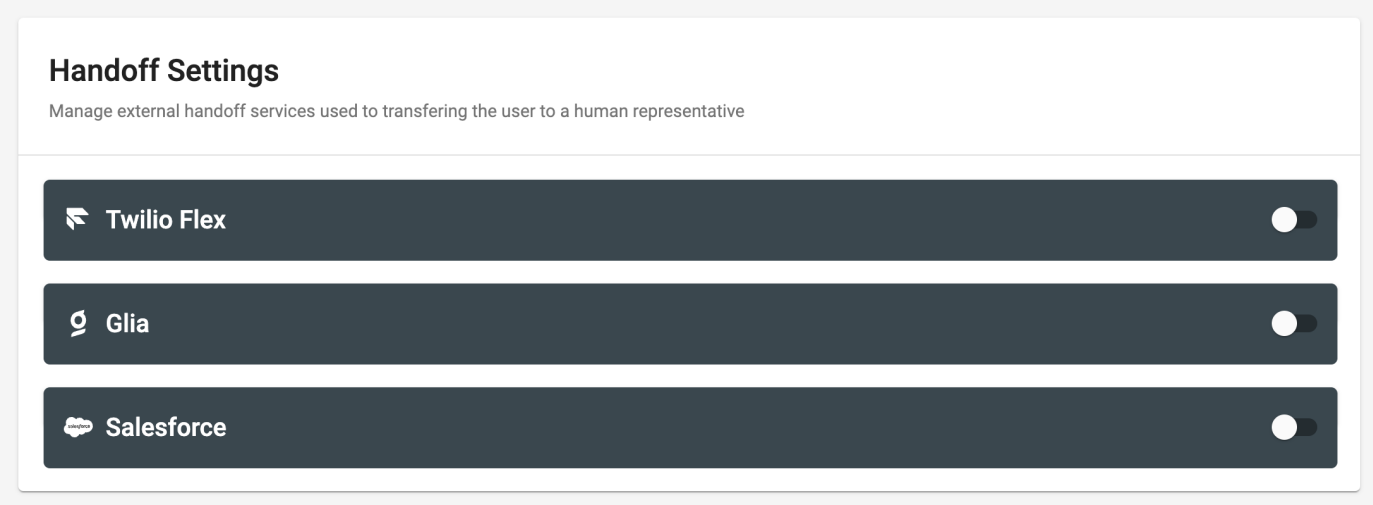
Overview

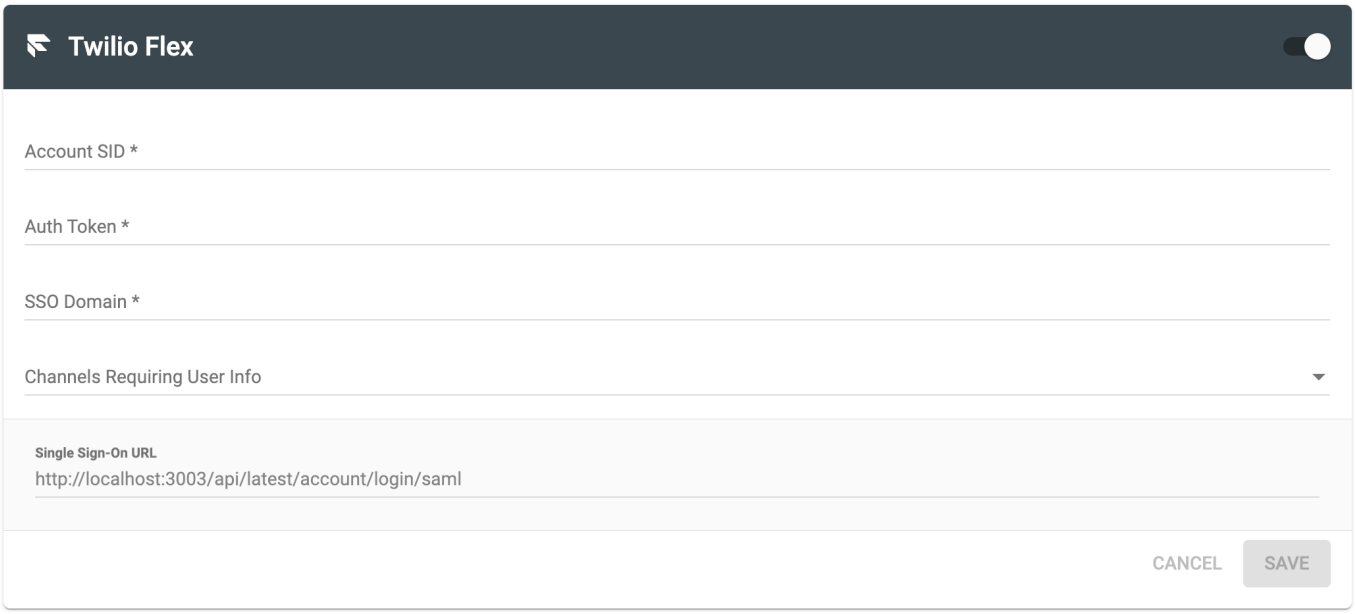
The handoff setting page is a view that allows user to configure or remove Human Handoff Providers. Only one human handoff provider can be configured per Assistant.
Providers

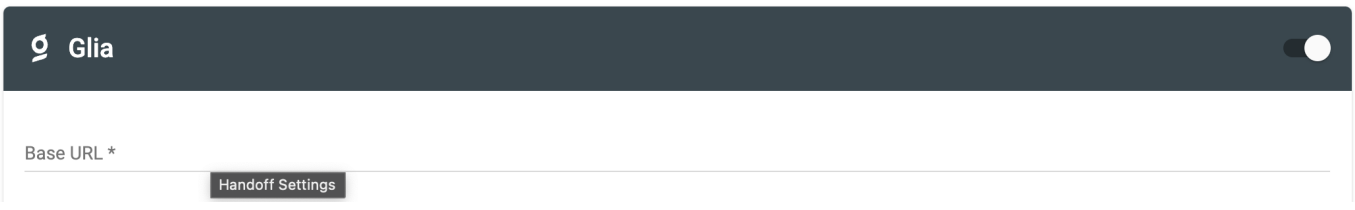
- Salesforce
  The Salesforce integration is a long polling driven architecture. In the Main Loop the Salesforce API is leveraged to pull new messages, and update the Salesforce session state. For instance, flagging the user as Idle or Active.
- Glia
  The Glia integration is a webhook driven architecture that requires passing an endpoint (the Channel-Connector service) when initializing a new human-handoff session with Glia.

  Rather than polling for new messages on every Main Loop iteration, new message events are passed to the provider backend via subscription to the Platform Event Bus.
- Flex (Contact Center Provider)
  The Flex integration leverages the Twilio-chat suite. Webhooks are used to read message events similarly to the Glia implementation, an endpoint to Channel-Connector is given to Twilio and messages are handled through the Platform Event Bus.
  Unlike the Glia implementation, the Twilio-chat suite leverages a web socket connection with event listeners for events. Users entering and exiting the chat, and the typing indicator are handled this way.

Screenshots







Site API Key ID *

Site API Key Secret *

Site ID *

User API Key ID *

User API Key Secret *

Channels Requiring User Info

CANCEL SAVE

Salesforce

Base URL *

Business Hours Name *

Deployment ID *

Organization ID *

REST API - Base URL *

REST API - Version *

SOAP API - Base URL *

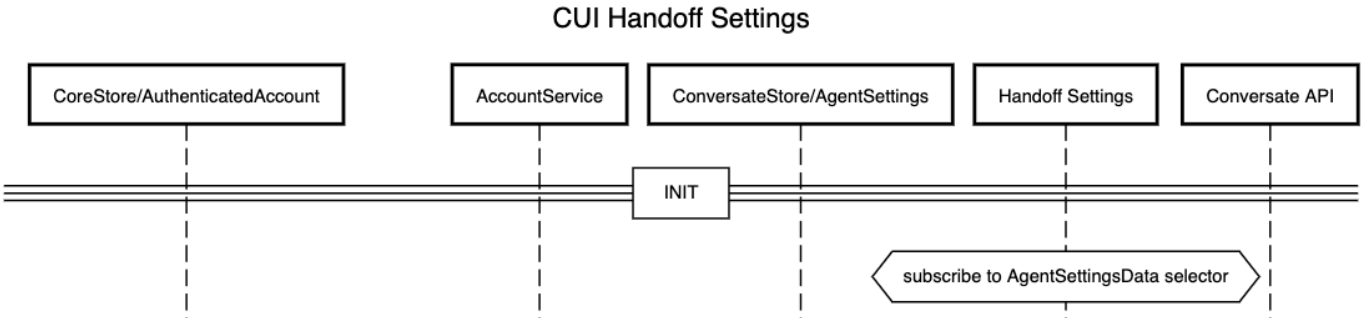
SOAP API - Username *

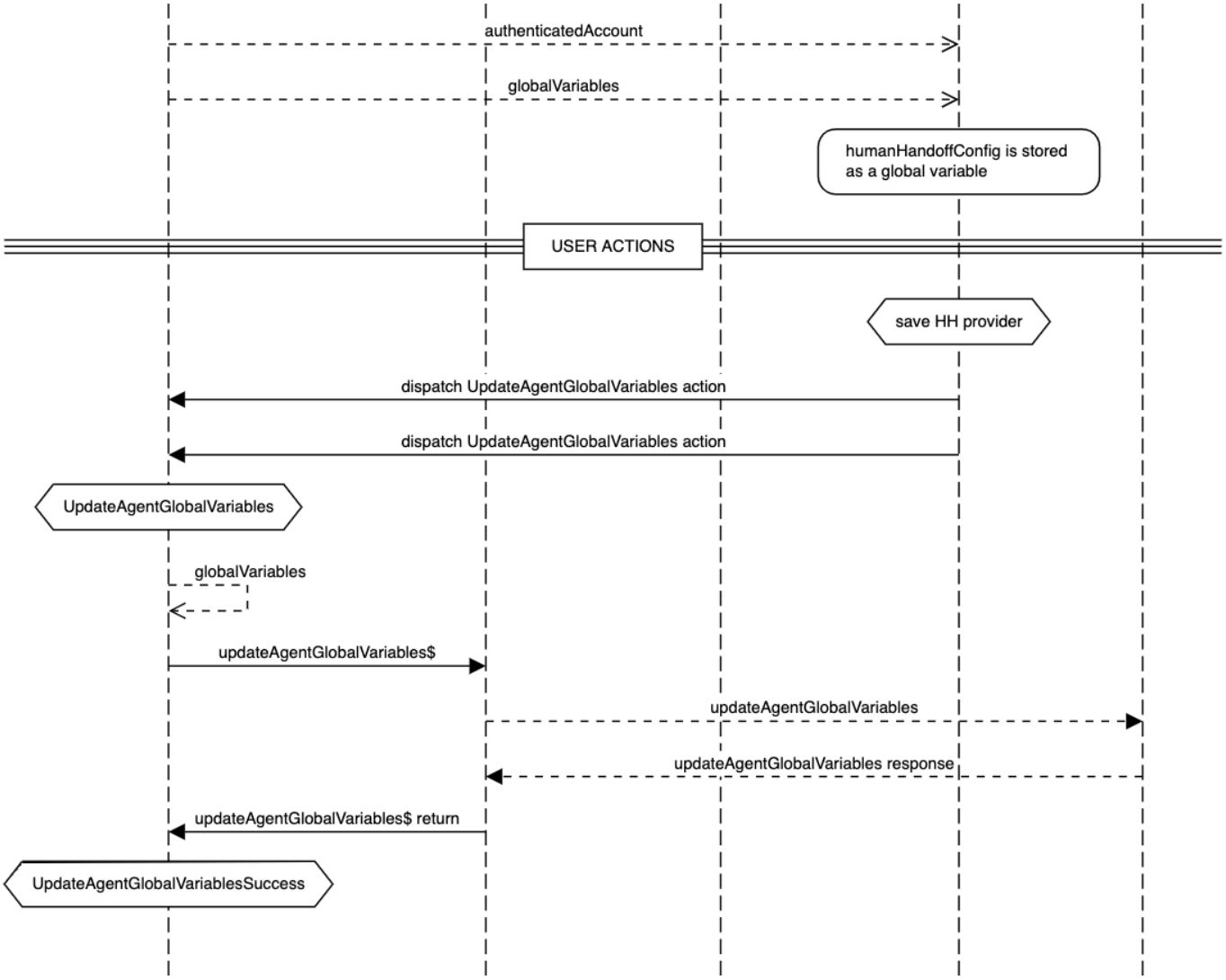
SOAP API - Password *

Channels Requiring User Info

CANCEL SAVE

Sequence Flow



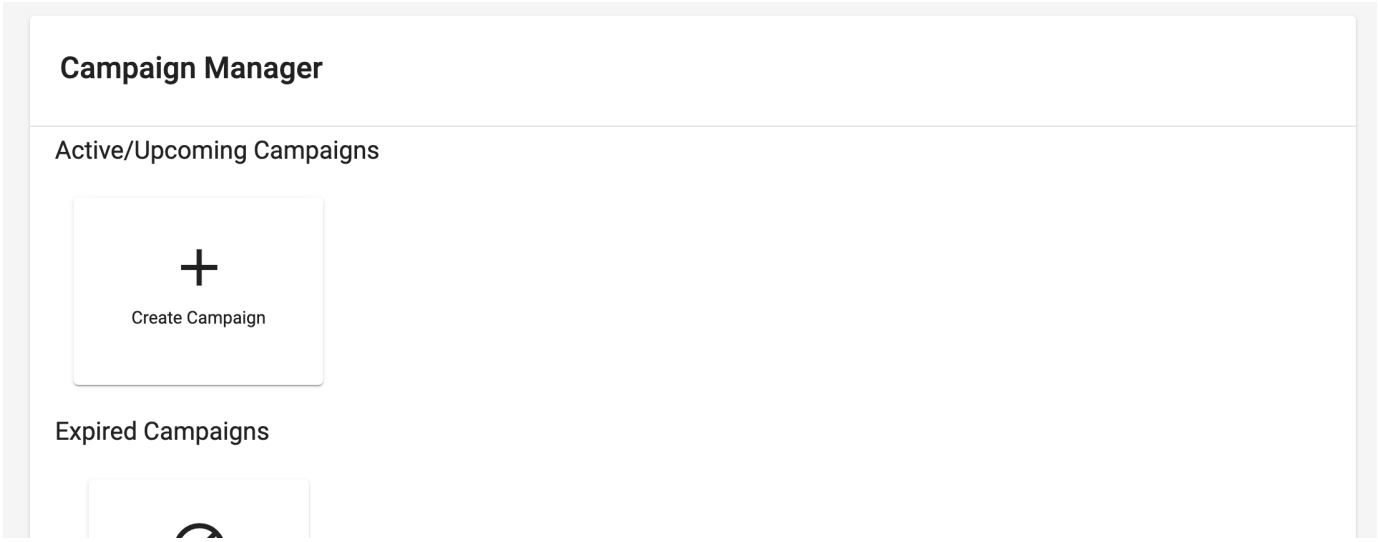


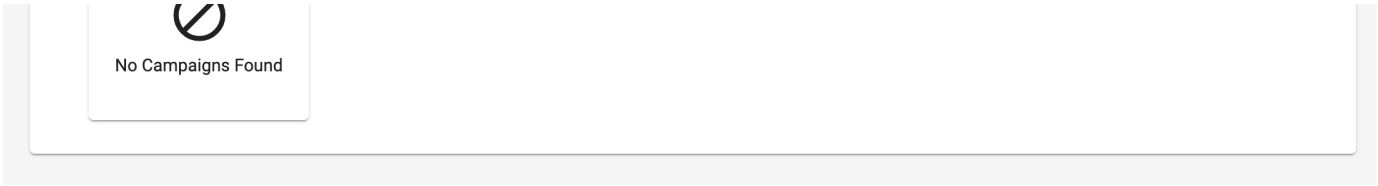
Campaign Manager

OverviewA view to manage campaigns for the assistant.

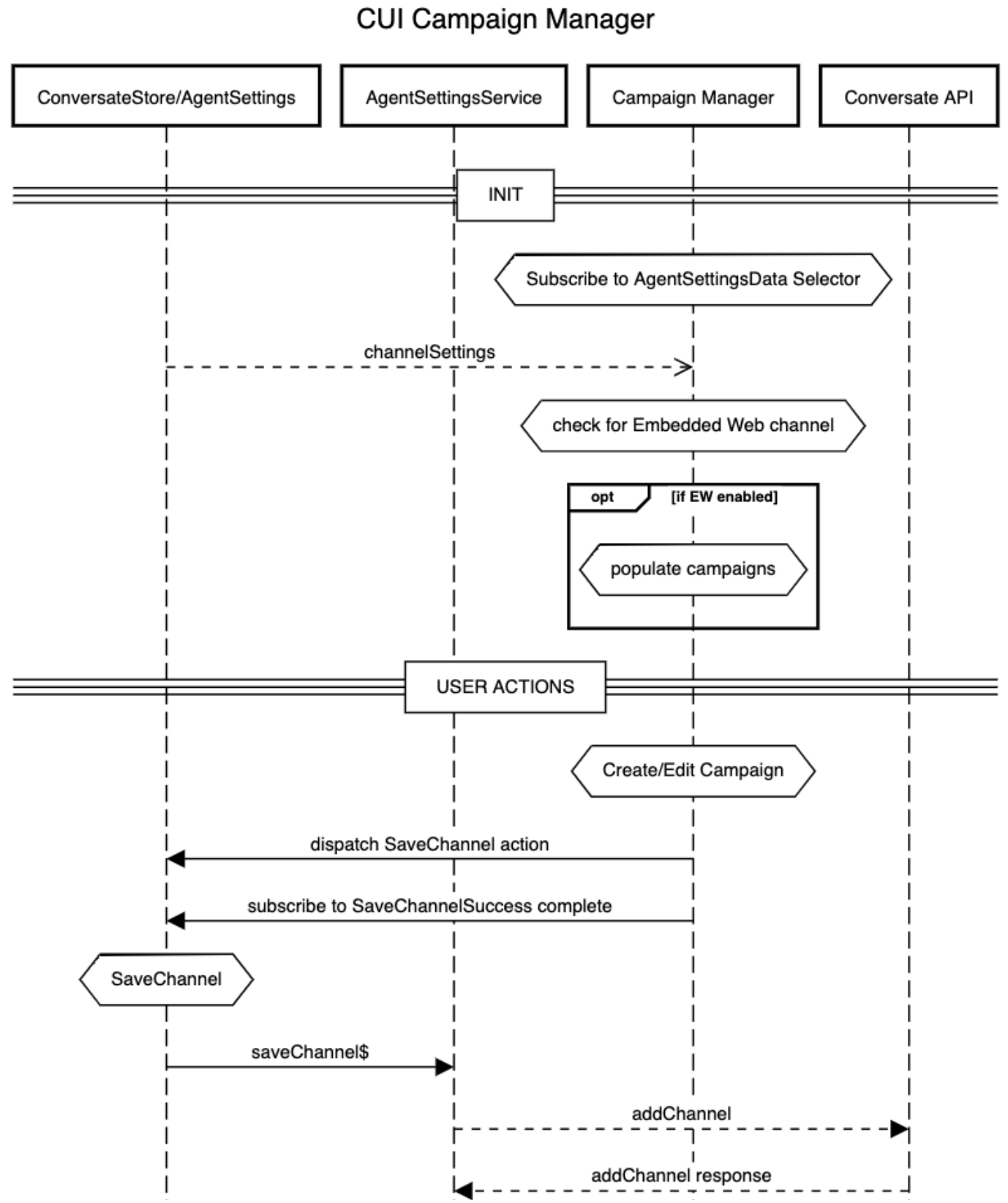
Campaigns are a feature we enable for Assistant's that utilize the Embedded Web client. Conversation Designers can set up campaigns for their assistant, so that the widget will auto trigger the CTA experience based on condition matching. Conditions are the url to trigger on, and a valid date range. Read more about Embedded Web Campaigns.

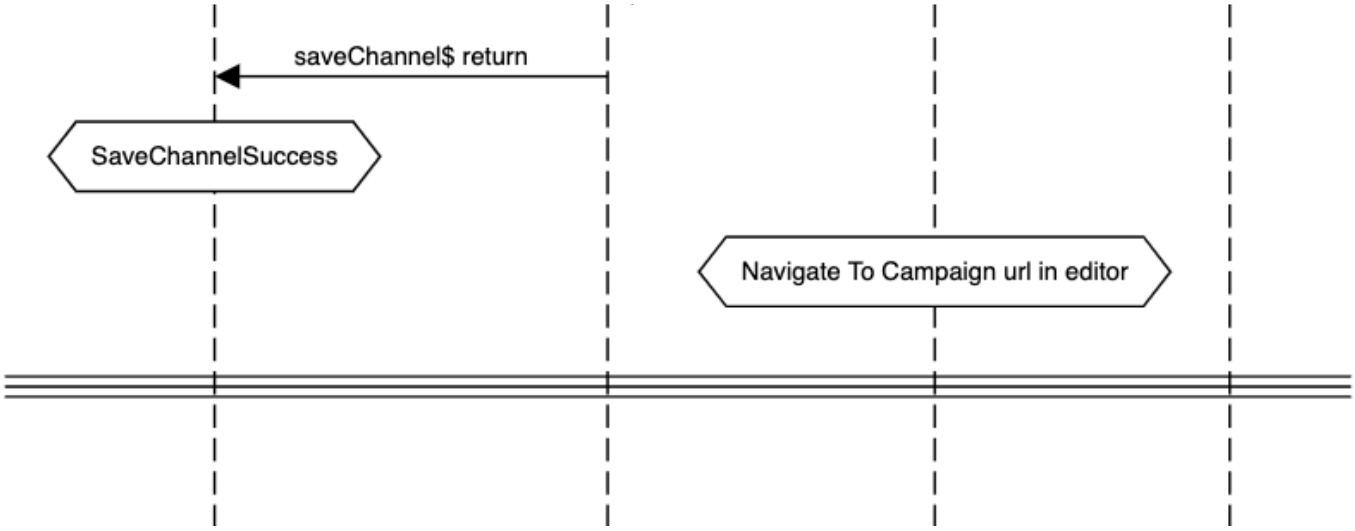
Screenshot





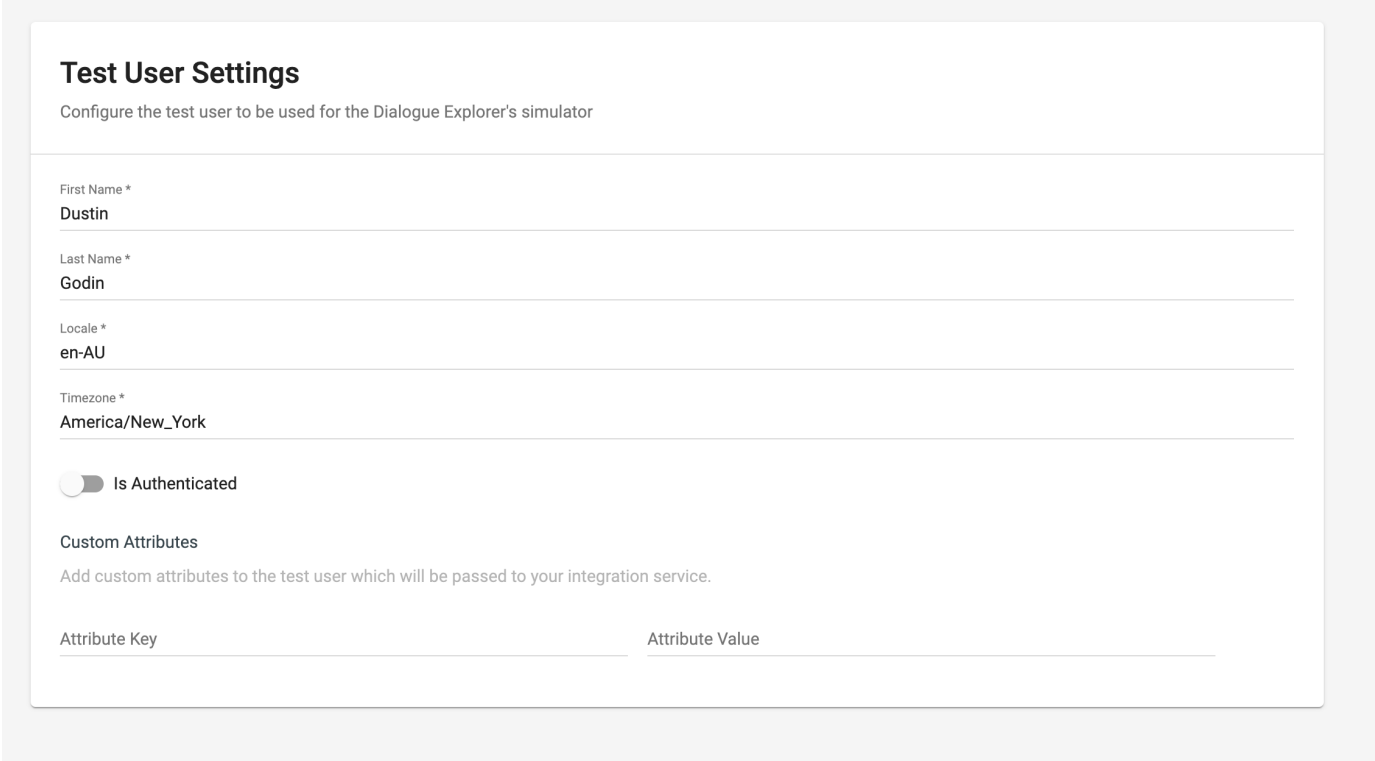
Sequence Flow



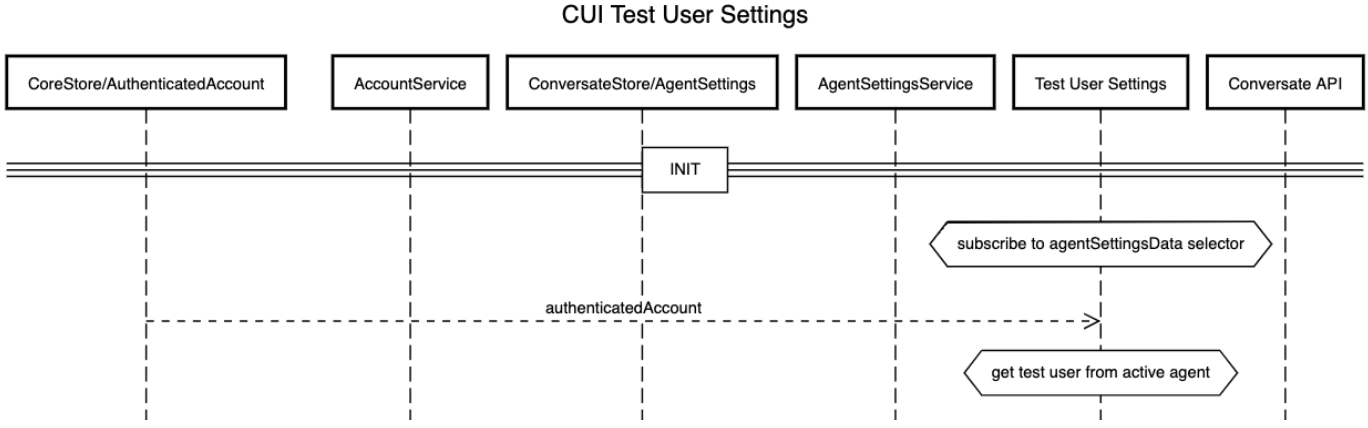


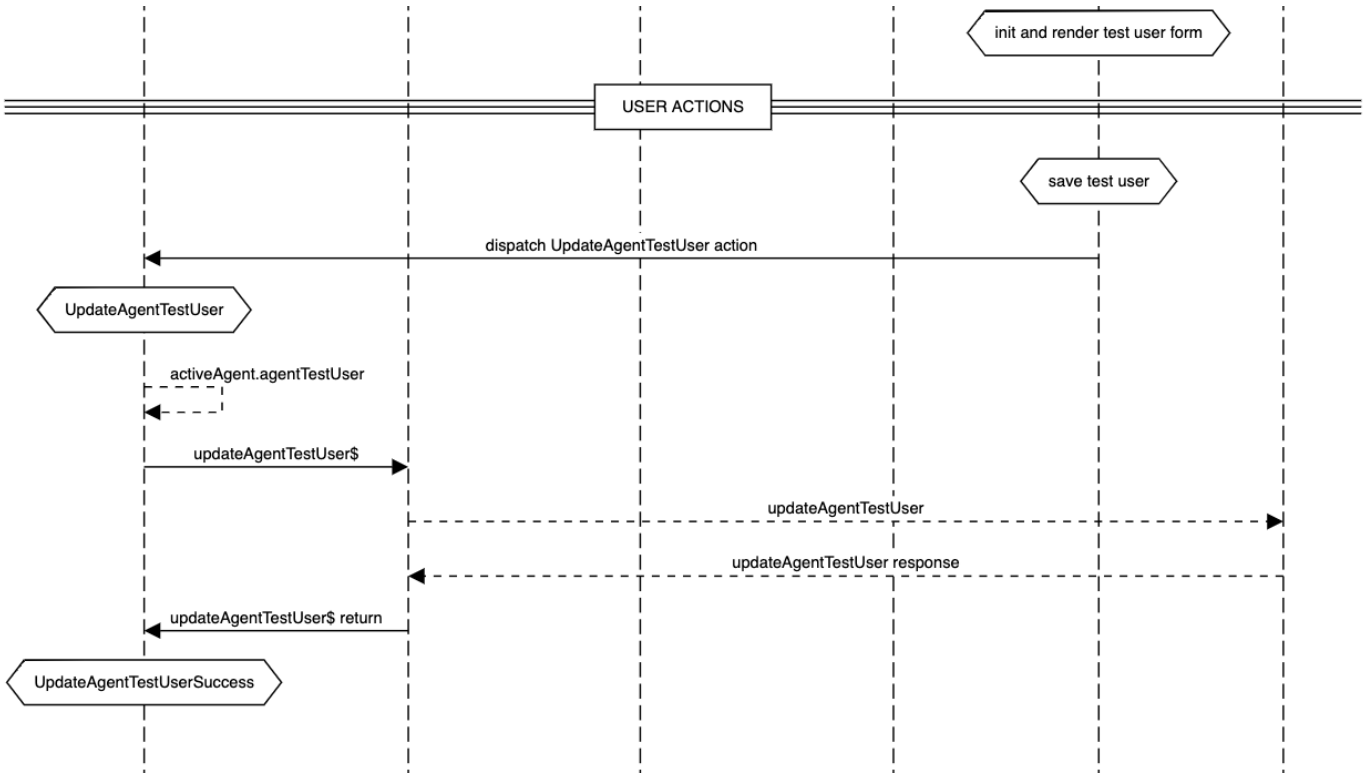
Test User Settings

OverviewConfigure the test user to be used for the Dialogue Explorer's simulatorScreenshots



Sequence Flow



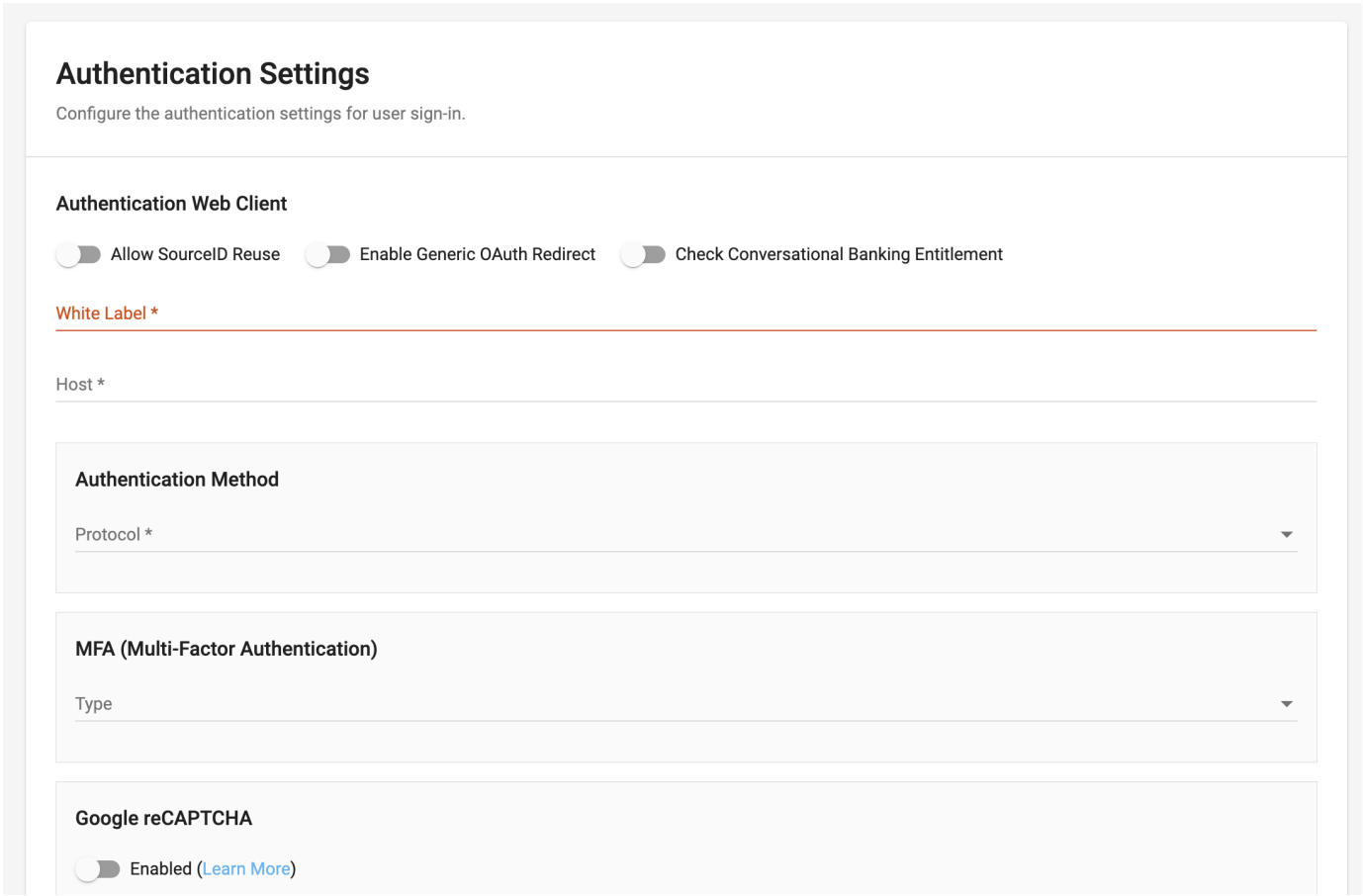


Authentication Settings

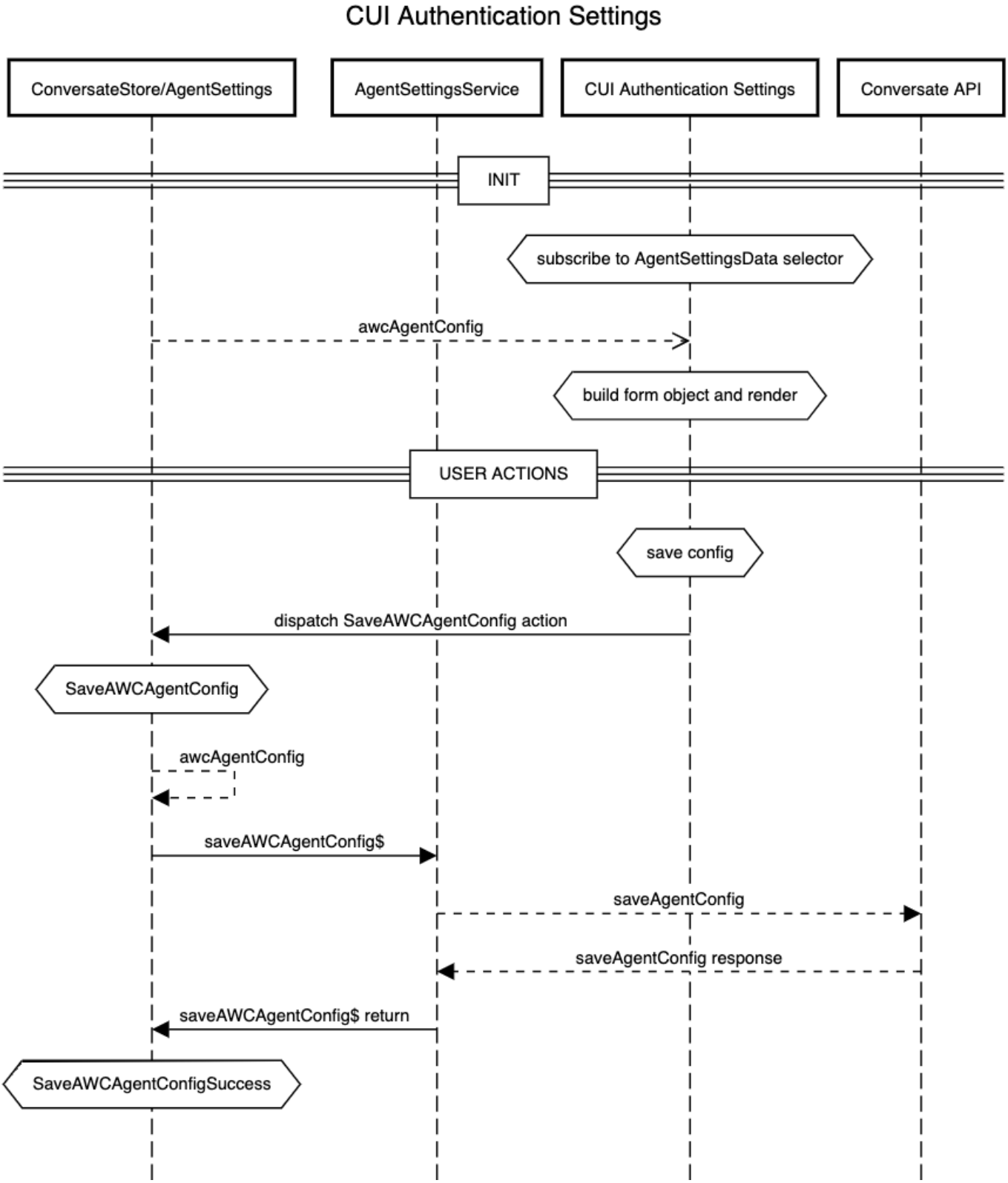
Overview

Allows Assistant managers to set up how end-users will authenticate via AuthWebClient.

Screenshots



Sequence Flow



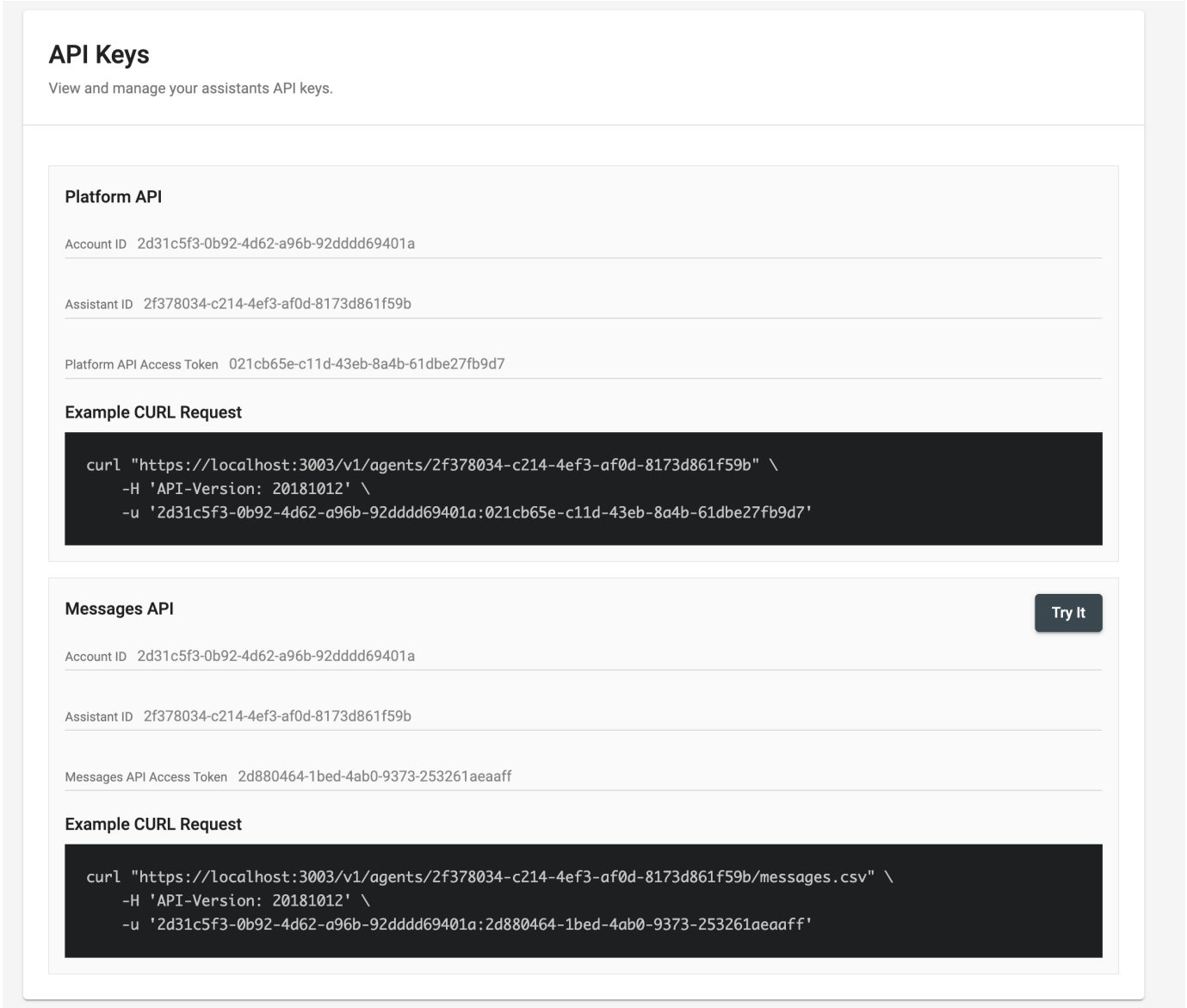
API Keys

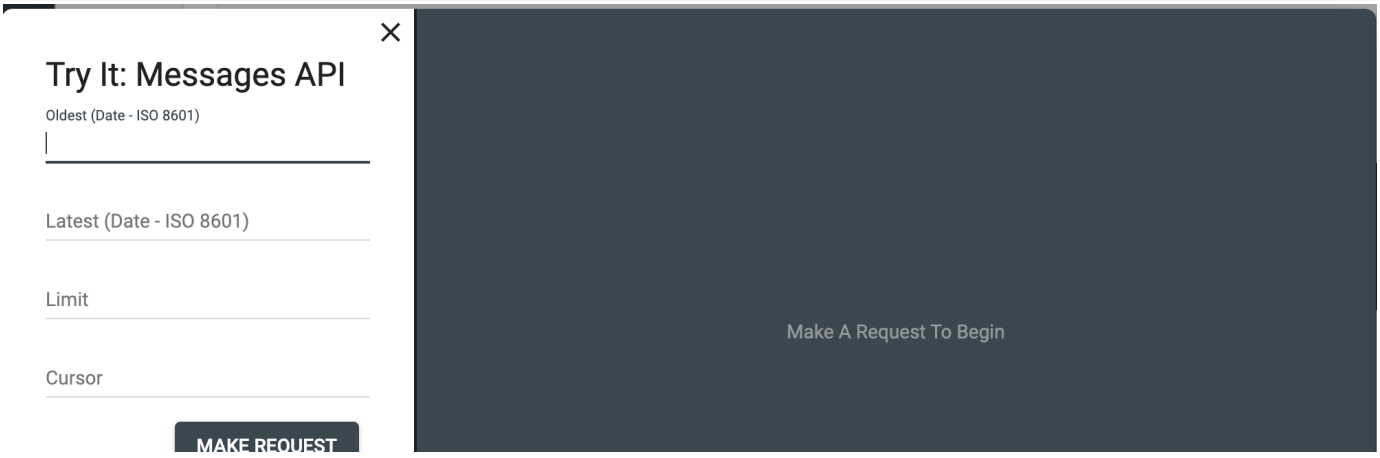
Overview

Every assistant can be interacted with via it’s own APIs.
Platform API

A general API that allows for access to get accounts, get Agents, create/delete agents, create/delete channels, import/export a config, or create /delete integration services.
Messages API

This api provides a connection to retrieve a CSV formatted list of messages, for a given time range, count, or starting location.
Screenshots





Compliance

Overview

Compliance module lists updates to the assistant's dialogue flow (such as intent creation/modification) and settings. Users can rollback changes to previous versions of config snapshots, in the event of an unintended consequence of a change. Additionally, users can download the CUI config for the assistant in JSON format.
What are Snapshots?

Anytime a CUI user updates an A.I. Assistants dialogue flow, our system will take the current version, and store it as a snapshot before updating the config in the database. These snapshots are useful because they give the ability to roll back to any previous snapshot in case of a bad configuration. Additionally, we can download a previous snapshot in JSON to manually compare or edit.

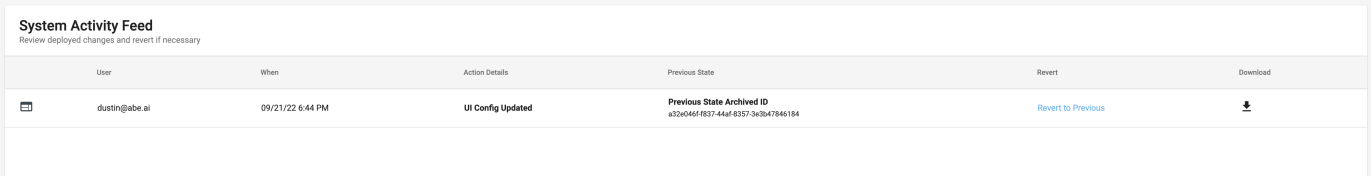
Compliance Guard

The `ComplianceGuard` is a singleton object that provides a route guard to be used in routing modules. It uses the canActivate interface, and performs an async check. If this guard is triggered, and the Compliance State is not hydrated, or in the process of hydrating, the guard will dispatch the LoadCompliance action. It will then wait for an update to the state, and then proceed. The result is the ability to apply this guard to a route, and that route will not activate until we know that the state has been hydrated with data.

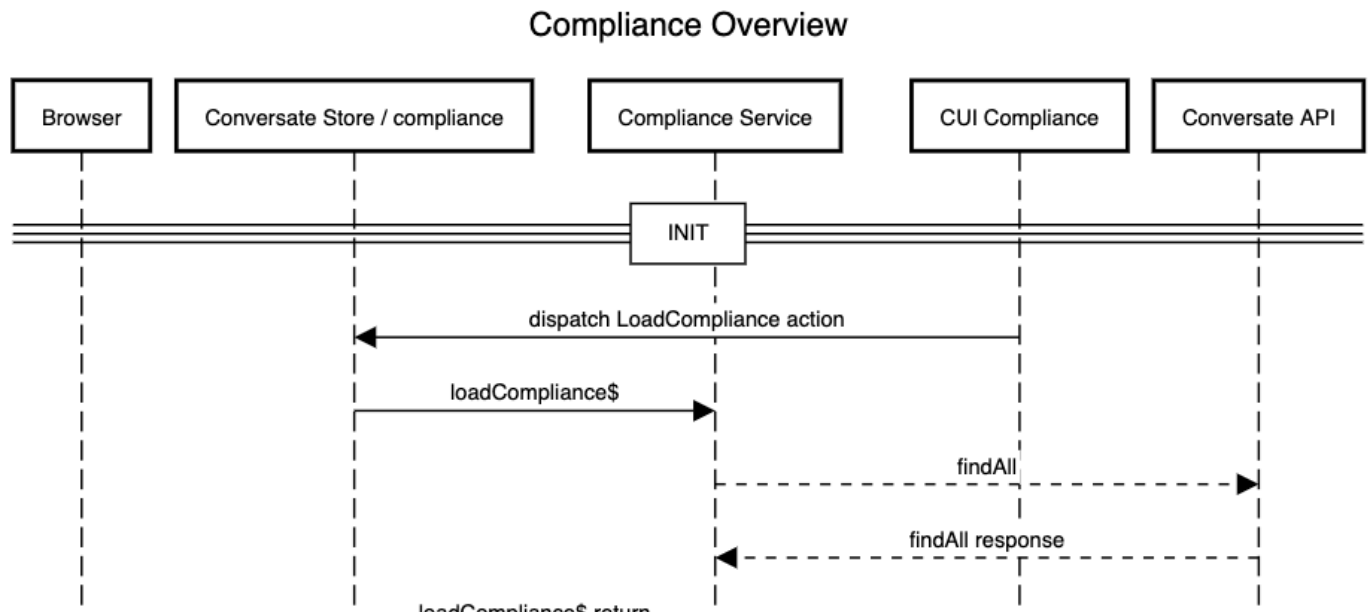
Compliance Service

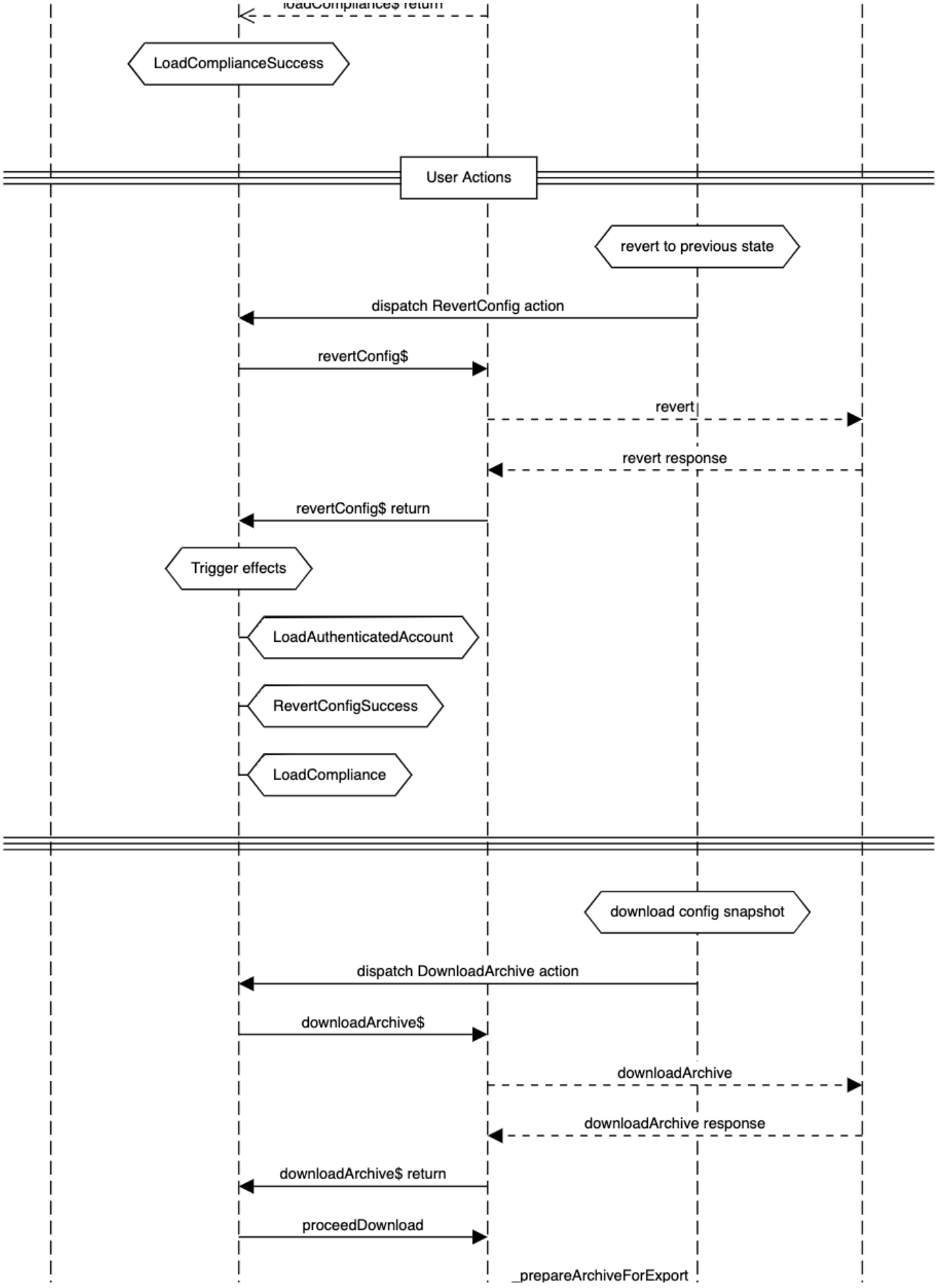
The `ComplianceService` is a singleton object, used by the Compliance State's action effects, providing methods that allow for interaction with Conversate api via the Socket Service.

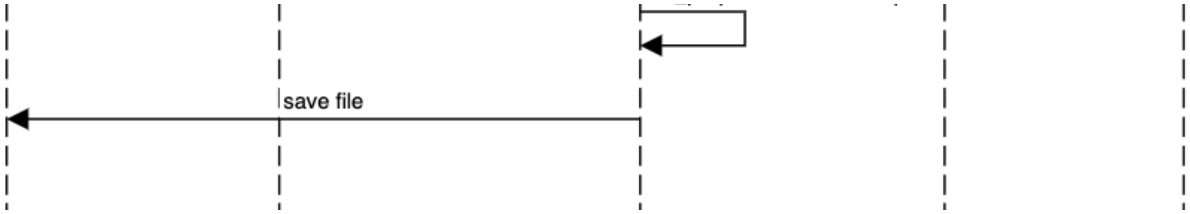
Compliance Component UI Preview



Sequence Flow




loadCompliance$ return
LoadComplianceSuccess
User Actions
revert to previous state
dispatch RevertConfig action
revertConfig$
revert
revert response
revertConfig$ return
Trigger effects
LoadAuthenticatedAccount
RevertConfigSuccess
LoadCompliance
download config snapshot
dispatch DownloadArchive action
downloadArchive$
downloadArchive
downloadArchive response
downloadArchive$ return
proceedDownload
_prepareArchiveForExport



refs

- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98636104745/Compliance+State#LoadCompliance
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98636104745/Compliance+State#RevertConfig
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98636104745/Compliance+State#DownloadArchive
- https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98637152263/Authenticated+Account+State#LoadAuthenticatedAccount

CUI Config

Overview

CUI Config module is a logic module, meaning it does not contain views to be rendered in the UI. It is a collection of models and logic to support management of a CUI config. It provides the `ConfigService` singleton, and the `CuiConfigGuard`. It interacts with the CUI Config State.

Config Service

The Config Service is a singleton object that provides methods to interact with `Conversate API` from within CUI. It is mostly directly used by the CUI Config State with its action effects.

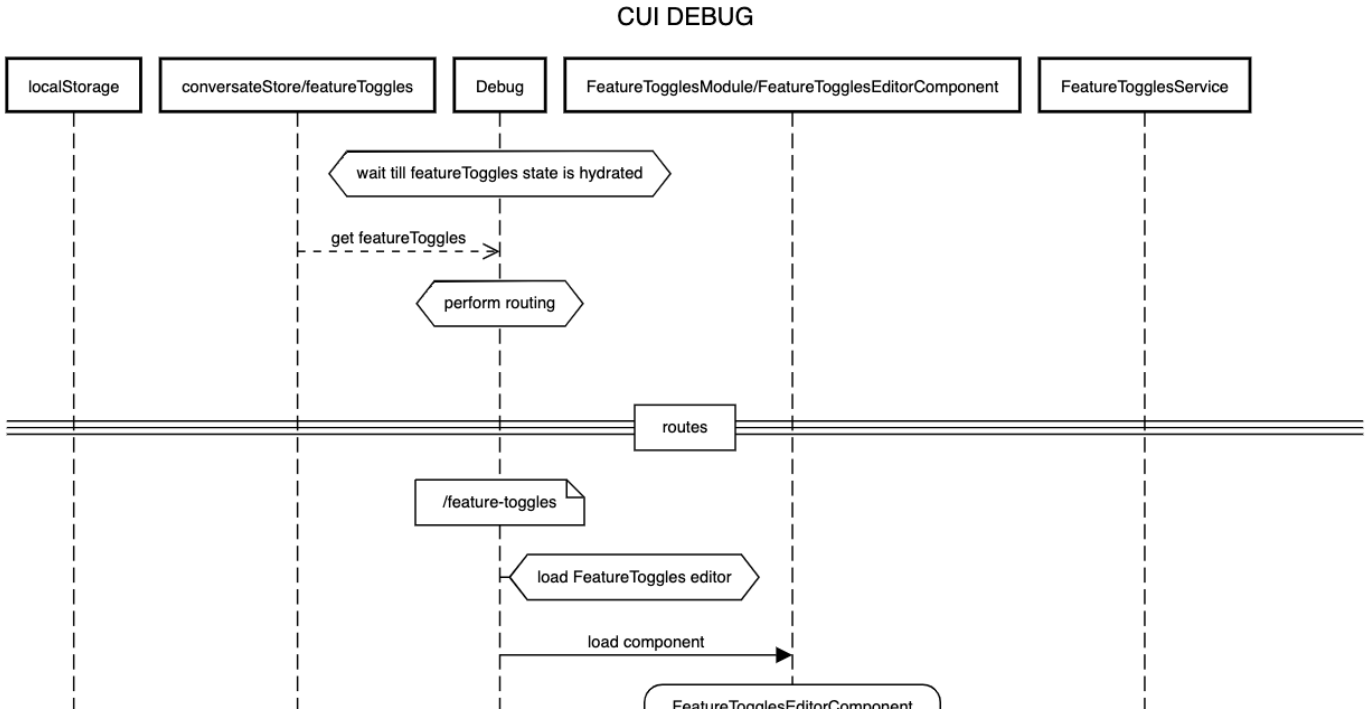
ConfigGuard

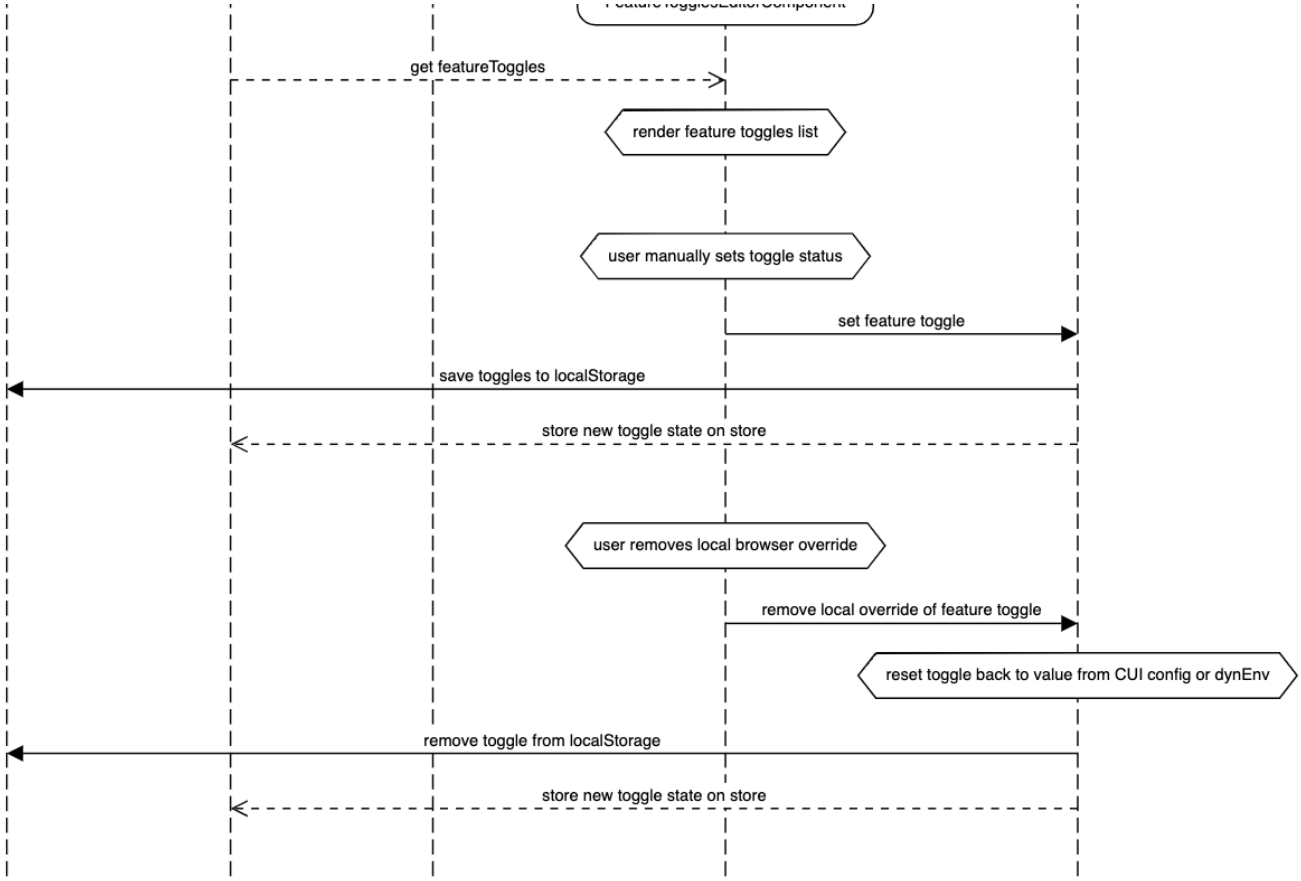
The ConfigGuard is a singleton object that provides a route guard to be used in routing modules. It uses the canActivate interface, and performs an async check. If this guard is triggered, and the CUI Config state is not hydrated, or in the process of hydrating, the guard will dispatch a https://envestnet.atlassian.net/wiki/spaces/CAPD/pages/98636137652/CUI+Config+State#LoadConfig action. It will then wait for an update to the state, and then proceed. The result is the ability to apply this guard to a route, and that route will not activate until we know that the CUI config state has been hydrated with data.

Debug

Overview

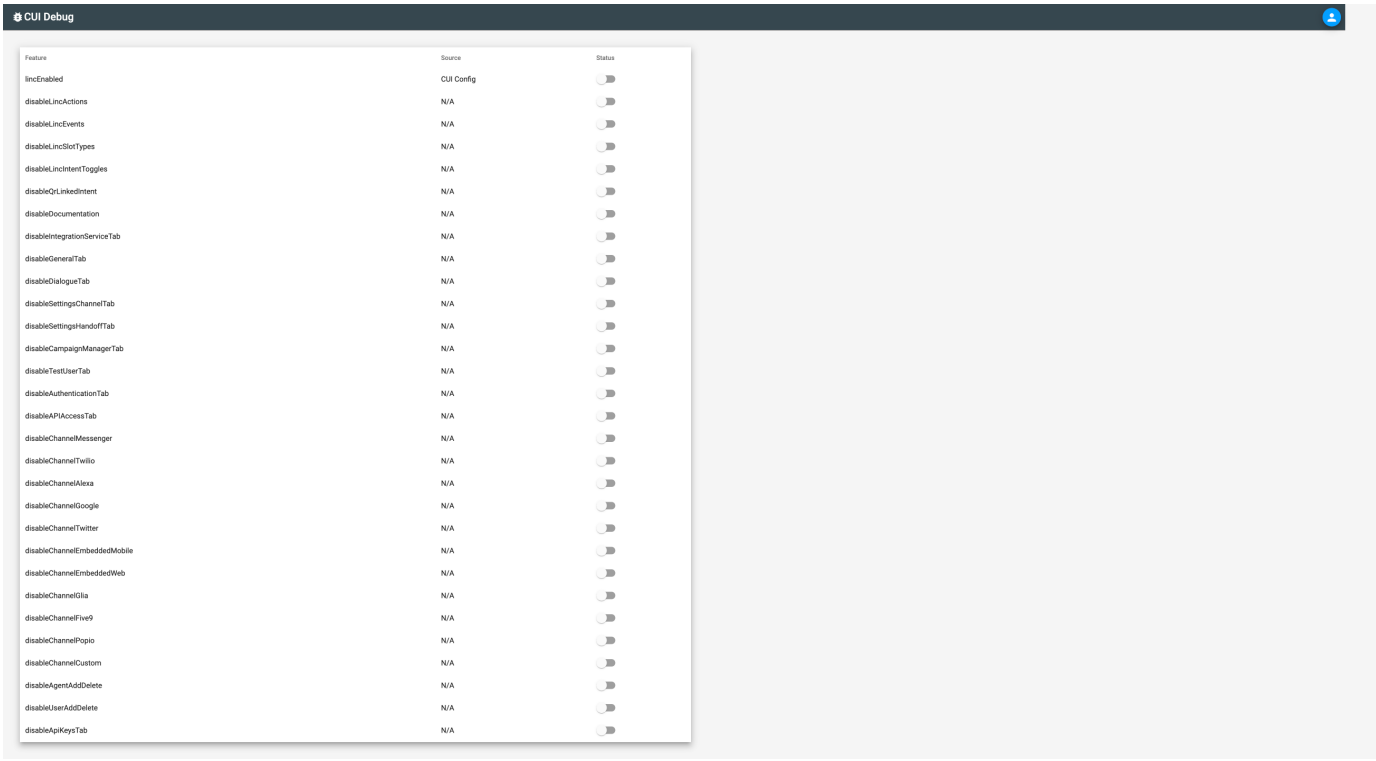
Debug module is a hidden module within the conversate app. We do not link to this route from anywhere, users must know the route exists, and manually route there via url. It can be used to set/reset feature flags.





Feature Flags Toggle UI

A simple UI to view the source of the current state for each feature flag, as well as set.remove local overrides. See Feature Toggle for more information on feature flag inheritance.



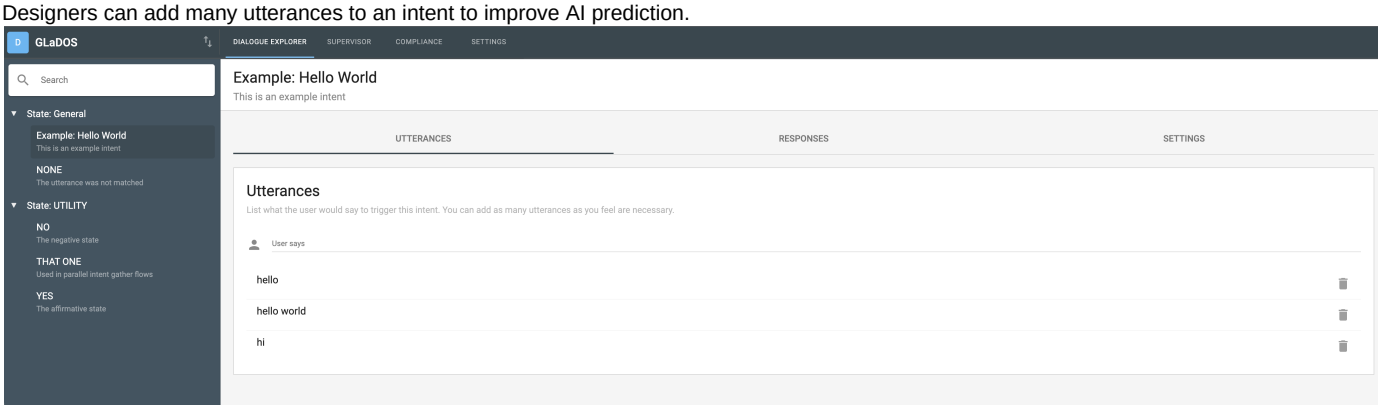
Dialogue Explorer

Overview

Dialogue Explorer is where conversation designers (CDs) manage and test an A.I. Assistants conversation flow.Intent Editing

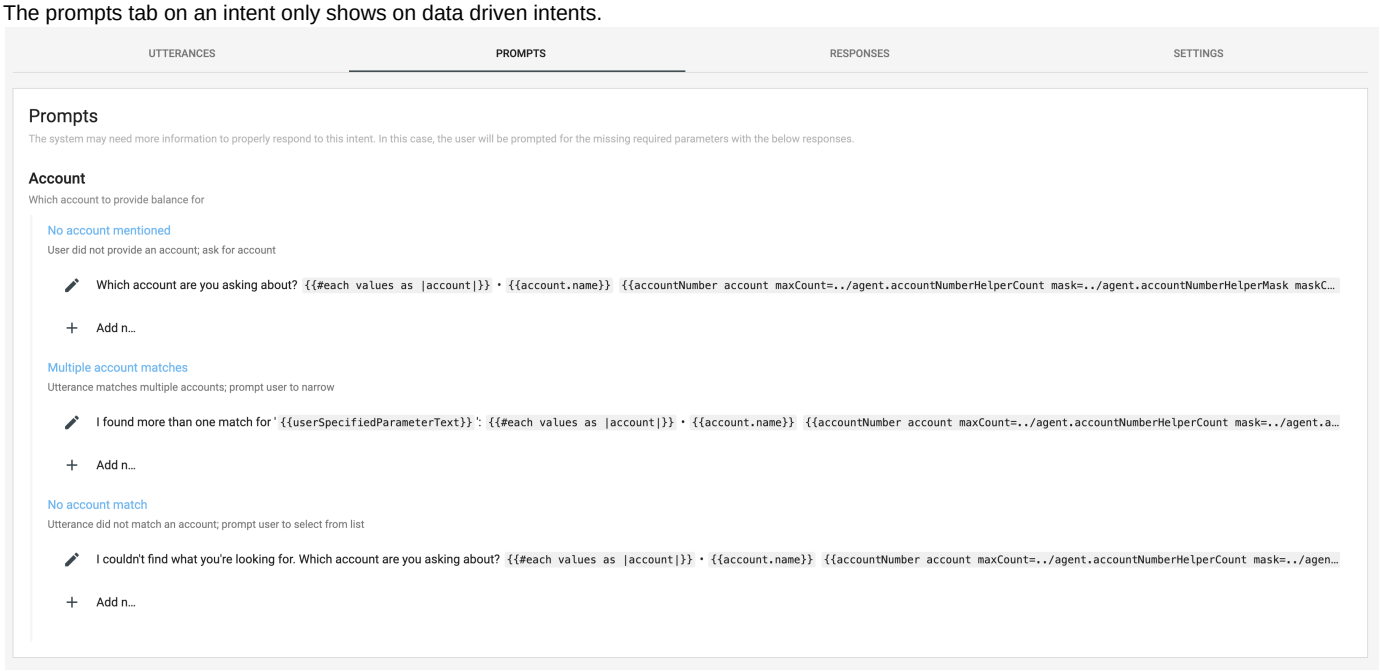
They have the ability to create/edit intents, their response templates, and their utterance training data. Additionally, CDs can choose to make an intent data driven, and tie it to an operation in a registered integration service.
Utterances

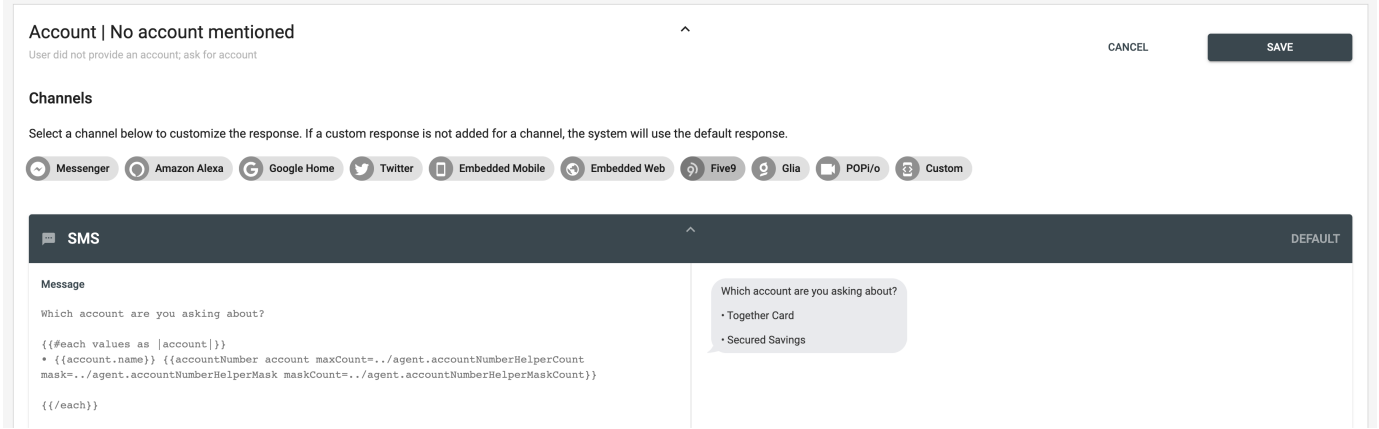
Designers can add many utterances to an intent to improve AI prediction.



Prompts

Prompts are responses to the end user that are triggered when a data driven intent needs more information in order to execute. They are part of g`ather mode`, and when all required data parameters are filled, the assistant will move on to the actual response template defined for the intent. The prompts tab on an intent only shows on data driven intents.

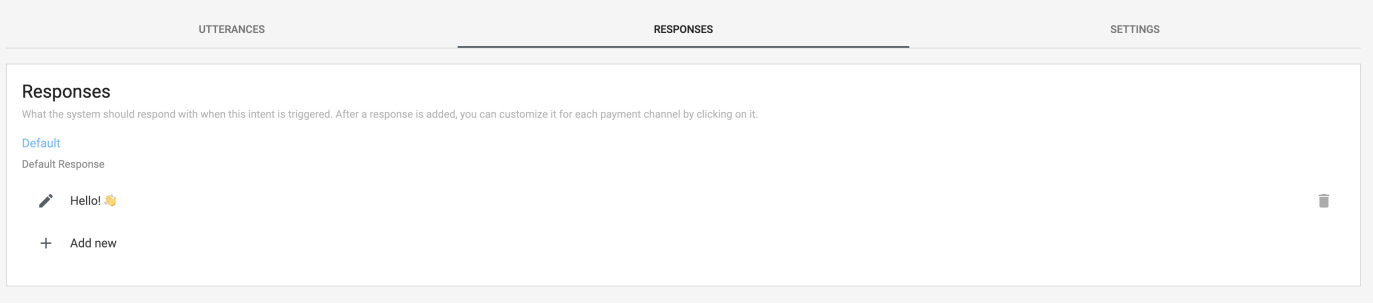


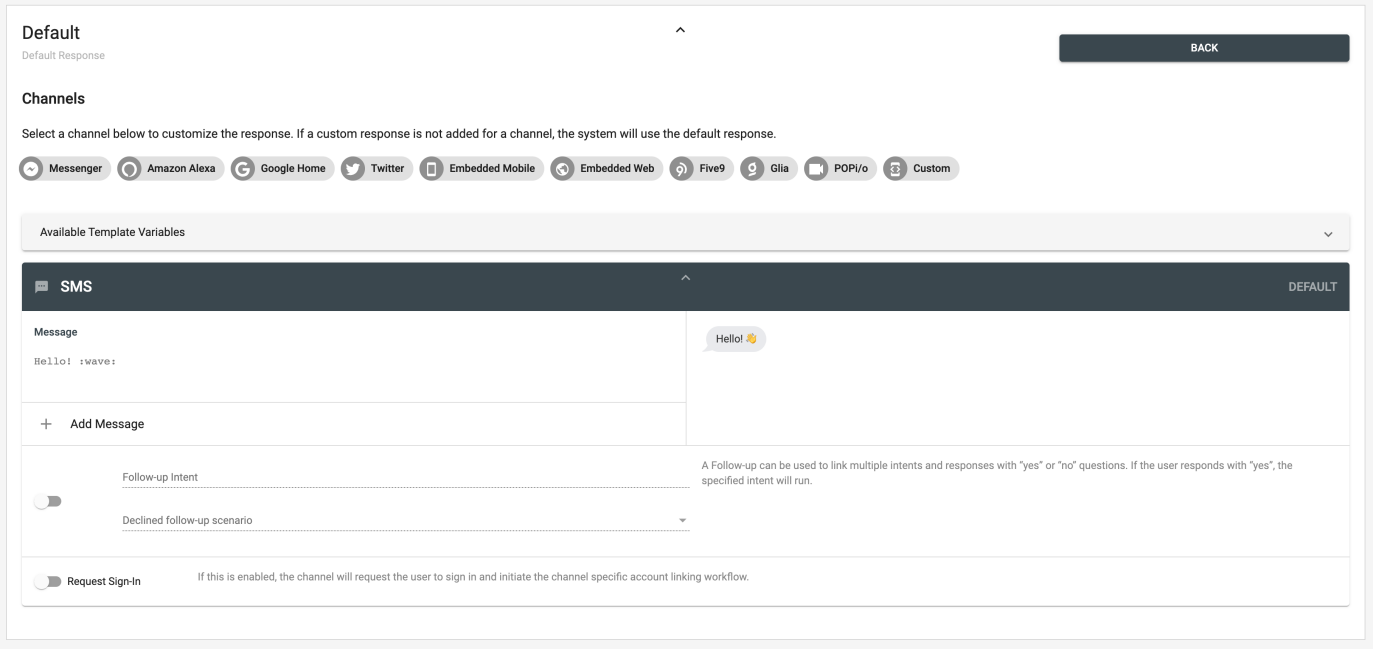


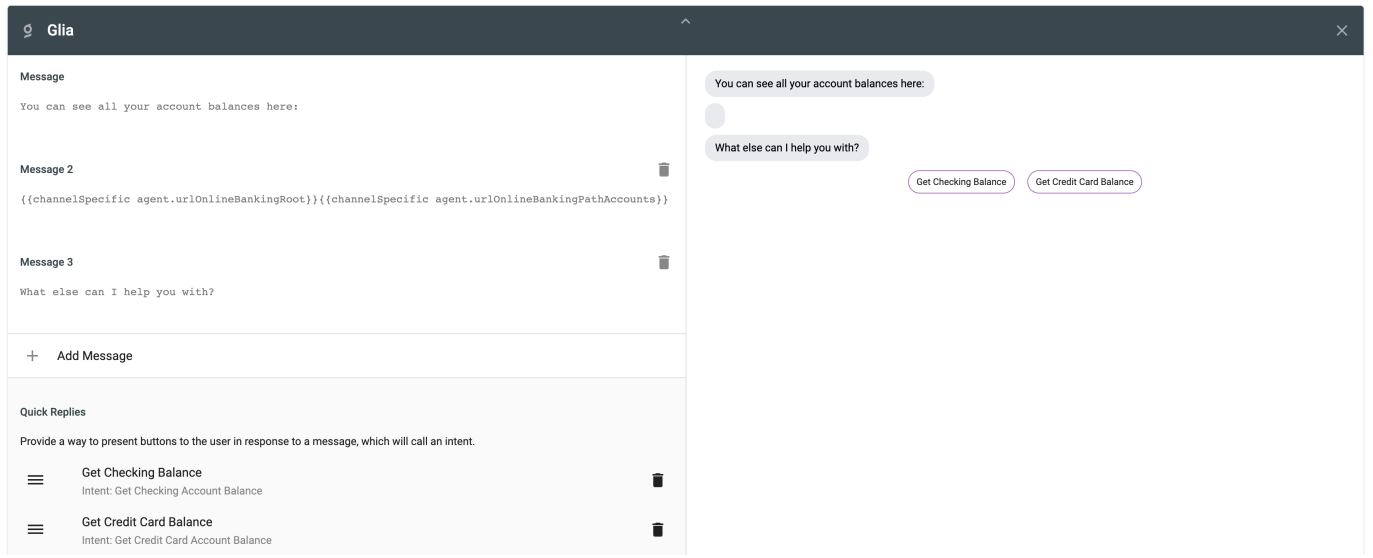
Responses

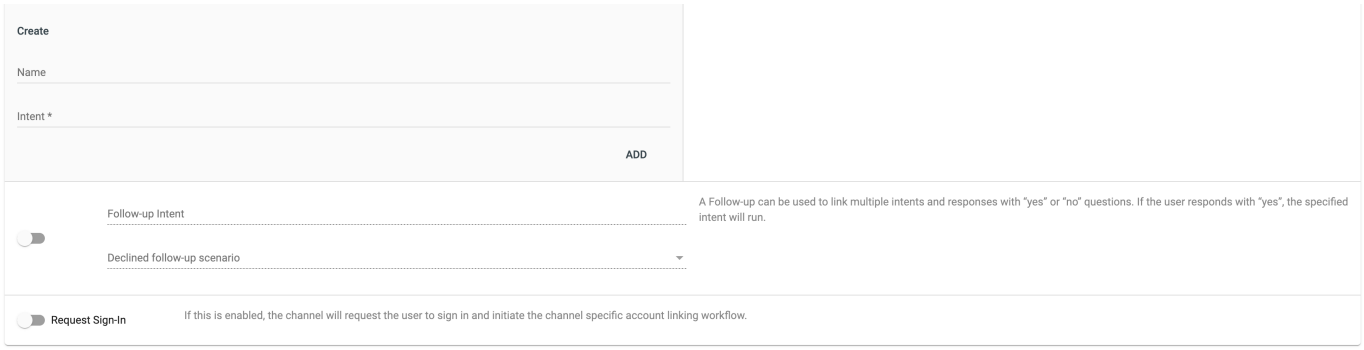
Responses can be configured to have a unique template per channel, and channels will fallback to SMS if they do not have a unique template. Multiple responses can be configured on an intent, this will result in a response being chosen at random when an intent is triggered. Responses can also be used to trigger a `follow up intent`, allowing CD's to create more dynamic flows. A response template is written with support for Handlebars templating syntax, as well as markdown style links.

Some channels have the ability to provide quick replies to the end user, packaged with the response content. This gives the end user a quick way to respond to a message from the AI Assistant via clickable bubble buttons.

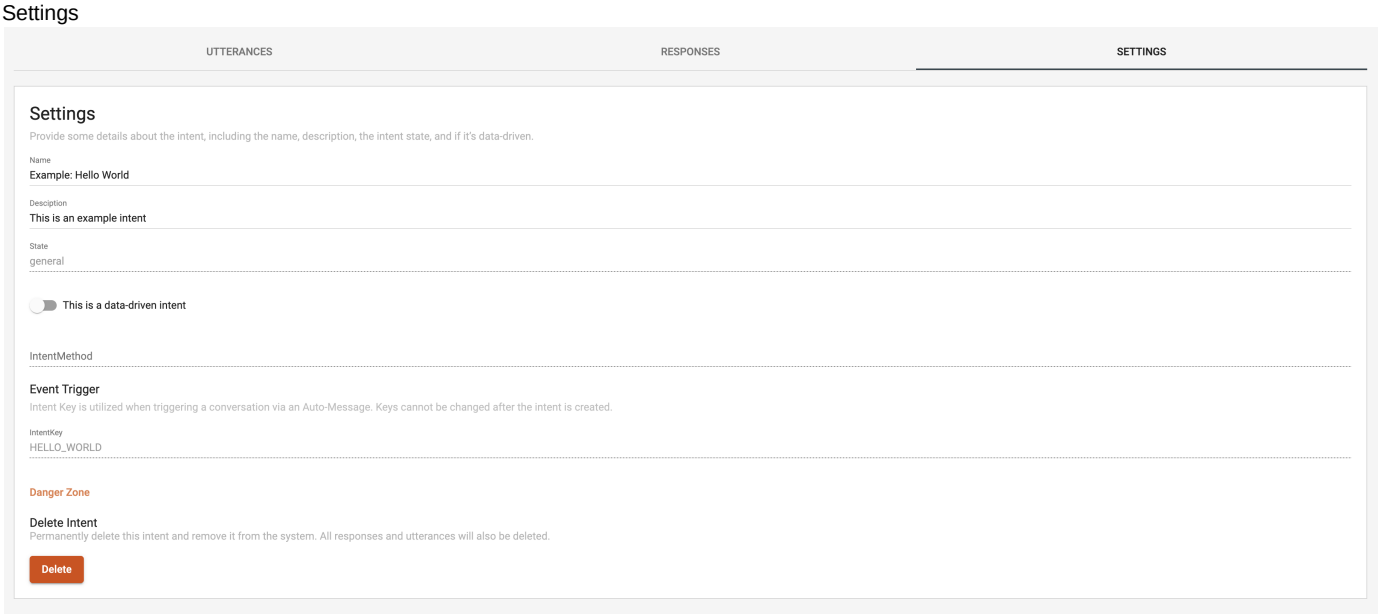






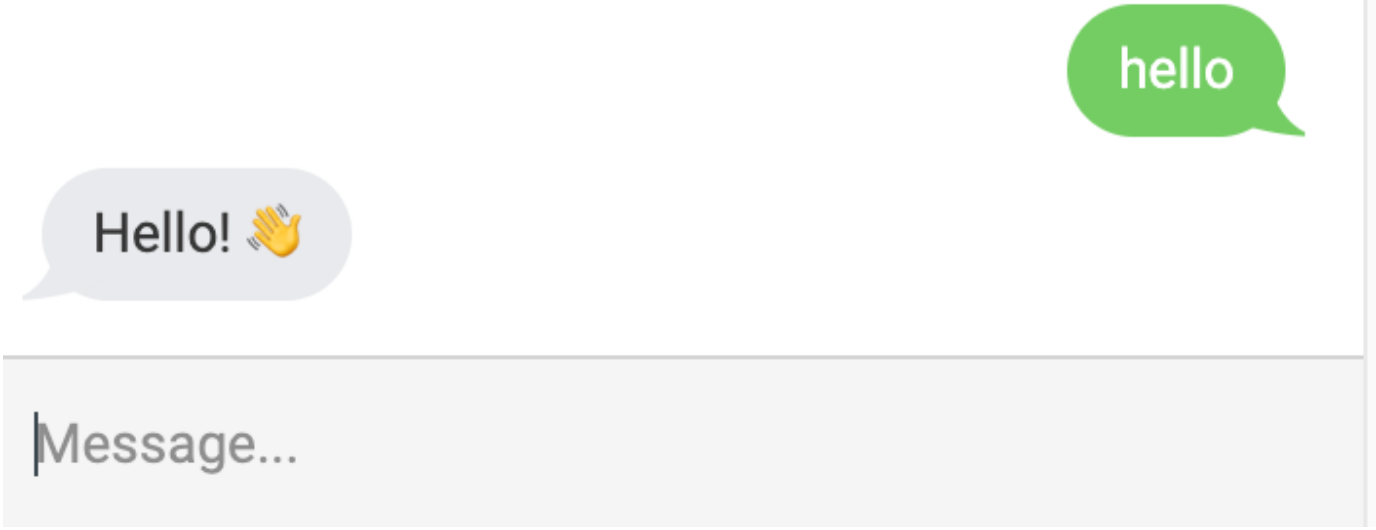


Settings

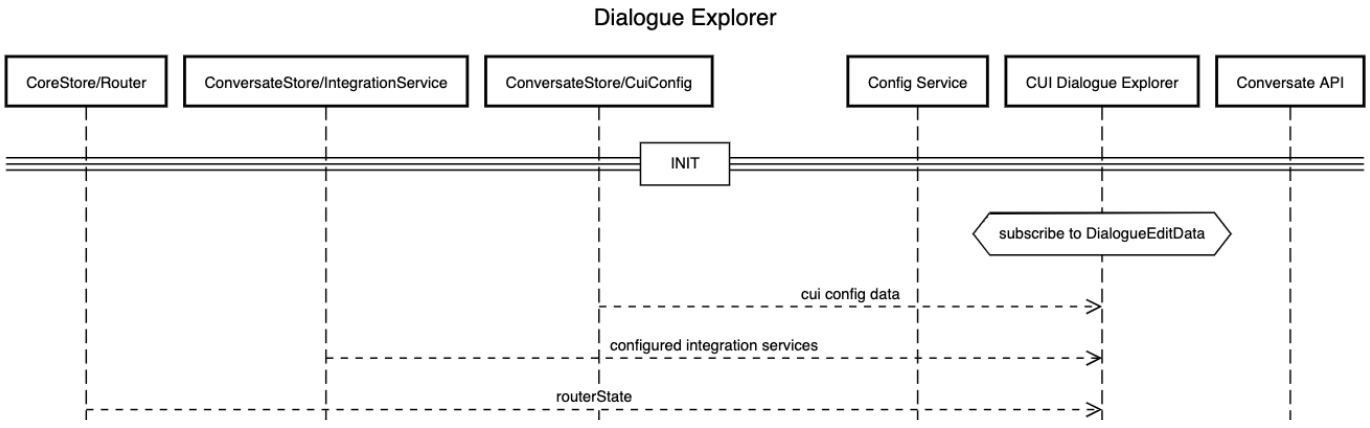


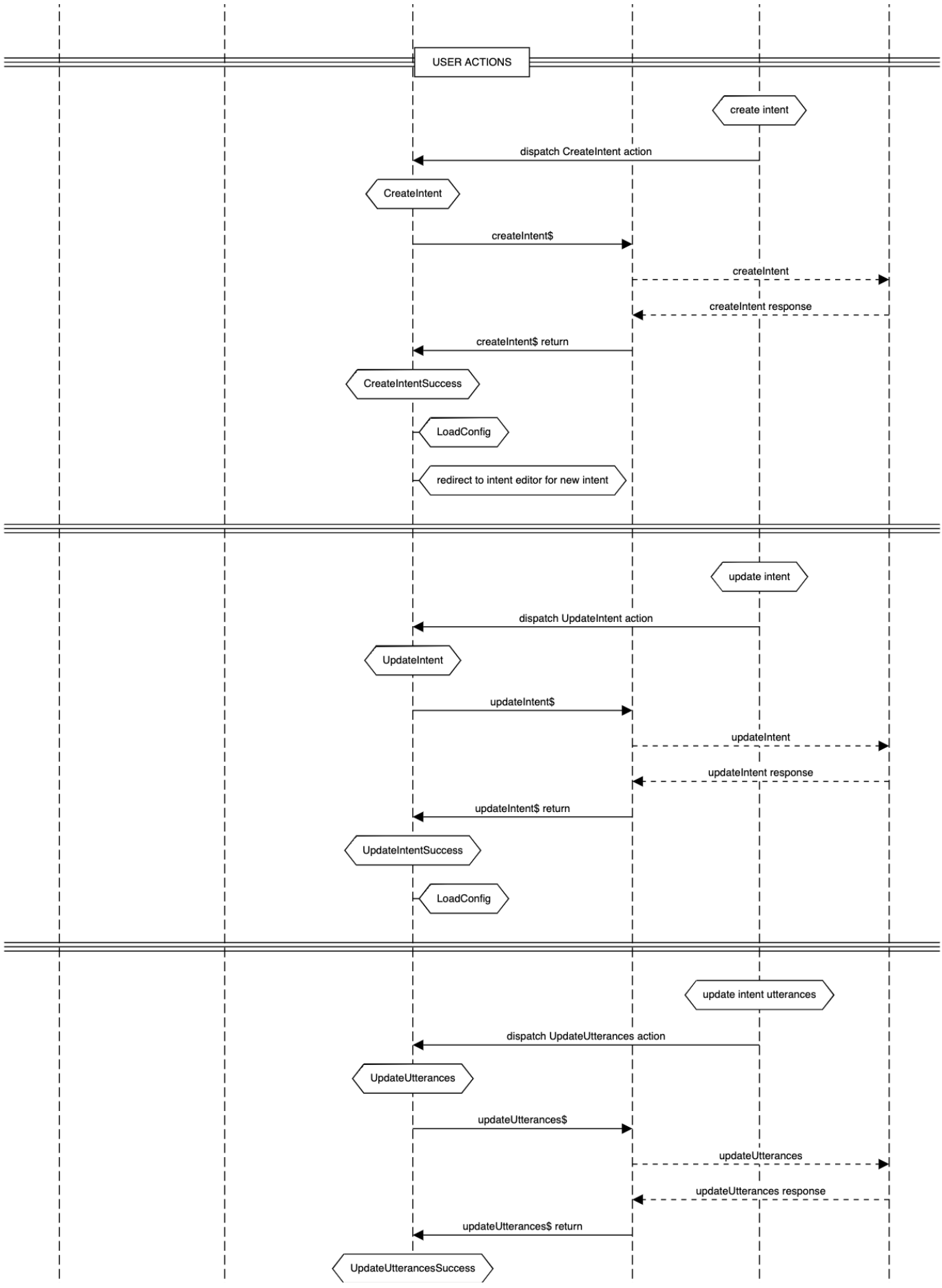
Dialogue Simulator

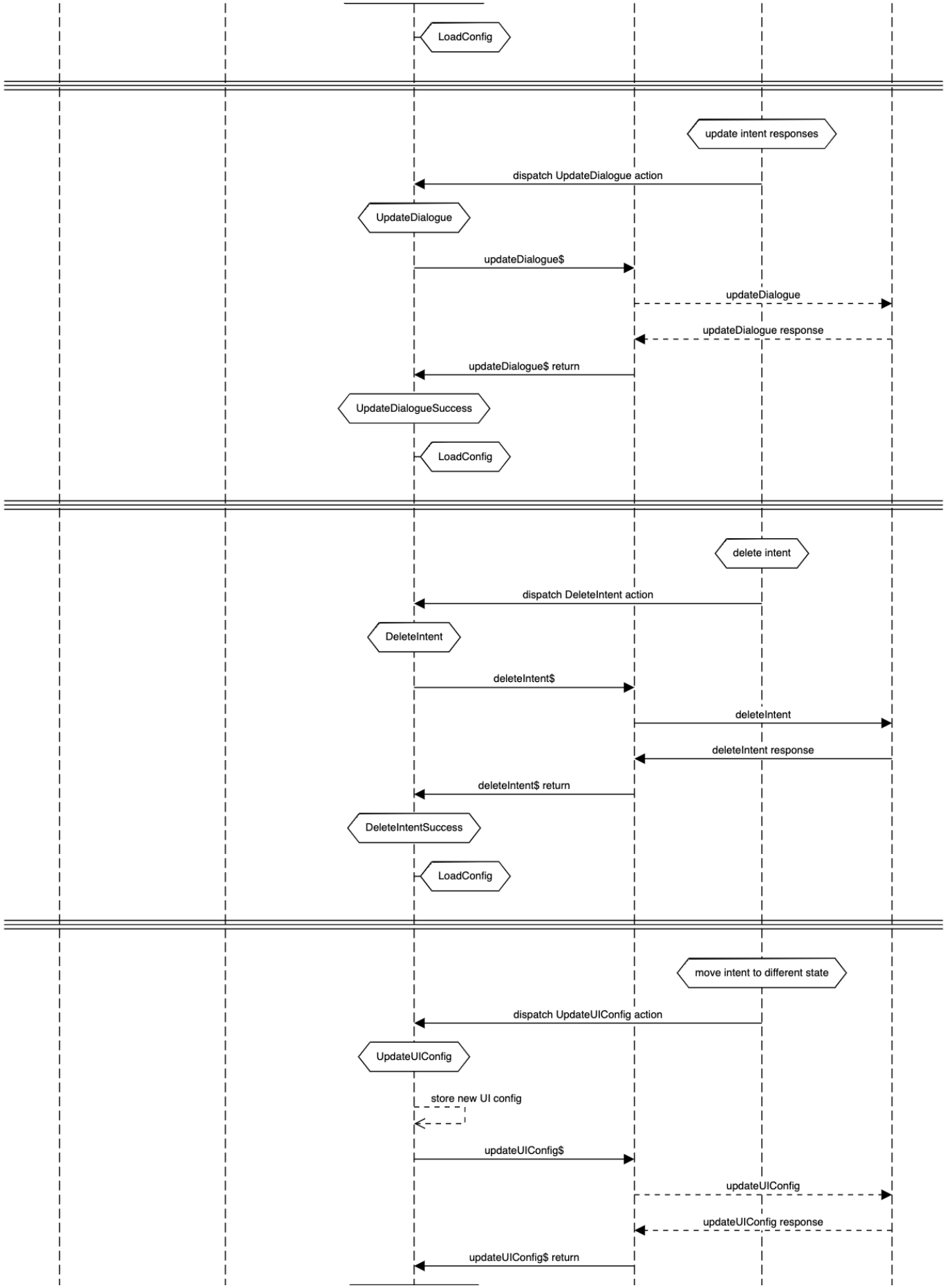
Dialogue Explorer also contains the Dialogue Simulator component, allowing CDs to interact with the trained assistant, and view how the assistant will respond in all channels. The active view channel can be changed via the channel picker at the top of the simulator.



Sequence Flow




USER ACTIONS
create intent
dispatch CreateIntent action
CreateIntent
createIntent$
createIntent
createIntent response
createIntent$ return
CreateIntentSuccess
LoadConfig
redirect to intent editor for new intent
update intent
dispatch UpdateIntent action
UpdateIntent
updateIntent$
updateIntent
updateIntent response
updateIntent$ return
UpdateIntentSuccess
LoadConfig
update intent utterances
dispatch UpdateUtterances action
UpdateUtterances
updateUtterances$
updateUtterances
updateUtterances response
updateUtterances$ return
UpdateUtterancesSuccess


LoadConfig
update intent responses
dispatch UpdateDialogue action
UpdateDialogue
updateDialogue$
updateDialogue
updateDialogue response
updateDialogue$ return
UpdateDialogueSuccess
LoadConfig
delete intent
dispatch DeleteIntent action
DeleteIntent
deleteIntent$
deleteIntent
deleteIntent response
deleteIntent$ return
DeleteIntentSuccess
LoadConfig
move intent to different state
dispatch UpdateUIConfig action
UpdateUIConfig
store new UI config
updateUIConfig$
updateUIConfig
updateUIConfig response
updateUIConfig$ return

Pending Changes Guard

The `PendingChangesGuard` is a singleton object the uses the CanDeactivate interface. This guard will prevent users from leaving a page if there are pending changes. It uses the CUI Config State's property `pendingChanges`, to determine if it should prevent the user from navigation. If there are pending changes, the user will be presented with a modal, and upon confirmation that the unsaved changes will be lost, the guard will let the navigation occur.

Conversation Service

The `ConversationService` is a singleton object that is used within Dialogue Explorer. It provides methods that connect to Conversate API to enable conversations with the assistant via Dialogue Simulator.

Dialogue Utilities Service

The `DialogueUtilitiesService` is a singleton object providing methods to aid in creating conversation flows.

Dialogue Explorer LINC (deprecated)

Overview

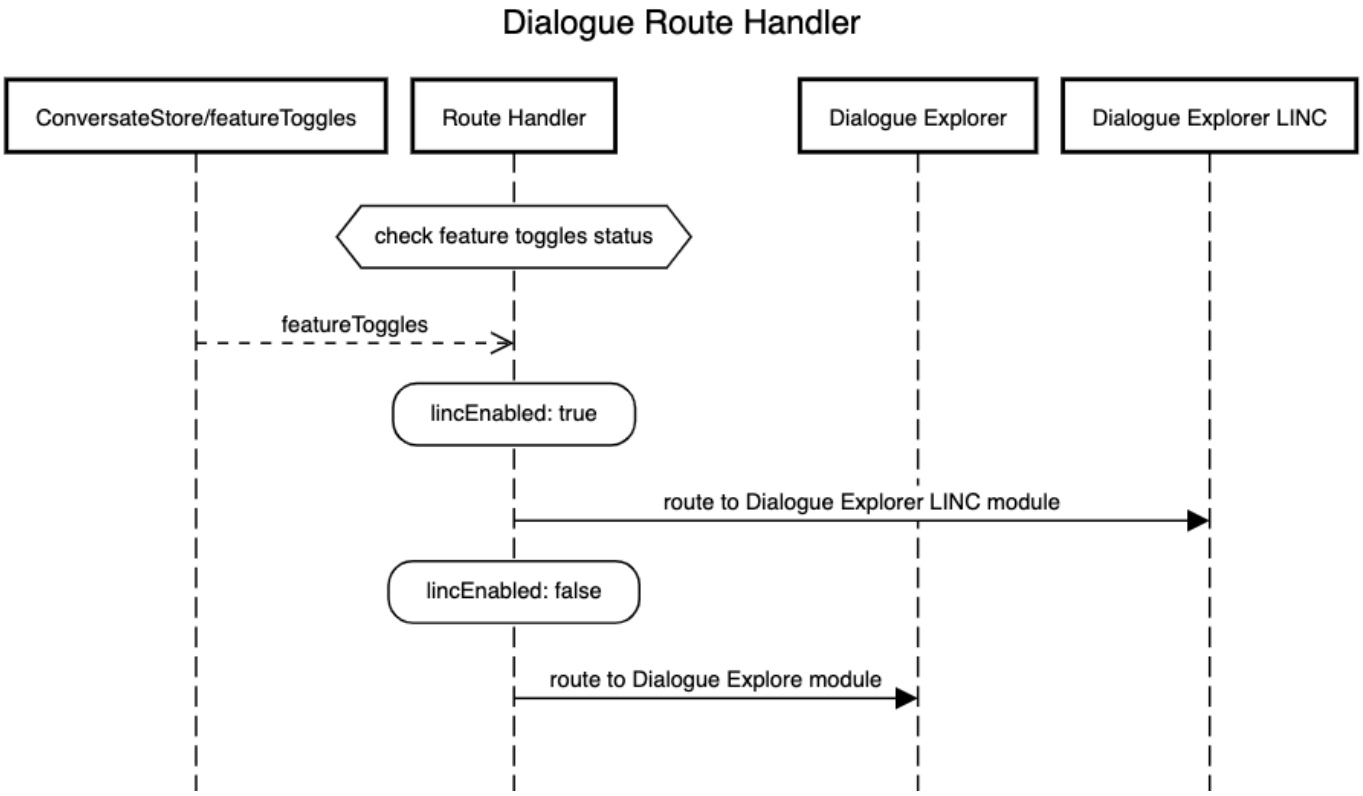
Dialogue Explorer LINC is where the deprecated LINC project is housed. It was intended to be a visual block based flow builder for conversations.

Dialogue Route Handler

Overview

Dialogue Route Handler is a special module with no views. It's purpose is to properly handle routing for the shared /dialogue route. This route is shared between Dialogue Explorer and Dialogue Explorer LINC, and determines which should be activated based on the lincEnabled feature flag.

Logic Flow



Note

With the deprecation of LINC, this module is still running upon navigation to `/dialogue`, but can be removed.

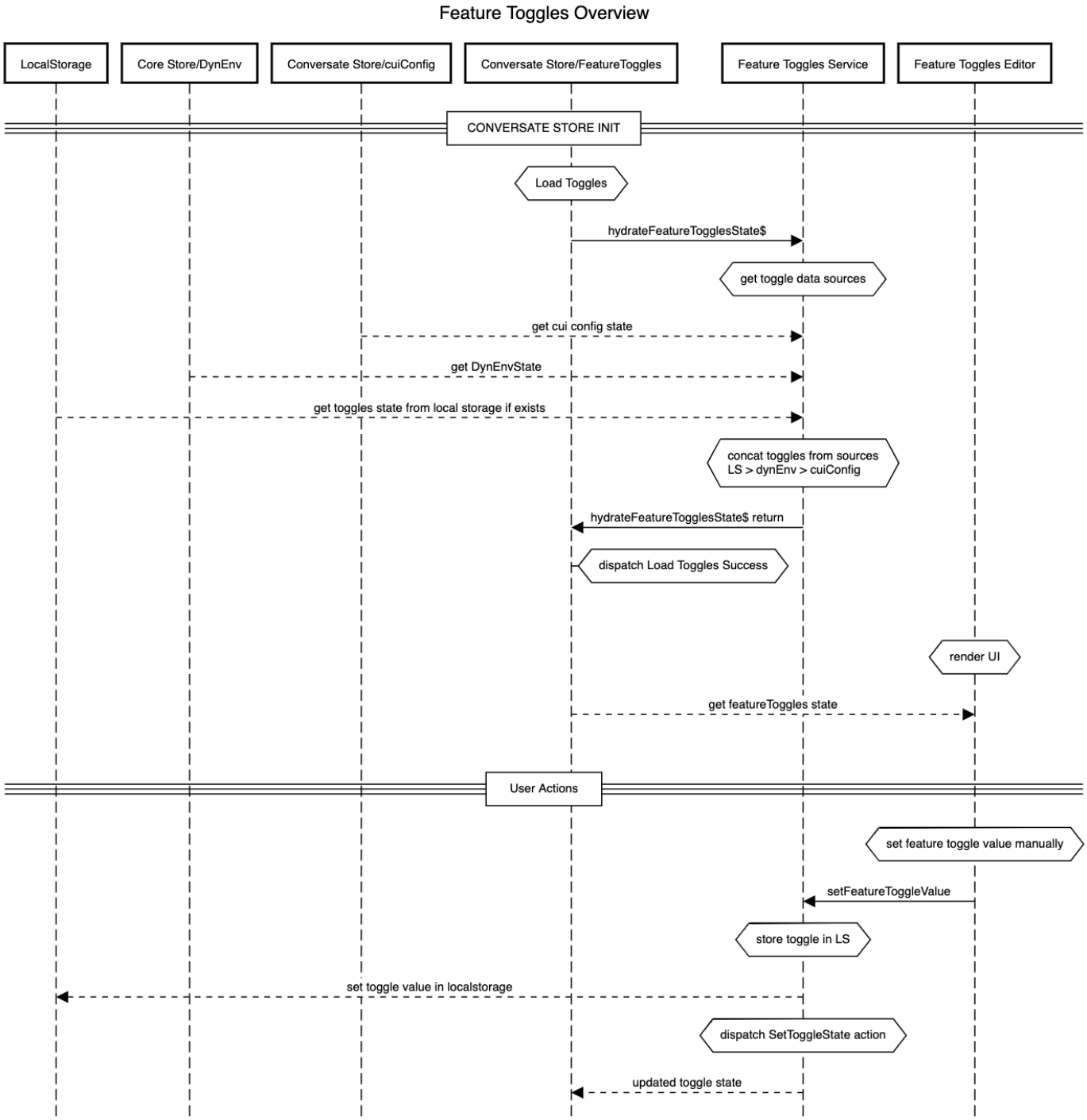
Feature Toggle

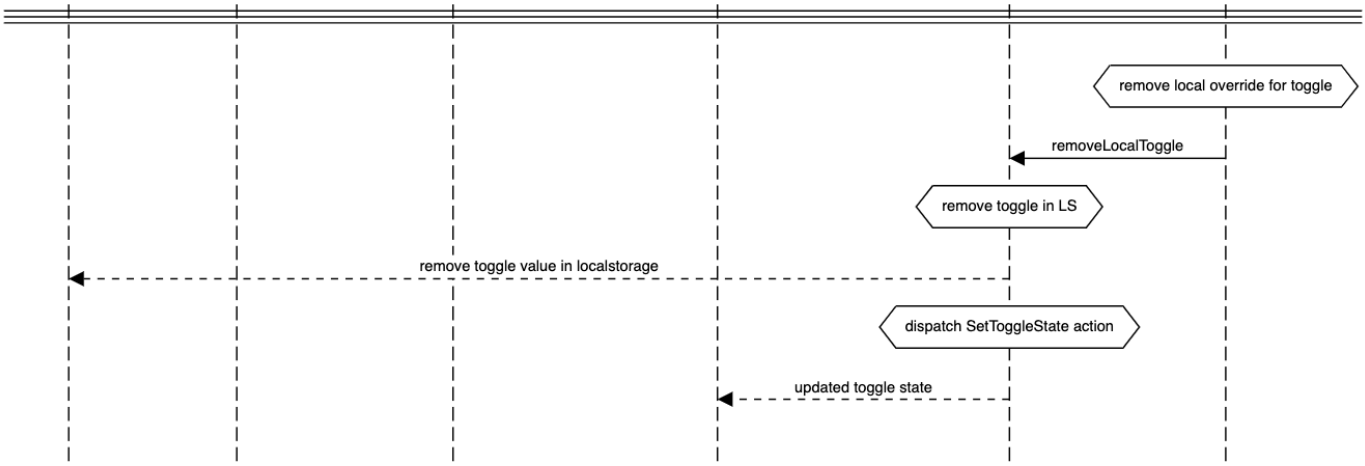
Overview

Feature Toggles Module is responsible for housing logic that configures, and hydrates the featureToggles State on Conversate Store. Additionally, it houses It also has a view component, the Feature Toggle Editor. To provide access to feature toggle logic to outside modules, we provide the Feature Toggles Service singleton.

Table Of Contents

Sequence Flow





Feature Toggles Editor Component

Overview

Feautre Toggles Editor Component is a simple UI that is exported from the Feature Toggles Module, and consumed by the debug module. It provides users the ability to manually set feature toggle states for their local browser. It also shows the current value of each toggle, as well as the source of the toggle value. If a toggle value is sourced from a manual override (stored in localstorage), then the user will be given the option to remove the local override, and rehydrate the toggle from dynamic sources such as cui-config or dynenv.

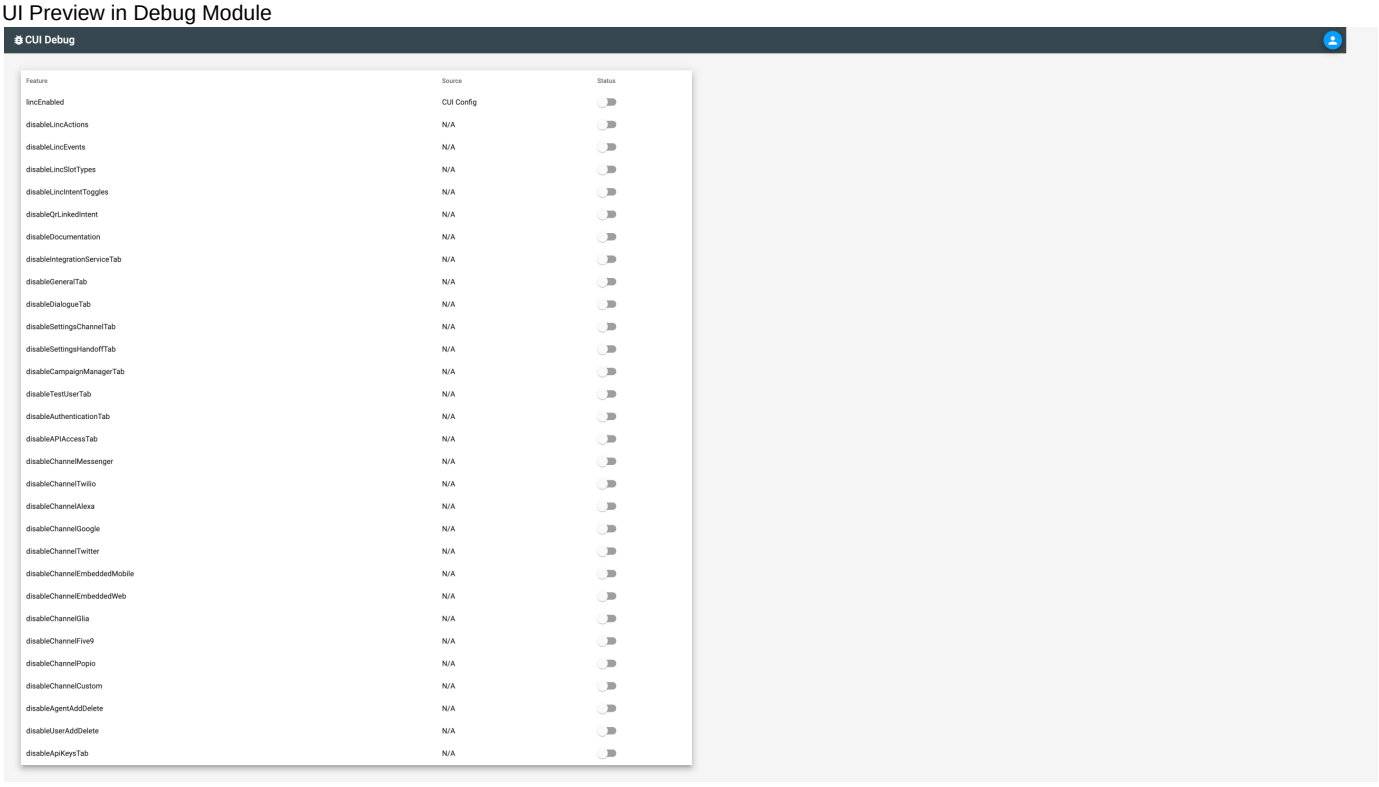
Dependencies

- Core Store, dynEnv state
- Conversate Store, cuiConfig & featureToggles states

Consumed by

- Debug Module routing

UI Preview in Debug Module



Feature Toggles Service

Overview

Feature Toggle Service is a singleton service responsible for interfacing with the featureToggle state on conversate store. It provides CRUD operation methods that operate on the featureToggle state. Additionally, it provides logical functions to do toggle state “authorization” matching.

Dependencies

- Conversate Store, cuiConfig & featureToggles states
- Core Store, DynEnv state

References

- Feature Toggle Editor Component
- conversate store / feature toggles effects

Integration Service

Overview

Integration service module is another module with no UI components. Its purpose is to house common logic surrounding Integration Services, such as the Integration Service Guard, the Configured Services Service singleton, the Template Services singleton, and typescript models representing an integration service.

Integration Service Guard

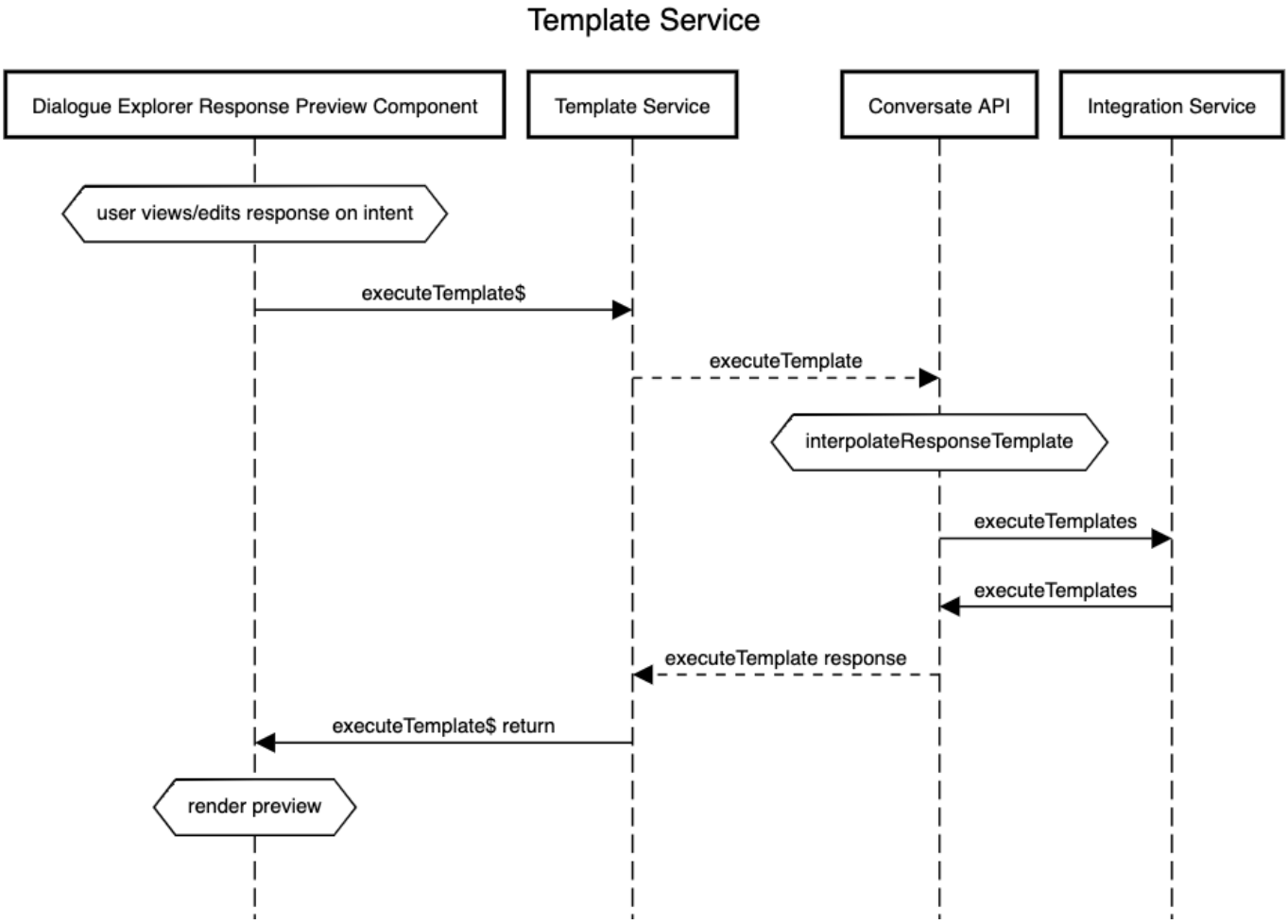
The Integration Service Guard is a singleton provided to be used as a route guard. Its purpose is to check to see if the integration service state on Conversate Store has been hydrated already. If it has not, it will trigger the LoadConfiguredServices action, and once the state is hydrated, it will allow the router to continue to whatever route we were guarding.

Configured Services Service

The Configured Services Service is a singleton that is provided for developers to be able to reuse common requests. This service is mostly used from within the Integration Service State 's effects, to make async calls over web-socket to Conversate .

Template Service

The template service is responsible for connecting to conversate api, when users are editing responses in Dialogue Explorer's response editor. When loading a response editor, it calls out to `executeTemplate`, a method on this service, to send the handlebars formatted response template to the default IS to be parsed and returned, so that the conversation designer can preview the response properly.



Supervisor

Overview

The Supervisor module gives the ability to analyze conversations that users have had with the A.I. Agent. It houses a service, a guard, and UI views.

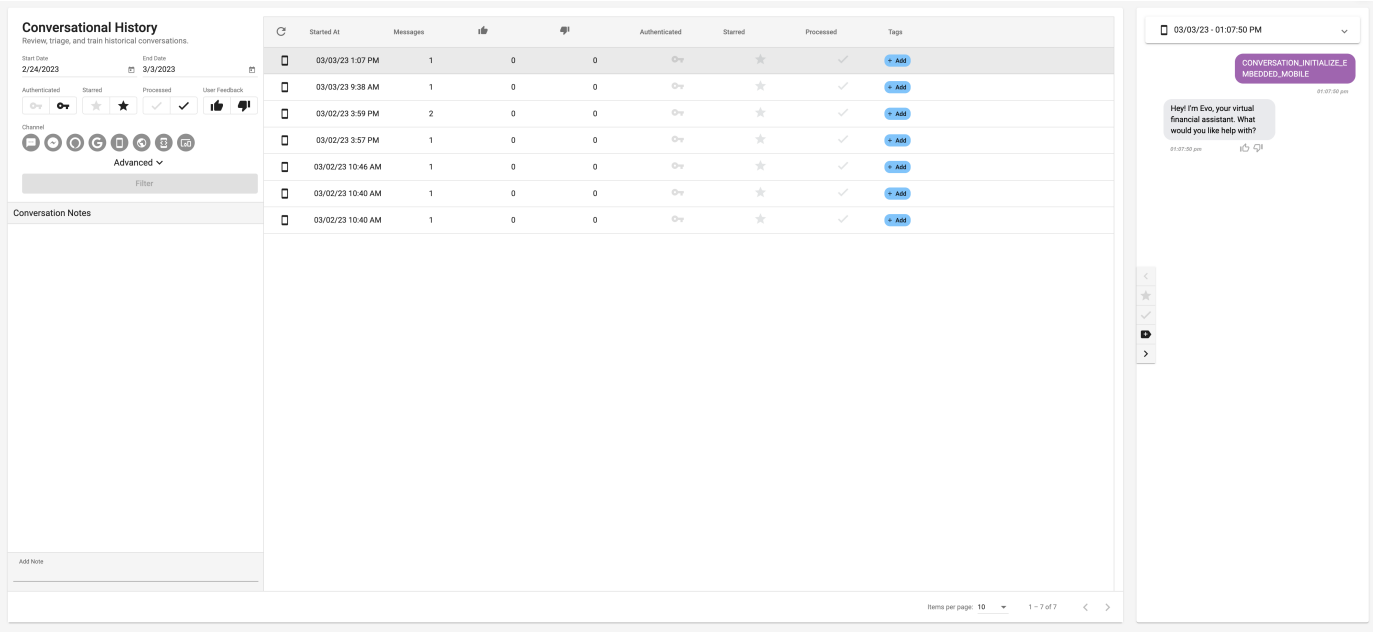
Supervisor Service

The `SupervisoreService` is a singleton object, used by the Supervisor State's action effects, providing methods that allow for interaction with Conversate api via the Socket Service.

Supervisor Guard

The `ComplianceGuard` is a singleton object that provides a route guard to be used in routing modules. It uses the canActivate interface, and performs an async check. If this guard is triggered, and the Supervisor State is not hydrated, or in the process of hydrating, the guard will dispatch the LoadConversations action. It will then wait for an update to the state, and then proceed. The result is the ability to apply this guard to a route, and that route will not activate until we know that the state has been hydrated with data.

Supervisor UI Preview



Validator (Deprecated)

The Validator module is no longer used, but was once responsible for providing a validator state to the store, and interfacing with the Validator service, for the purpose of validating Cui Configs

Conversate Store

Overview

conversate store is a store that contains data only used in conversate and it's sub modules. It used to be that all sub modules owned their own state reducers and actions, but now Conversate Module owns that code directly.

States

Each sub module in conversate has its own state to manage. You can read more about each state on their respective pages below
Agent Settings State

The Agent Settings state is responsible for housing information about the assistant's configuration for how end users of the assistant interact with Auth Web Client, and which client channels are connected. The UI to control those aspects are in Authentication Settings and Channel Settings respectively.
Compliance State

The compliance state is used to gather and store previous snapshots of the assistant config.
CUI Config State

the CUI Config state holds all objects related to managing an assistant's dialogue flow.
Feature Toggles State

The feature toggles state is responsible for hydrating and storing the current status of any feature toggle defined in our system
Integration Service State (ConfiguredServices)

The integration service state is responsible for housing all information regarding an assistant's configured integration services.
Linc Editor State (Deprecated)

The linc editor state is used to store objects needed to run the deprecated linc editor module, including the local working copy of the linc config for the assistant.
Navigation State

The navigation state is responsible for storing information about the top nav UI.
Supervisor State

The supervisor state houses data needed to run the Supervisor module, including conversations, and the active conversation.

Agent Settings State

Overview

The Agent Settings state is responsible for housing information about the assistant's configuration for how end users of the assistant interact with Auth Web Client, and which client channels are connected. The UI to control those aspects are in Authentication Settings and Channel Settings respectively.

State Structure

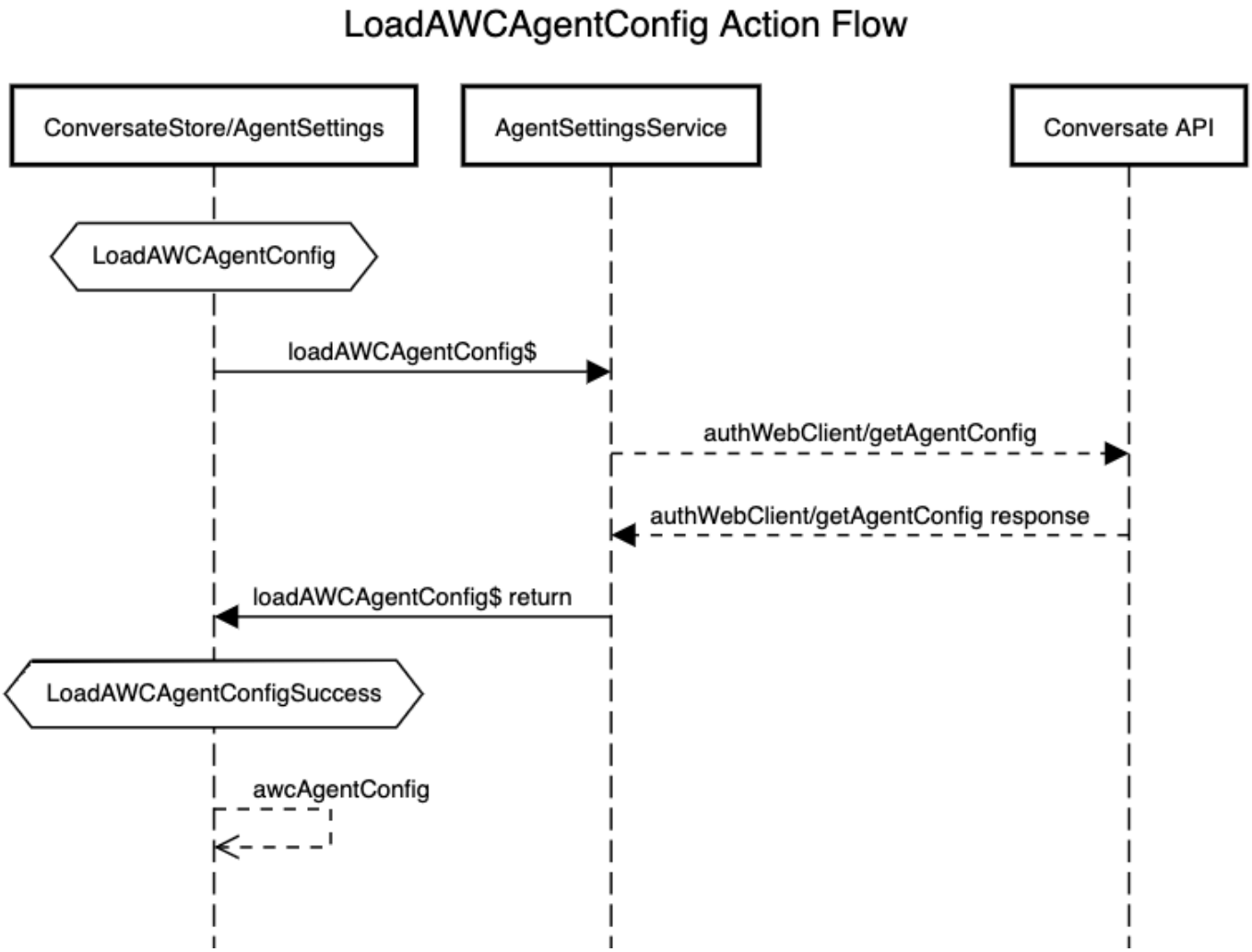
```
{
  awcAgentConfig: {
    id: string;
    config: {
      axosApi?: {
        url: string;
        xApiKey: string;
        facingBrandId: string;
      };
      yodleeApi?:
        | {
        baseUrl: string;
        cobrand: {
          name: string;
          login: string;
          password: string;
        };
        apiVersion: string;
      }
        | {
        samlUrlPath: string;
        yodleeUserDataUrl: string;
        yodleeApiVersion: string;
      };
      homeCUApi?: {
        appId: string;
        org: string;
        sharedKey: string;
        authUrl: string;
```

```
      };
      q2Api?: {
        vendorId: string;
        url: string;
      };
      mxApi?: {
        apiKey: string;
        clientId: string;
        sso: {
          baseUrl: string;
        };
        nexus: {
          baseUrl: string;
        };
      };
      openIdApi?: {
        url: string;
        scopes: string;
      };
      genericOauthRedirectEnabled?: boolean;
      validateConversationalBanking: boolean;
    };
    host: string;
    schema: string;
    backend: string;
    mfaSchema?: string;
    mfaBackend?: string;
    whiteLabel: string;
    allowSourceIdReuse: boolean;
    passcodeEnabledChannels: string[];
    timezone: string;
    recaptcha: {
      v3?: {
        siteKey: string;
        secretKey: string;
        scoreThreshold: number;
      };
      v2?: {
        siteKey: string;
        secretKey: string;
      };
    };
  } | null;
  channelSettings: {
    [key: string]: Channel;
  } | null;
  loaded: boolean;
  loading: boolean;
}
```

Sequence Flows

LoadAWCAgentConfig

Gets the Auth Web Client config from Conversate

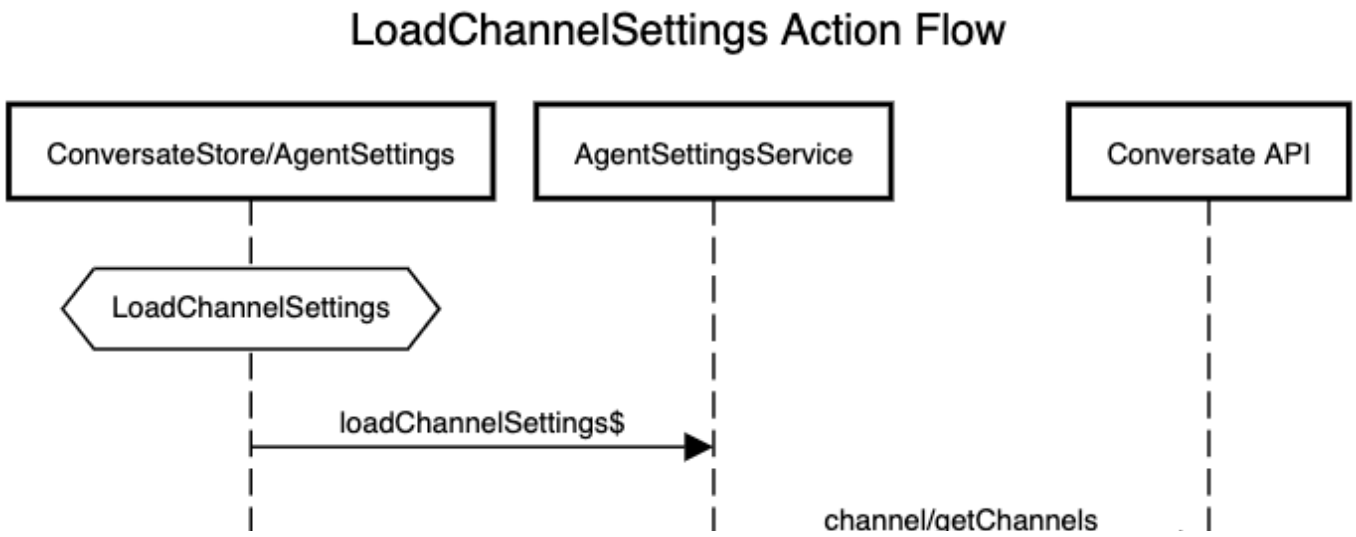


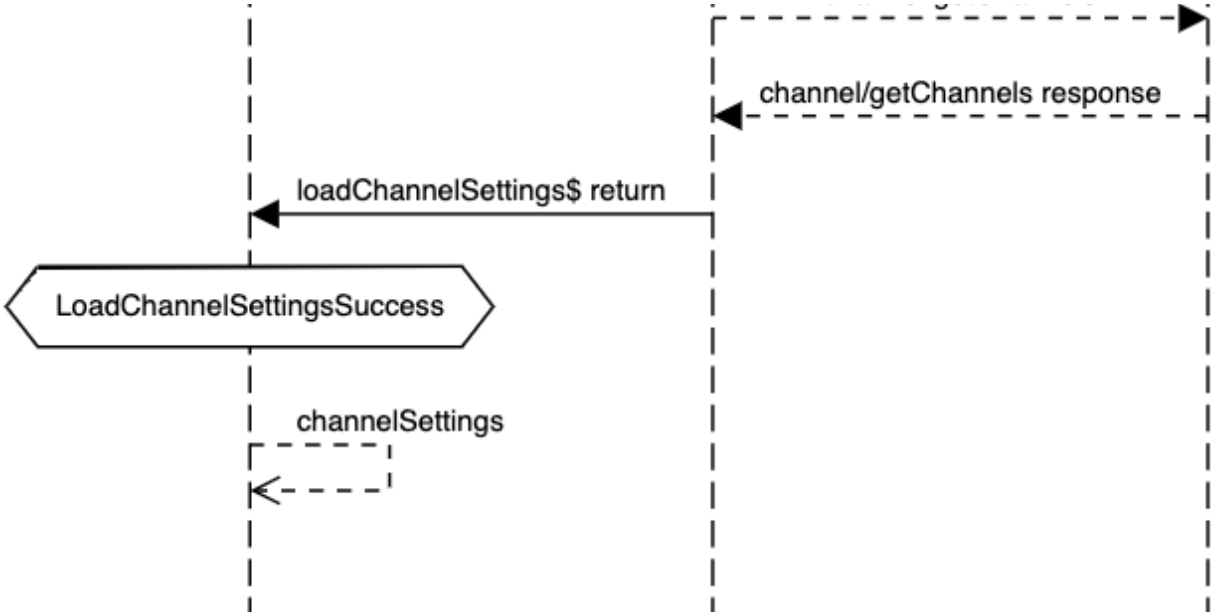
ResetAWCAgentConfig

Sets the `awcAgentConfig` object back to uninitialized state.

LoadChannelSettings

Gets the assistant's channel settings from Conversate



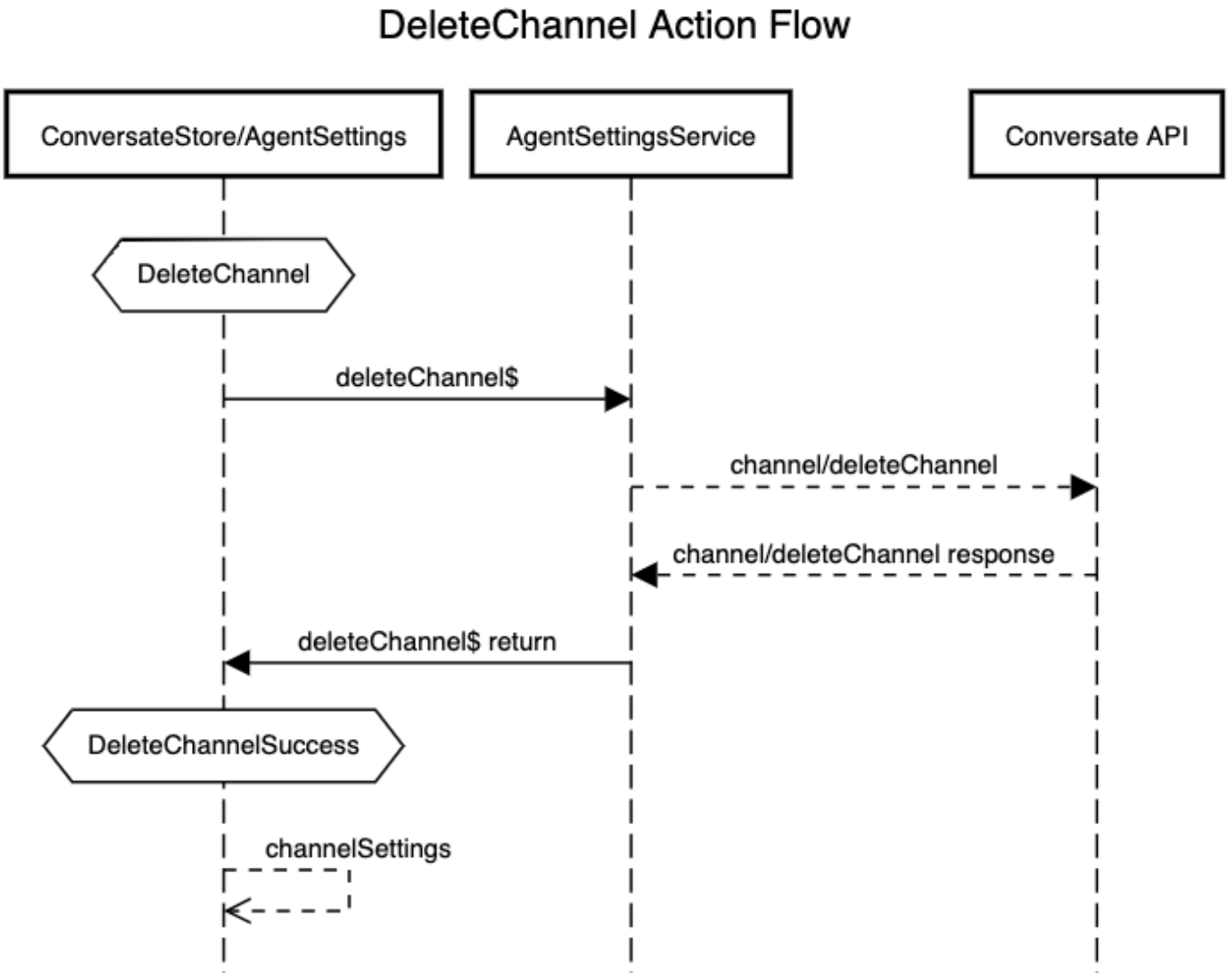


ResetChannelSettings

Resets the channel settings object back to uninitialized state

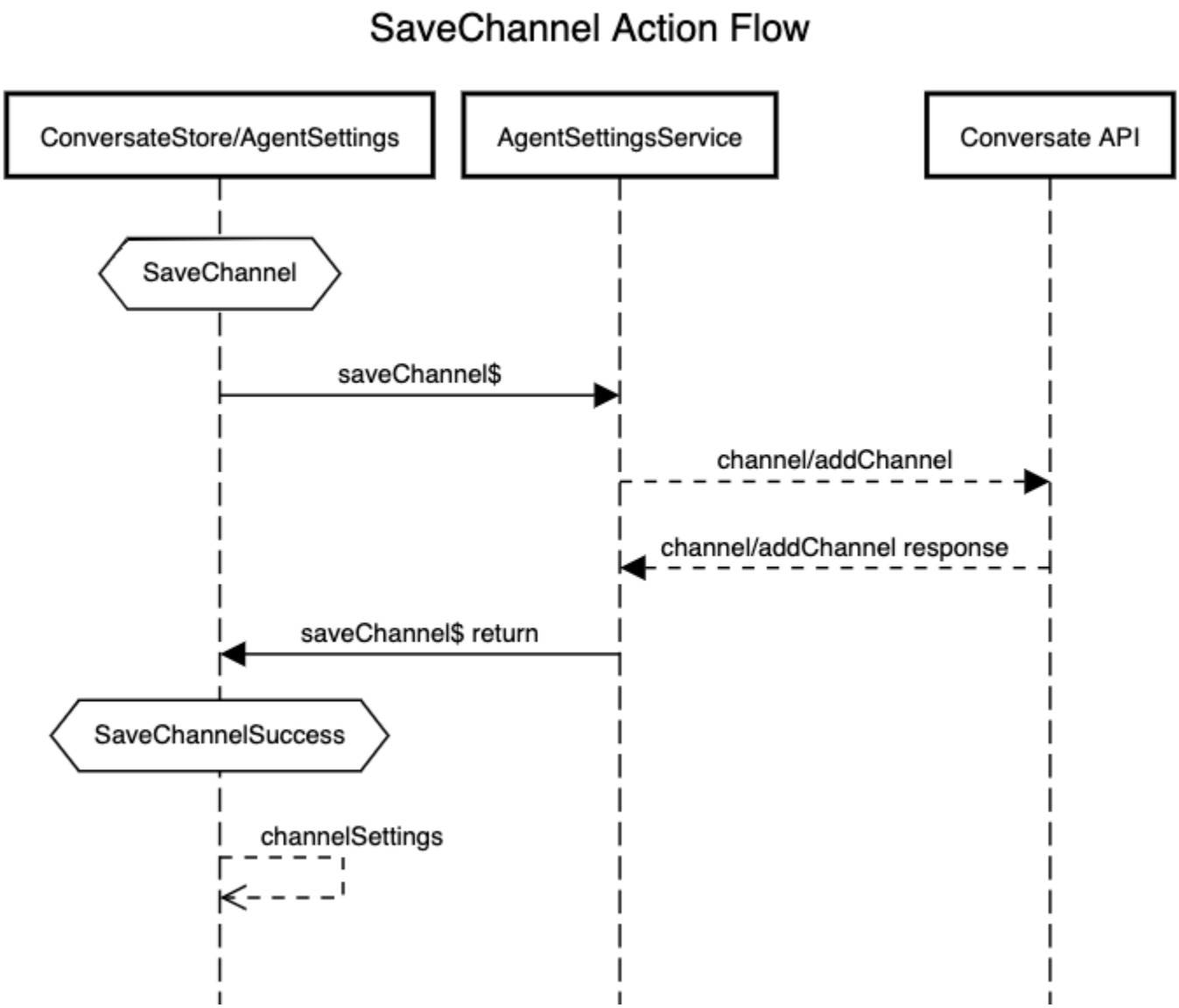
DeleteChannel

Removes a channel configuration from the assistant



SaveChannel

Creates/updates a channel configuration for and assistant



Compliance State

Overview

The compliance state is used to gather and store previous snapshots of the assistant config.

State Structure

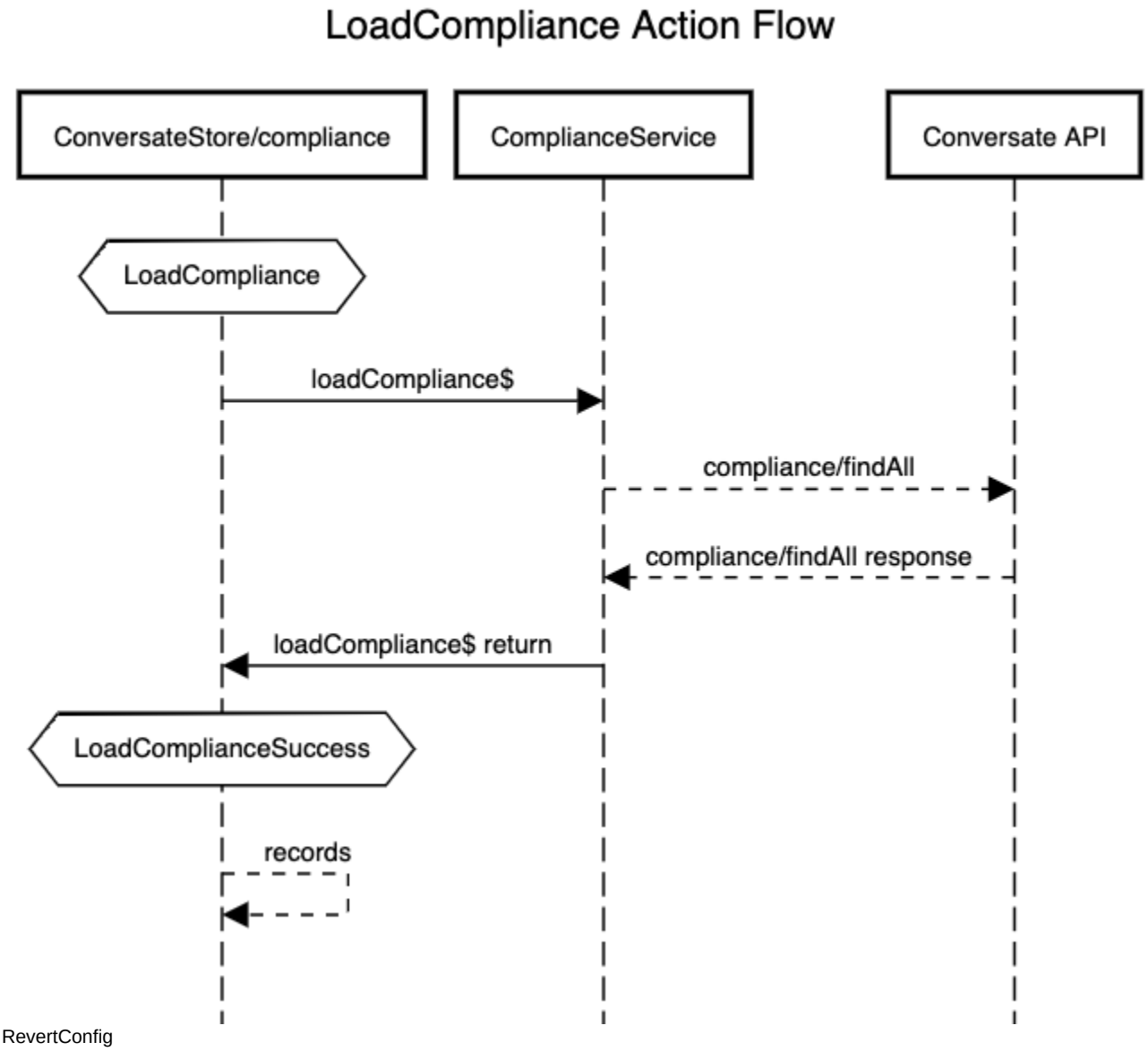
```
{
  records: {
    action: ComplianceRecordAction;
    agentID: UUID;
    created_at: DateString;
    id: UUID;
    meta?: StateIntentMeta | RevertMeta;
```

```
    nluHash: UUID;
    updated_at: DateString;
    user: Email;
  }[] | null;
  loaded: boolean;
  loading: boolean;
}
```

Sequence Flows

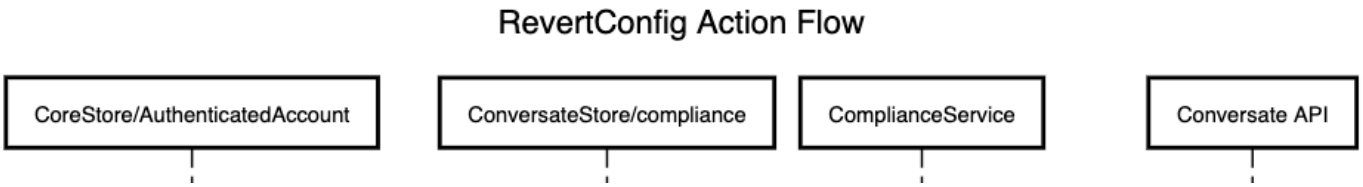
LoadCompliance

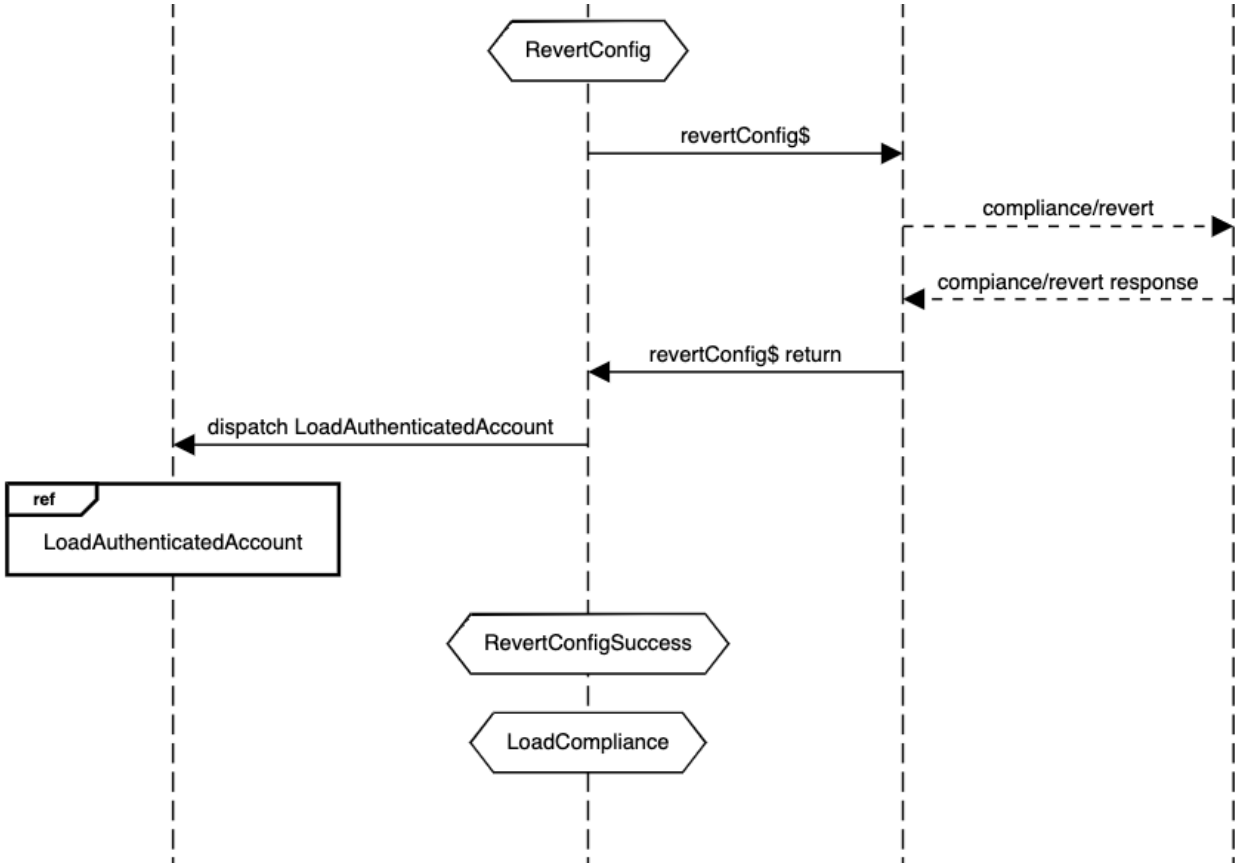
Loads Assistant's Config snapshots from Conversate



RevertConfig

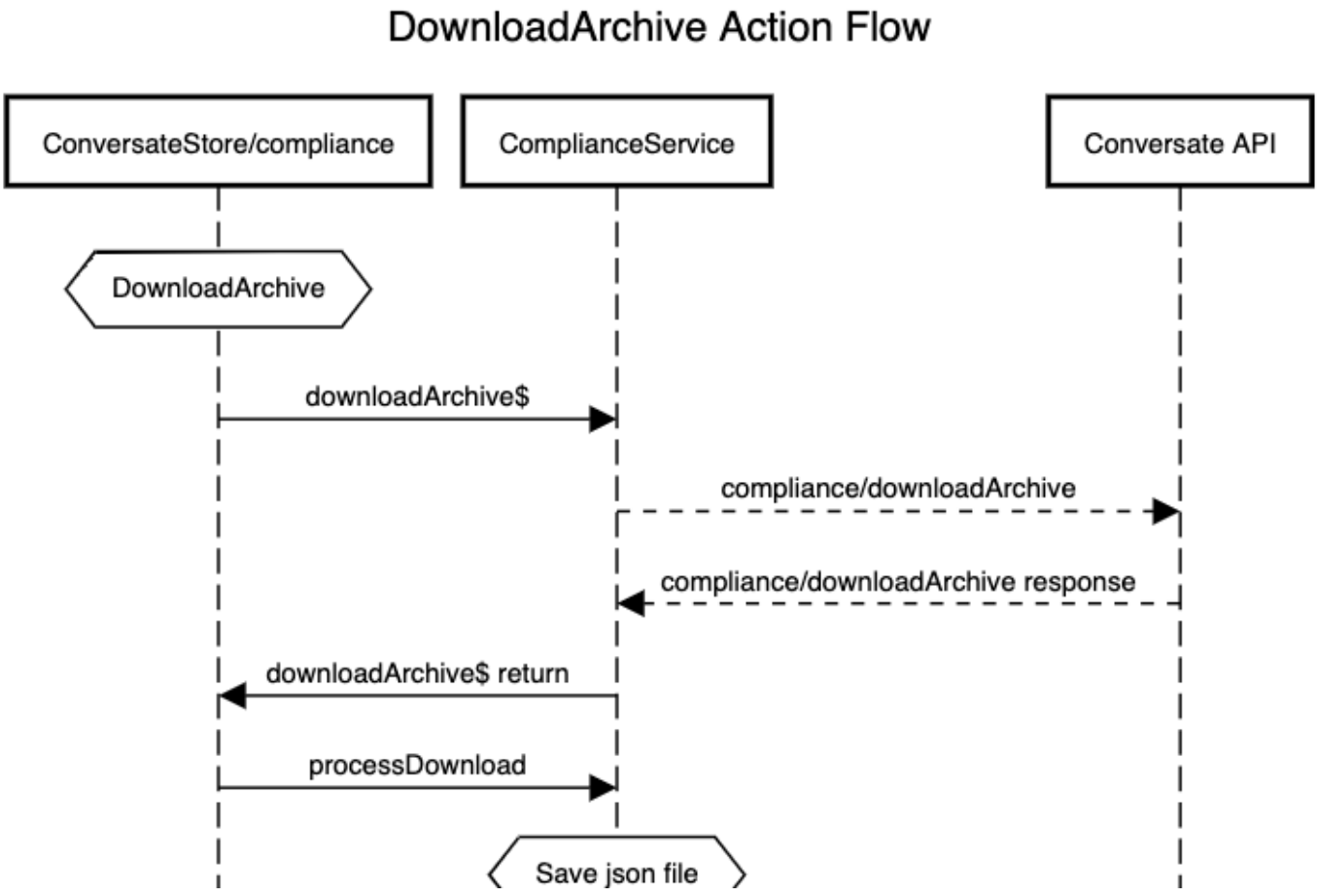
Reverts to a previous config snapshot

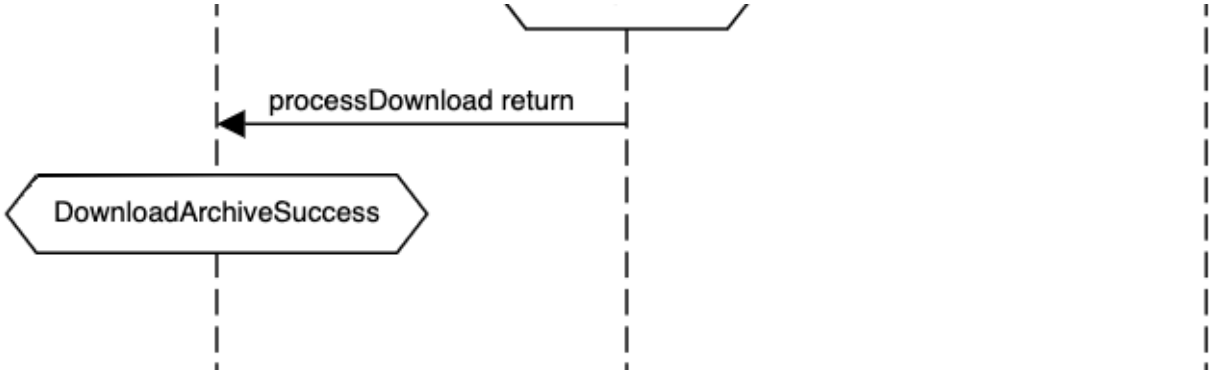




DownloadArchive

Downloads a specific configuration snapshot





CUI Config State

Overview

The CUI Config state is responsible for housing both the raw CUI config for the assistant, as well as the parsed configuration objects.

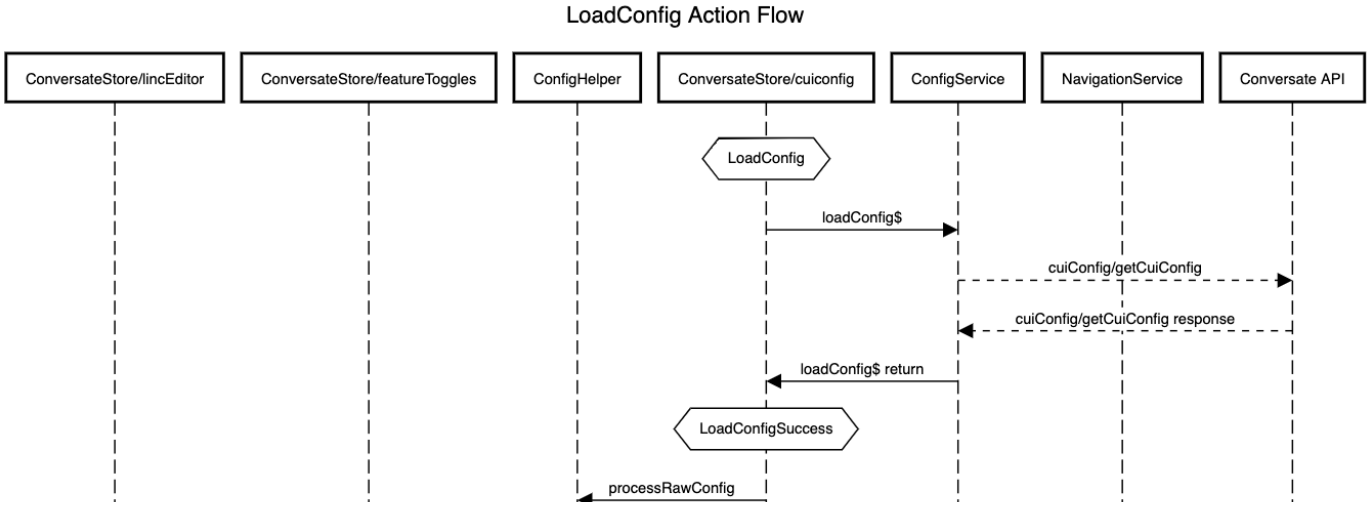
State Structure

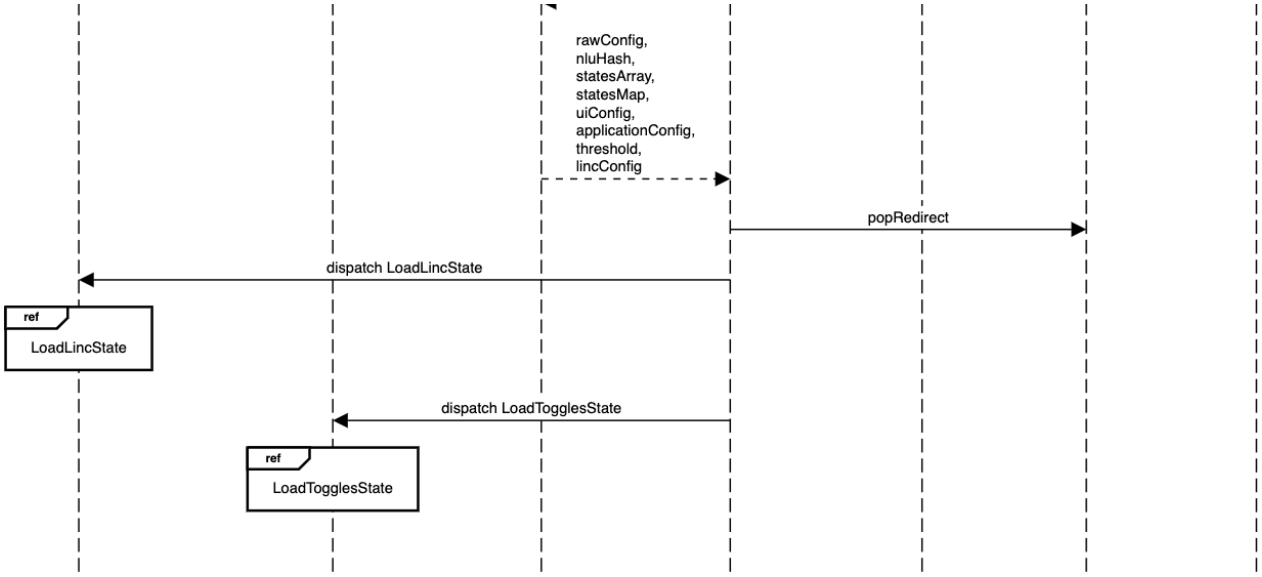
```
{
  rawConfig: RawConfig | null;
  statesMap: StatesMap | null;
  statesArray: State[] | null;
  nluHash: HASH | null;
  uiConfig: UIConfig | null;
  threshold: number | null;
  pendingChanges: boolean;
  lincConfig?: ILincConfig;
  loaded: boolean;
  loading: boolean;
  applicationConfig: ApplicationConfig | null;
}
```

Sequence Flows

LoadConfig

Get Assistant's CUI Config

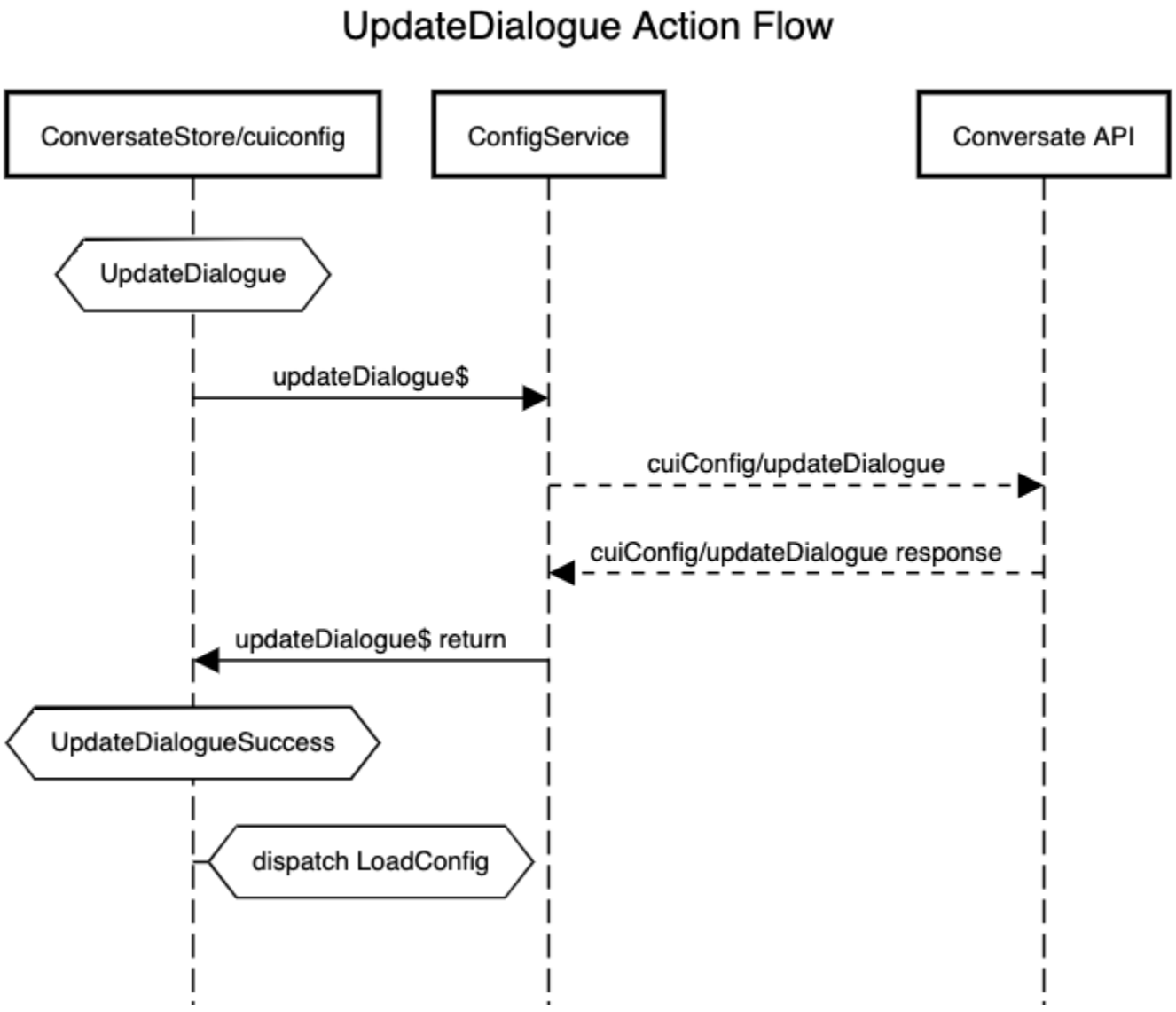




ResetConfig

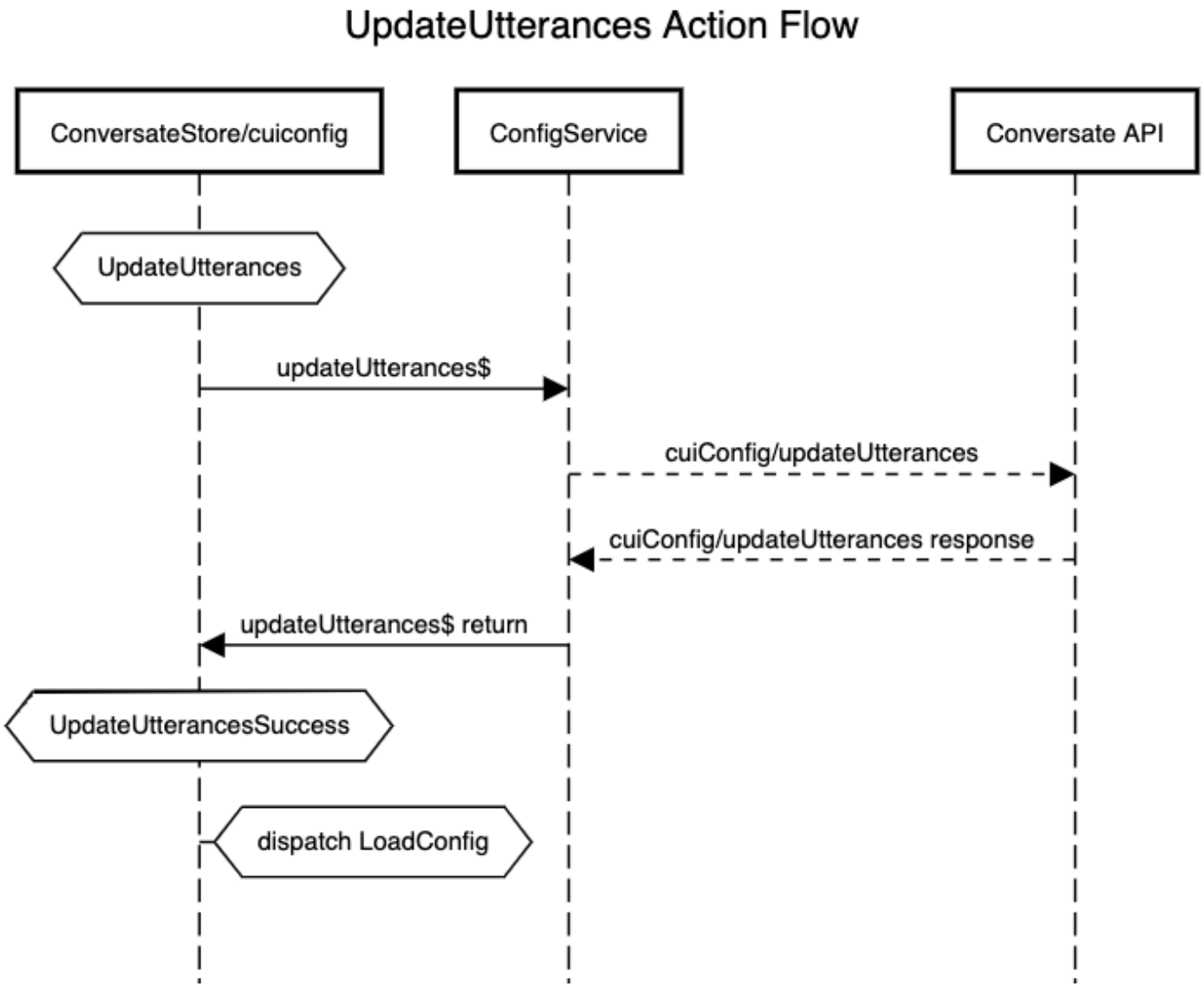
Resets cui config state back to initial state to be re-hydrated by LoadConfig.
UpdateDialogue

Triggered when users update an intent’s responses.



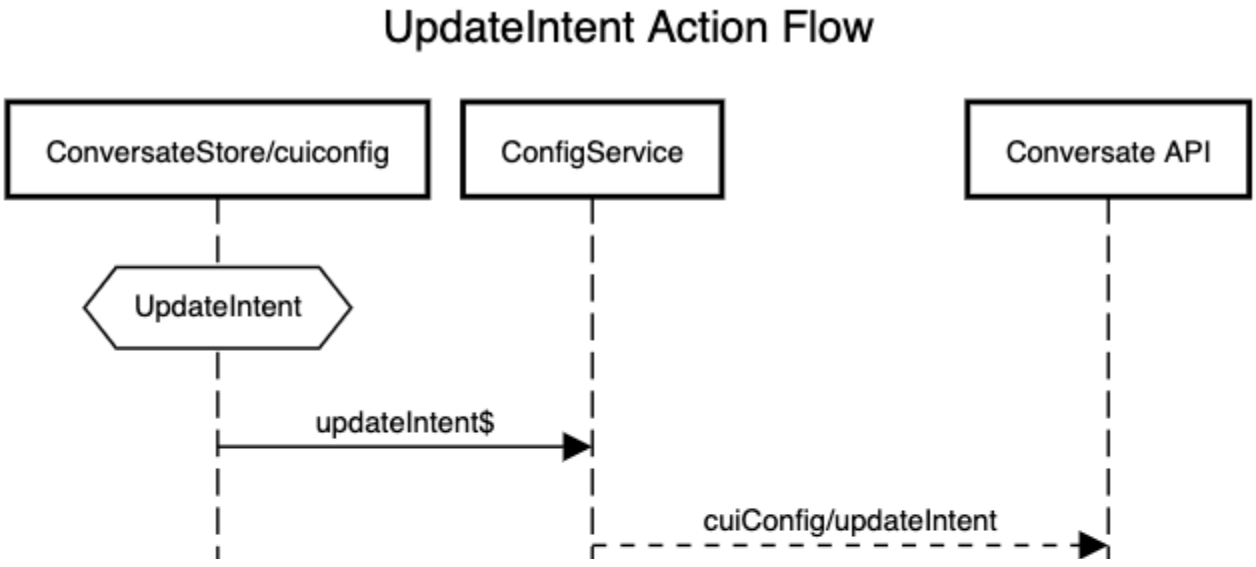
UpdateUtterances

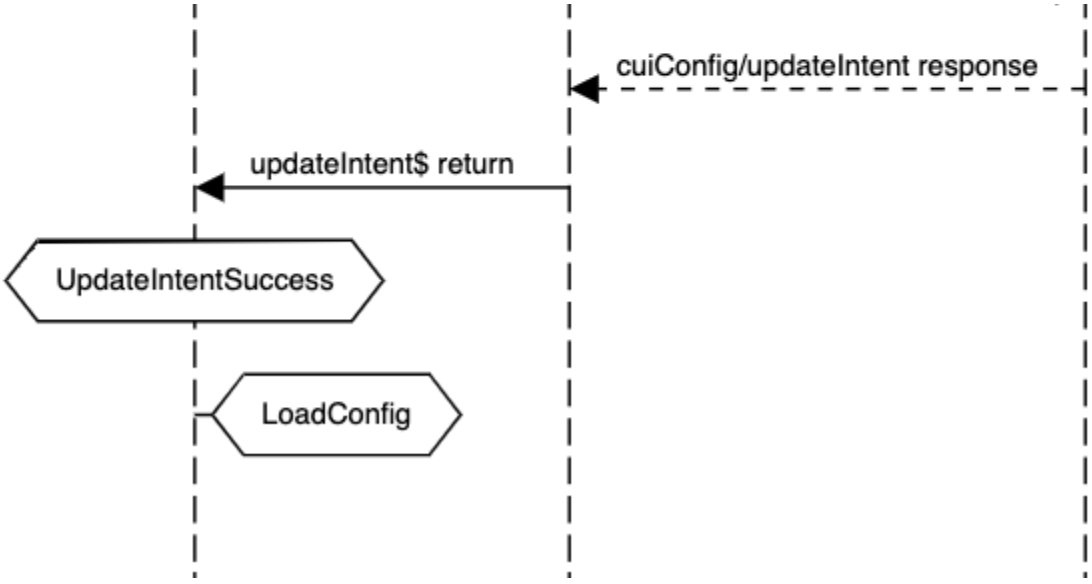
Updates utterance list on an intent



UpdateIntent

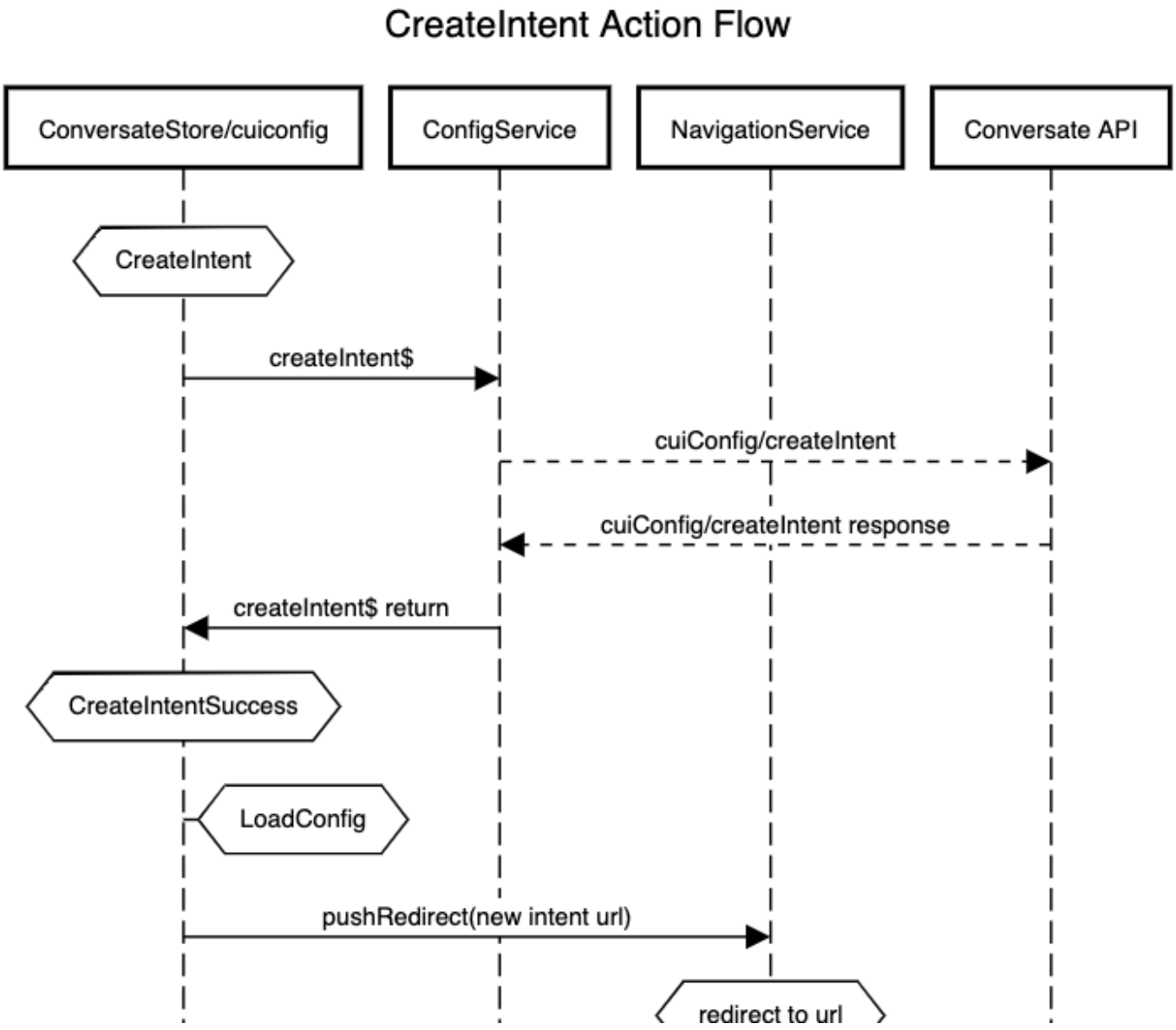
Used to update intent settings such as name or integration service method

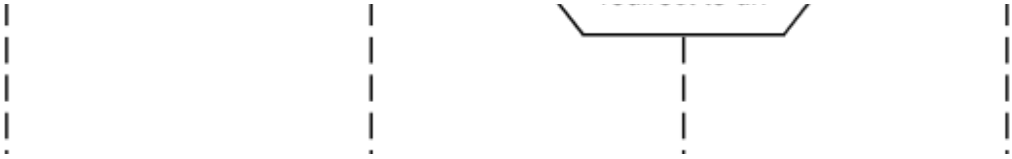




CreateIntent

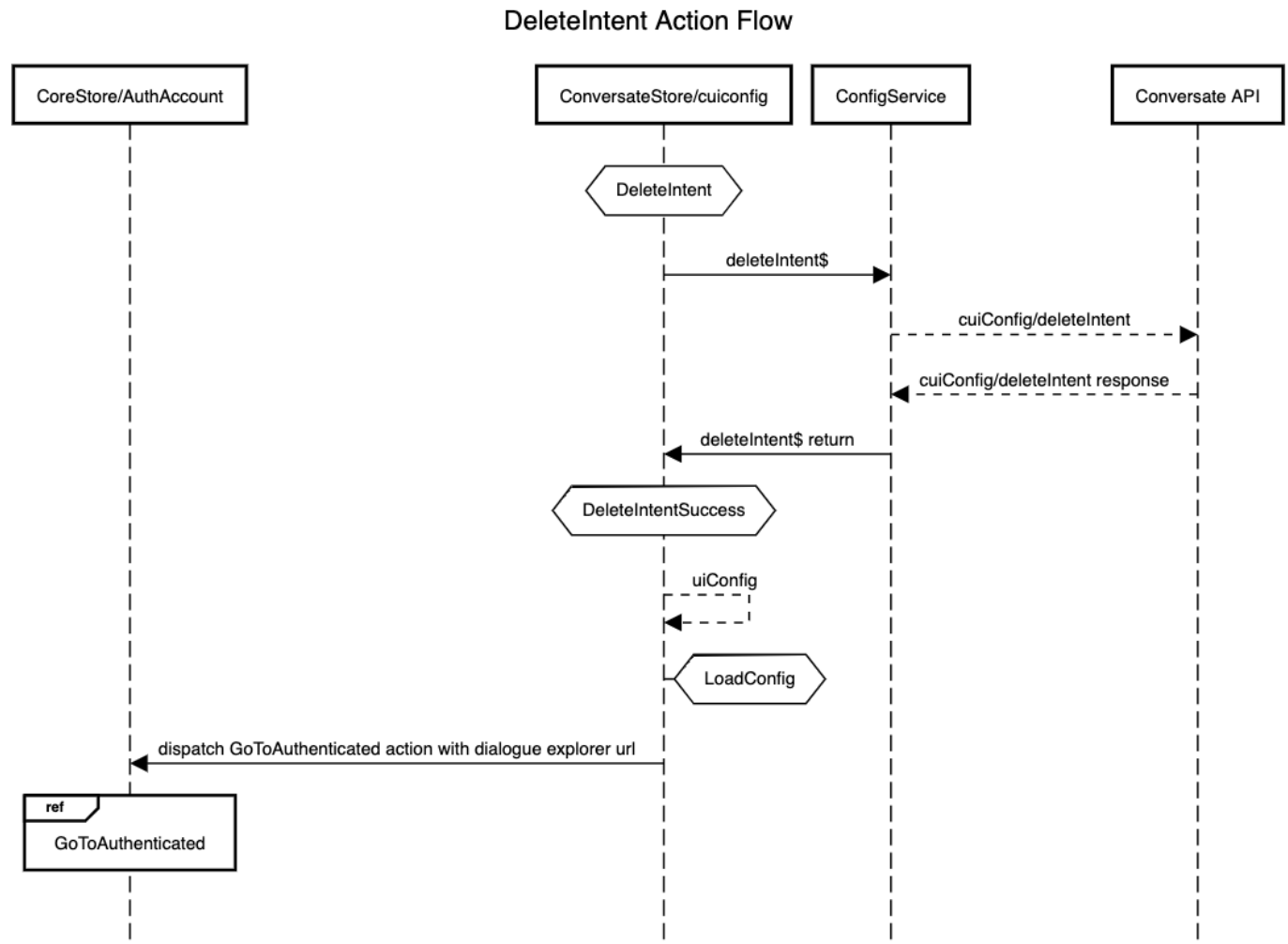
Creates an intent





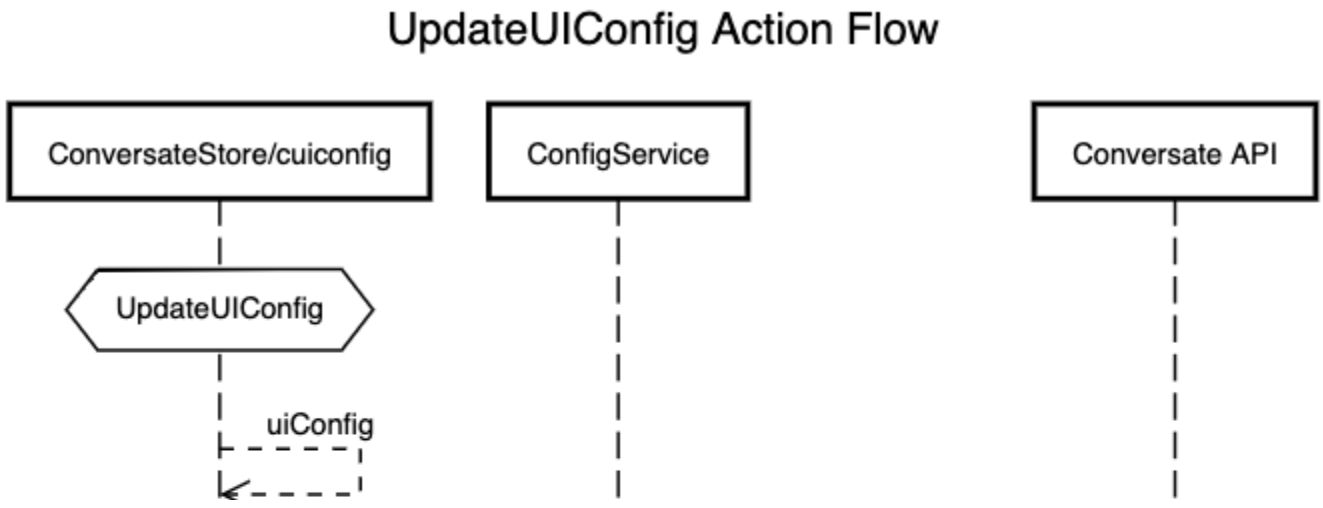
DeleteIntent

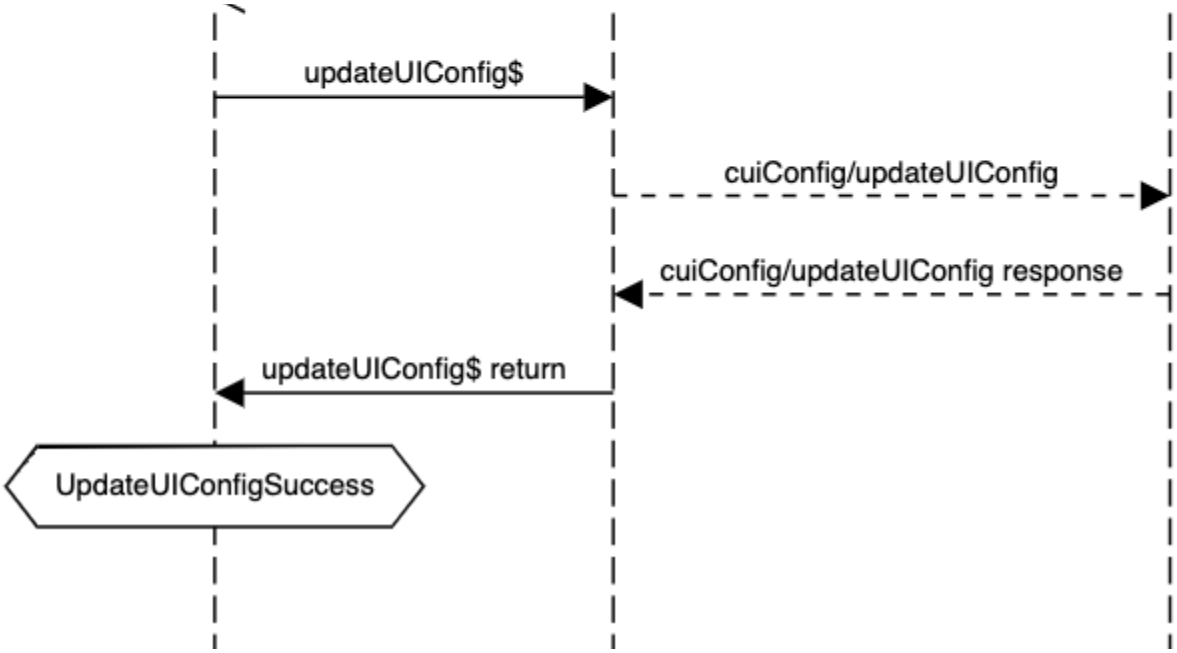
Remove an intent from the assistant



UpdateUIConfig

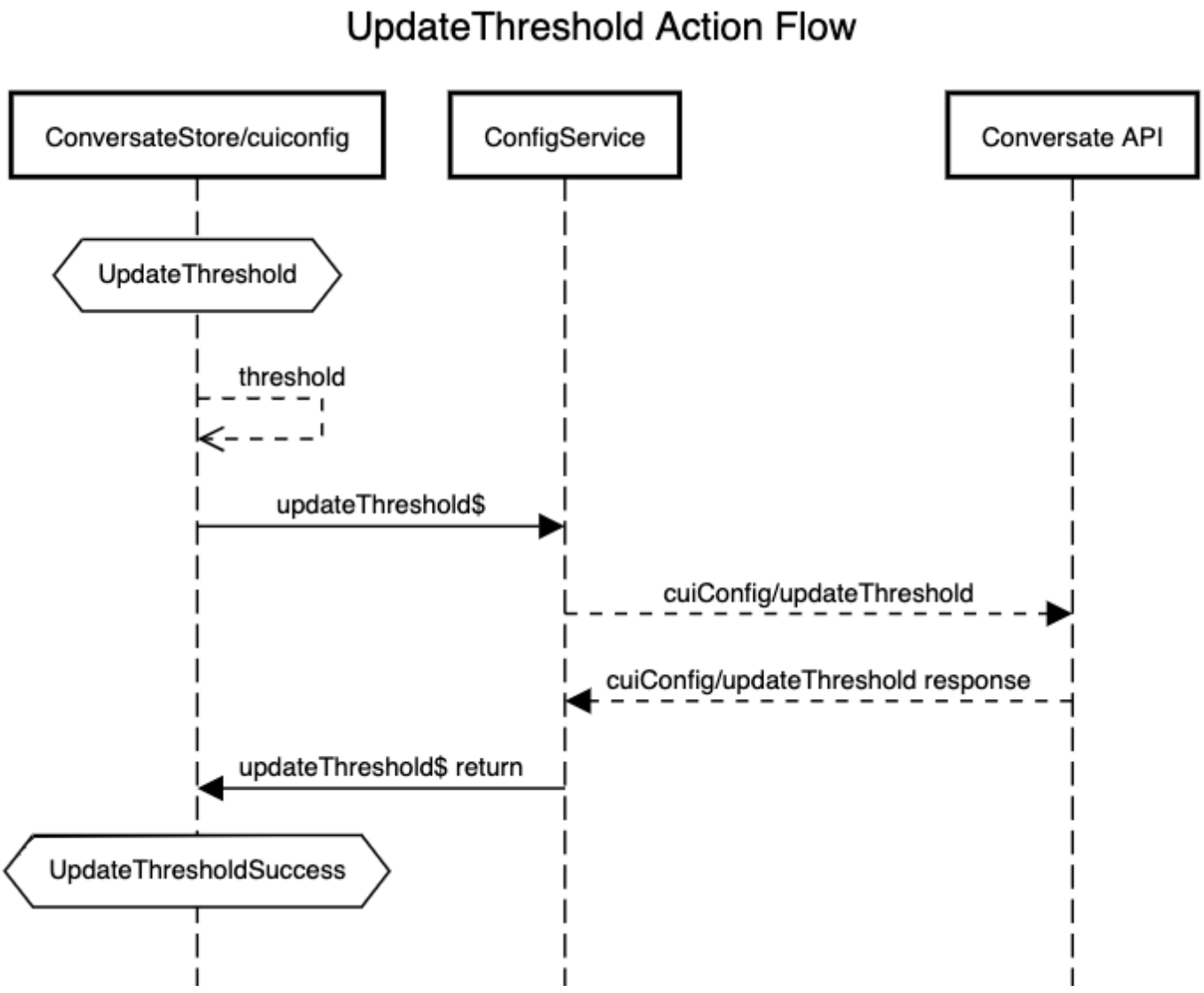
triggered by creating/deleting intent groups, or moving intents





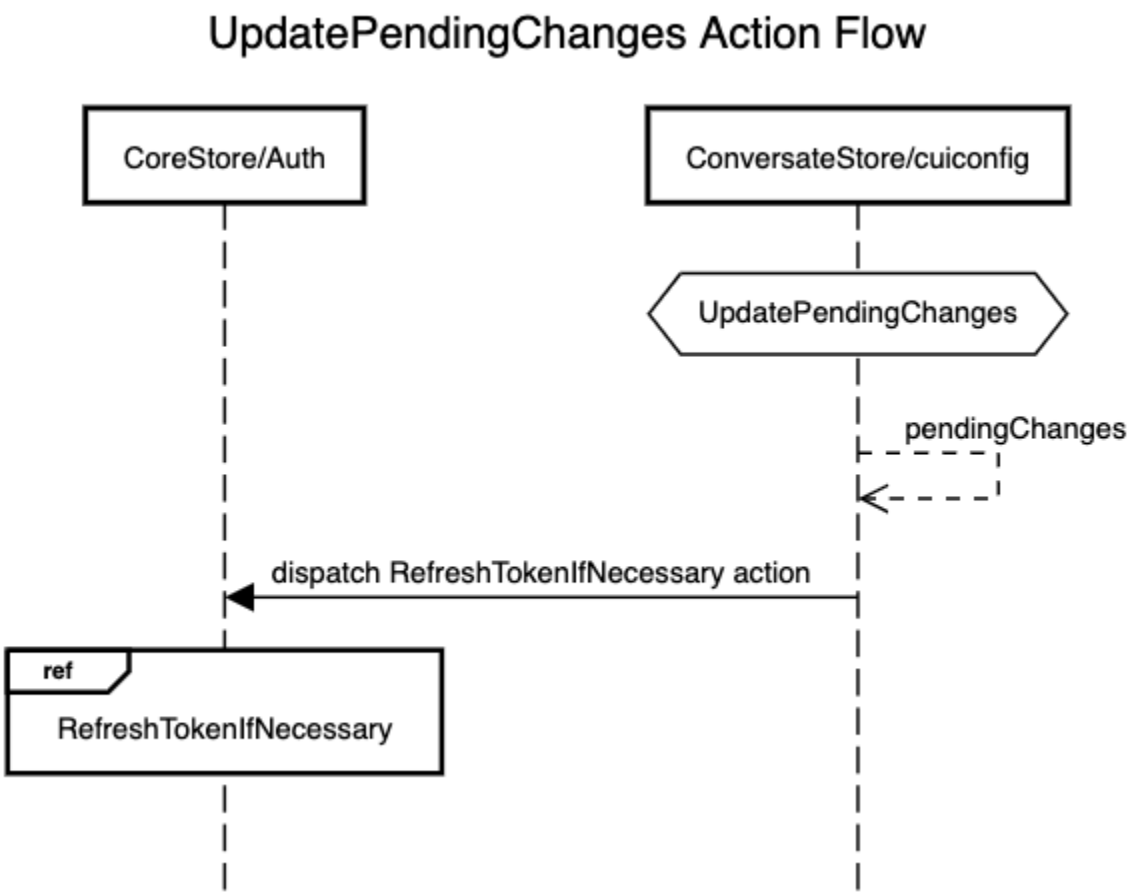
UpdateThreshold

update assistant’s confidence threshold



UpdatePendingChanges

updates pending changes from the dialogue explorer module and triggers a token refresh check



Feature Toggles State

Overview

The feature toggles state is responsible for hydrating and storing the current status of any feature toggle defined in our system

State Structure

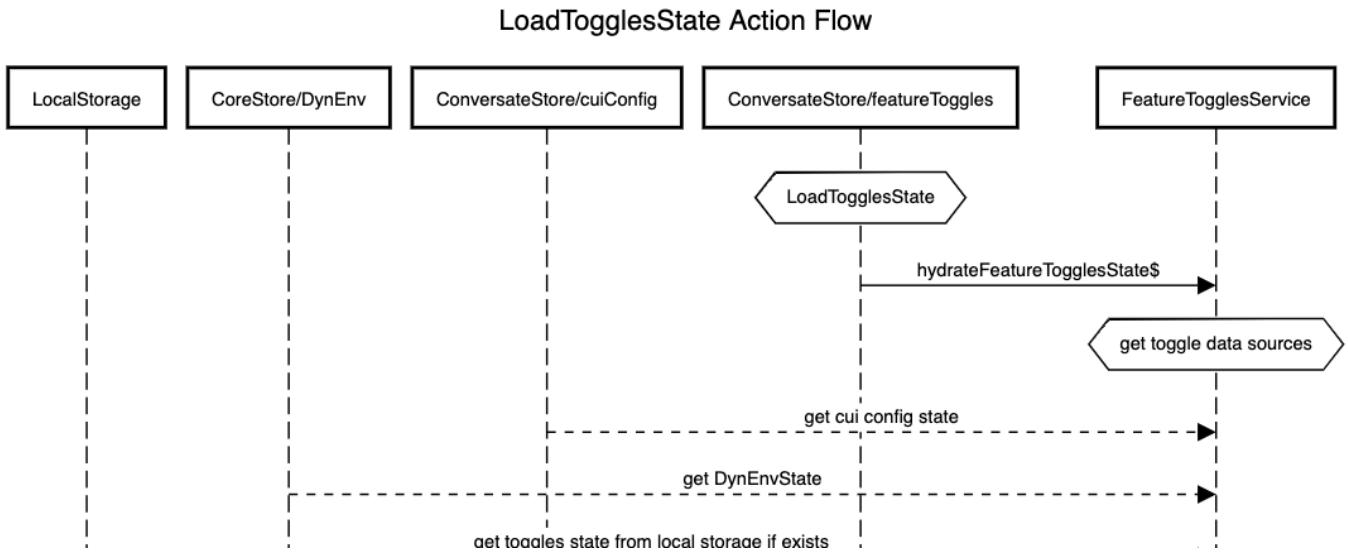
```
{
  loaded: false,
  loading: false,
  toggles: {
  lincEnabled: FeatureToggleState | null;
  disableLincActions: FeatureToggleState | null;
  disableLincEvents: FeatureToggleState | null;
  disableLincSlotTypes: FeatureToggleState | null;
  disableLincIntentToggles: FeatureToggleState | null;
  disableQrLinkedIntent: FeatureToggleState | null;
  // user menu
  disableDocumentation: FeatureToggleState | null;
  // settings
  disableIntegrationServiceTab: FeatureToggleState | null;
```

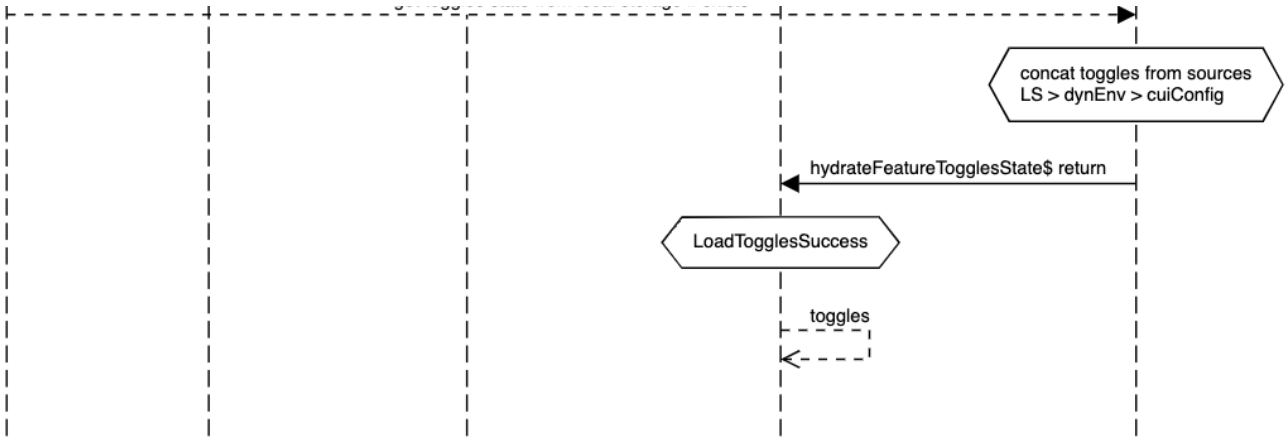
```
  disableGeneralTab: FeatureToggleState | null;
  disableDialogueTab: FeatureToggleState | null;
  disableSettingsChannelTab: FeatureToggleState | null;
  disableSettingsHandoffTab: FeatureToggleState | null;
  disableCampaignManagerTab: FeatureToggleState | null;
  disableTestUserTab: FeatureToggleState | null;
  disableAuthenticationTab: FeatureToggleState | null;
  disableAPIAccessTab: FeatureToggleState | null;
  // specific channels
  disableChannelMessenger: FeatureToggleState | null;
  disableChannelTwilio: FeatureToggleState | null;
  disableChannelAlexa: FeatureToggleState | null;
  disableChannelGoogle: FeatureToggleState | null;
  disableChannelTwitter: FeatureToggleState | null;
  disableChannelEmbeddedMobile: FeatureToggleState | null;
  disableChannelEmbeddedWeb: FeatureToggleState | null;
  disableChannelGlia: FeatureToggleState | null;
  disableChannelFive9: FeatureToggleState | null;
  disableChannelPopio: FeatureToggleState | null;
  disableChannelCustom: FeatureToggleState | null;
  // Account Overview
  disableAgentAddDelete: FeatureToggleState | null;
  disableUserAddDelete: FeatureToggleState | null;
  disableApiKeysTab: FeatureToggleState | null;
 },
}

interface FeatureToggleState {
  key: FeatureToggleKeysEnum;
  value?: FeatureValue;
  source?: 'User Override' | 'CUI Config' | 'Environment Variable';
}
```

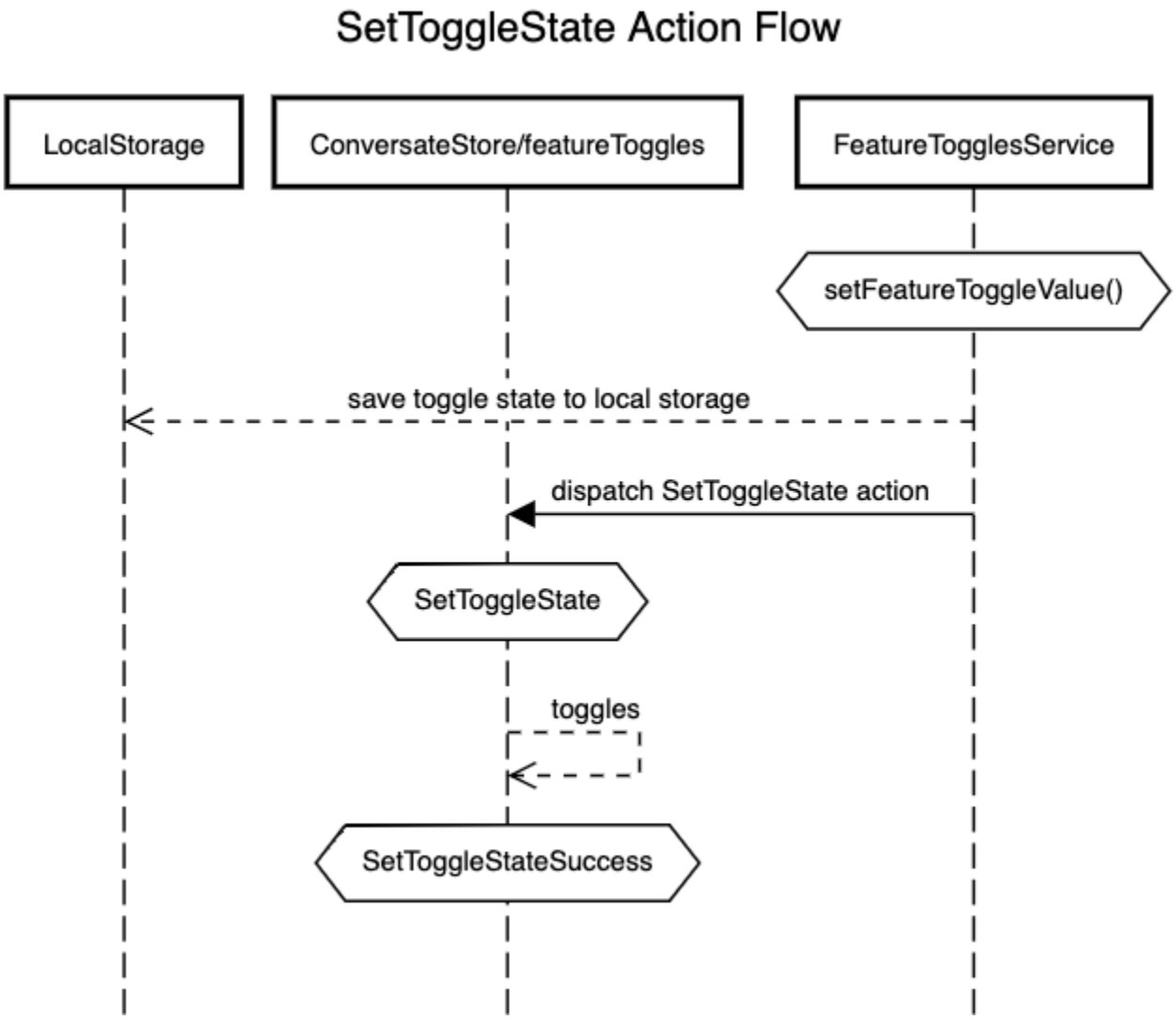
Sequence Flows

LoadTogglesState

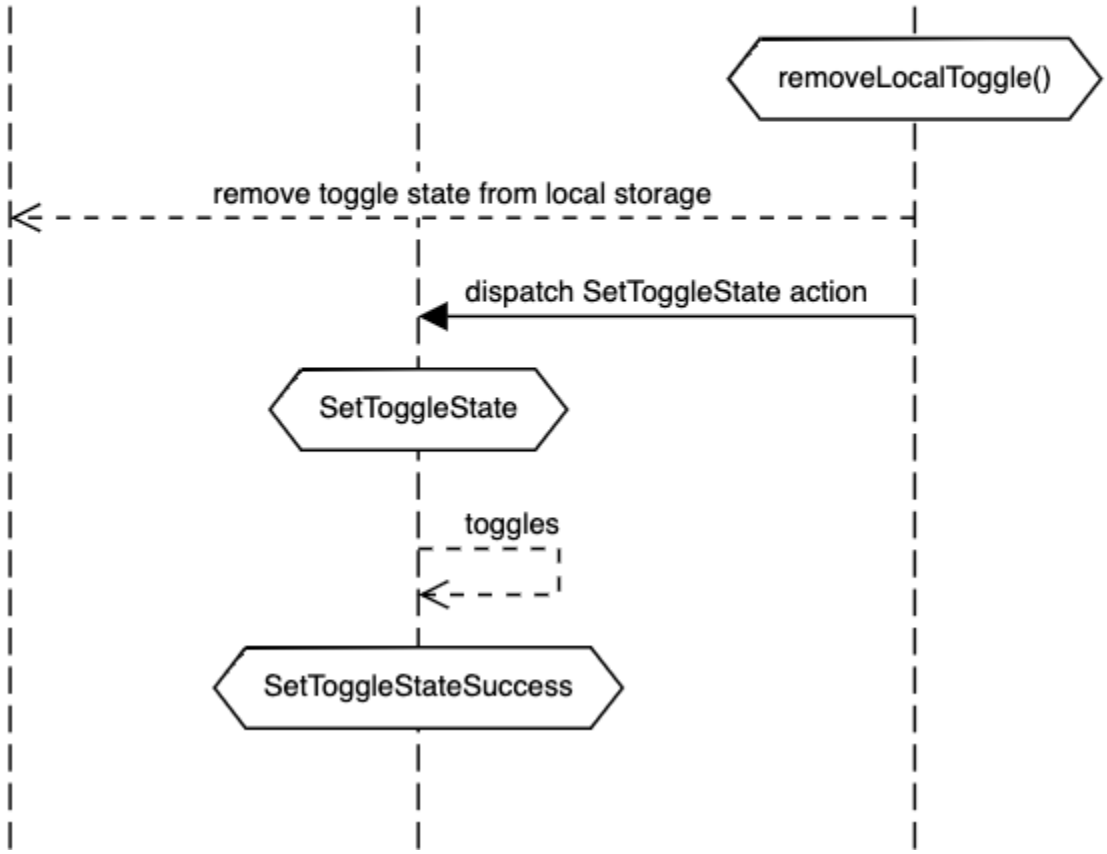




SetTogglesState



RemoveLocalToggle



Integration Service State (ConfiguredServices)

Overview

The integration service state is responsible for housing all information regarding an assistant's configured integration services.

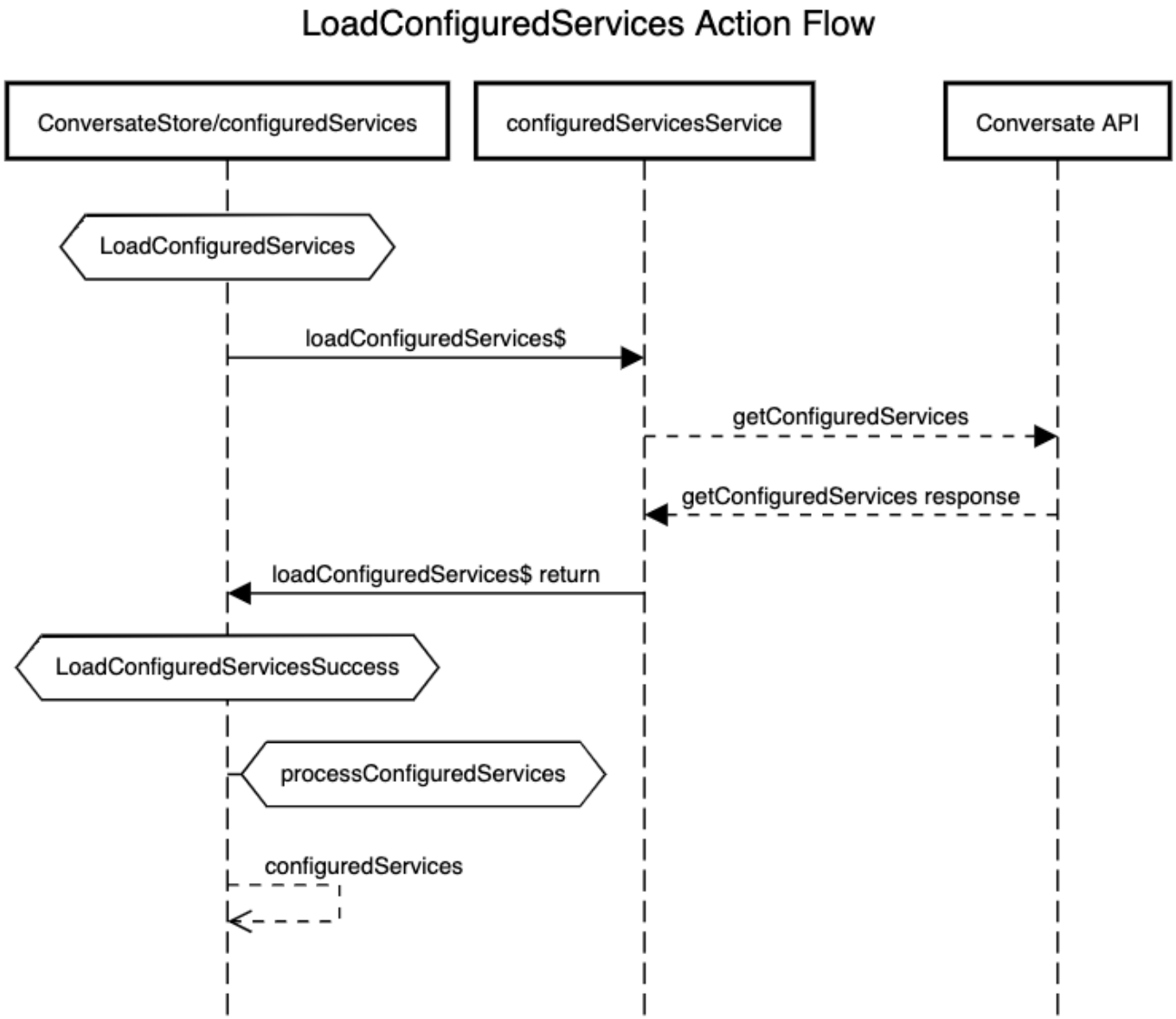
State Structure

```
{
  integrationServiceArray: {
    integrationServiceConfigSchema: IntegrationServiceConfigSchema |
null;
    form?: FormGroup;
    integrationServiceConfig: IntegrationServiceConfigWrapper | null;
    base: URLString;
    intentMethodsArray: IntentMethod[];
    intentMethodsMap: IntentMethodMap;
    isEnvVar: boolean;
    name?: string;
    test: URLString;
    url: URLString;
    online: boolean;
    serviceError: boolean;
  }[] | null;
  integrationServiceMap: IntegrationServiceMap | null;
  default: string | null;
  loaded: boolean;
```

```
  loading: boolean;
}
```

Sequence Flows

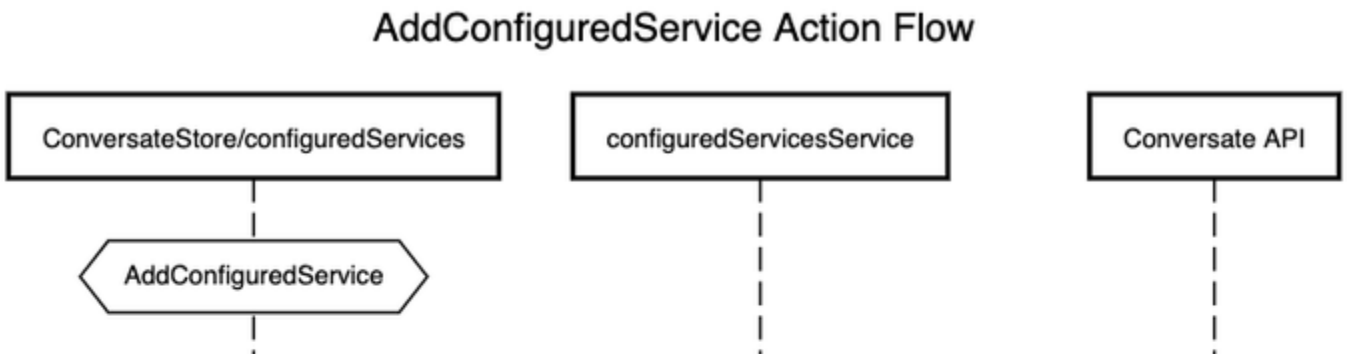
LoadConfiguredServices

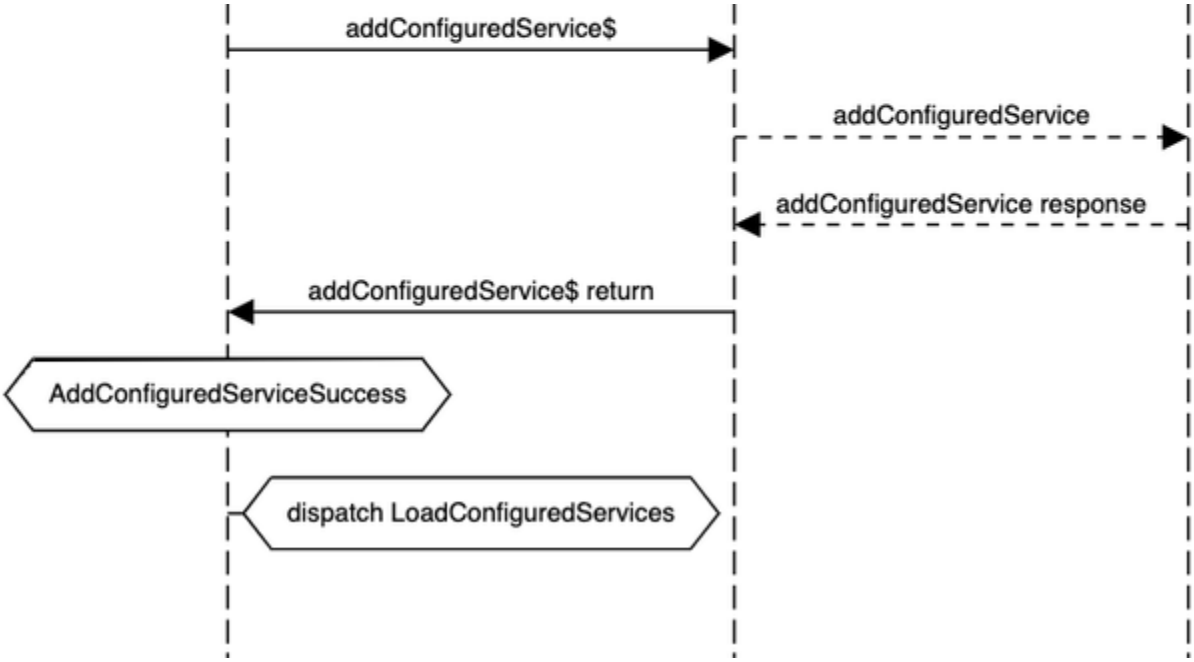


ResetConfiguredService

resets the integration service state back to uninitialized state.

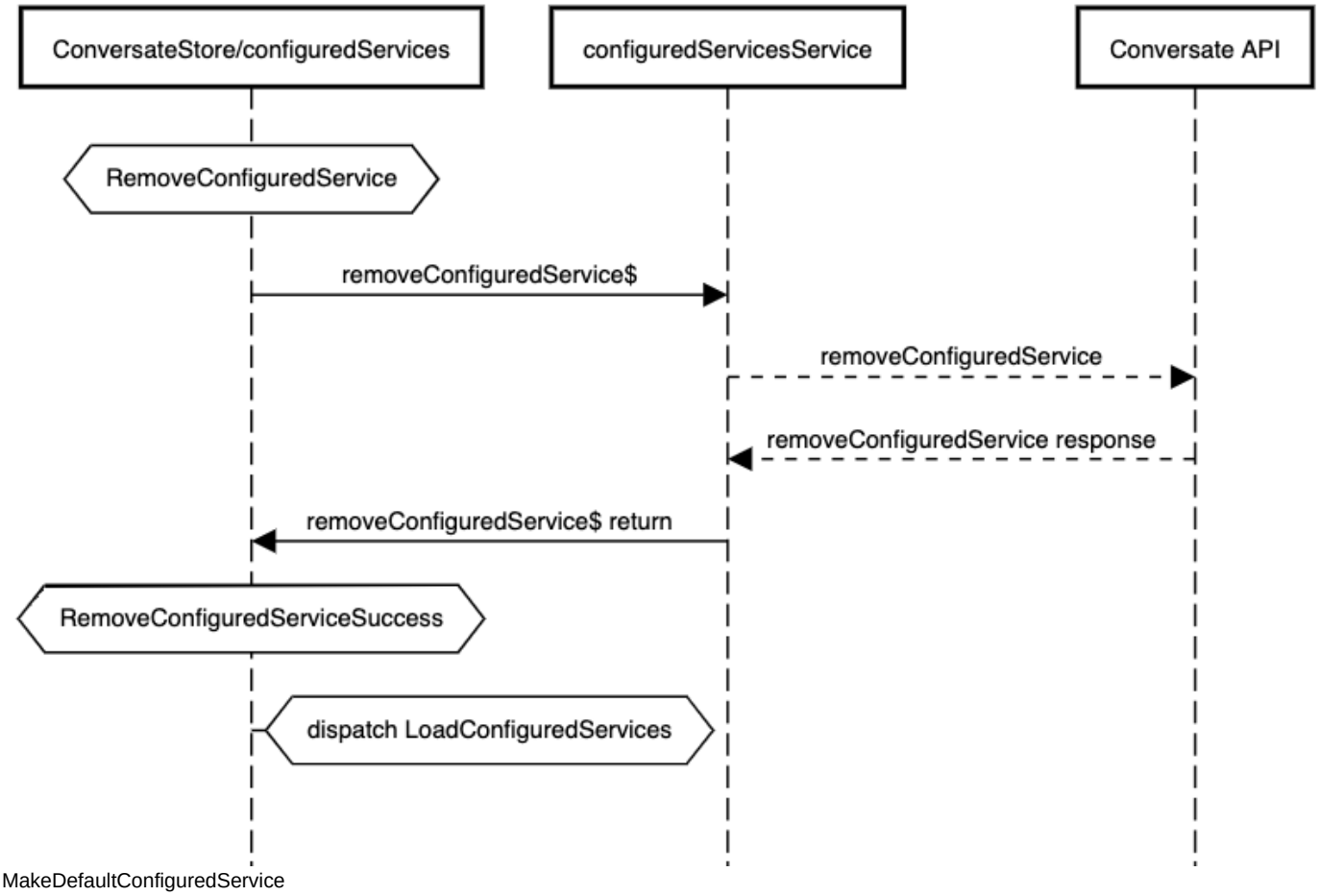
AddConfiguredService





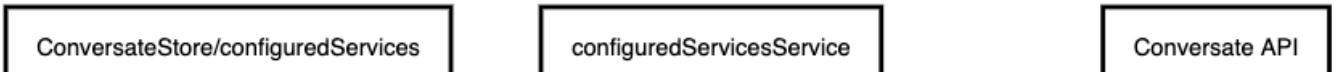
RemoveConfiguredService

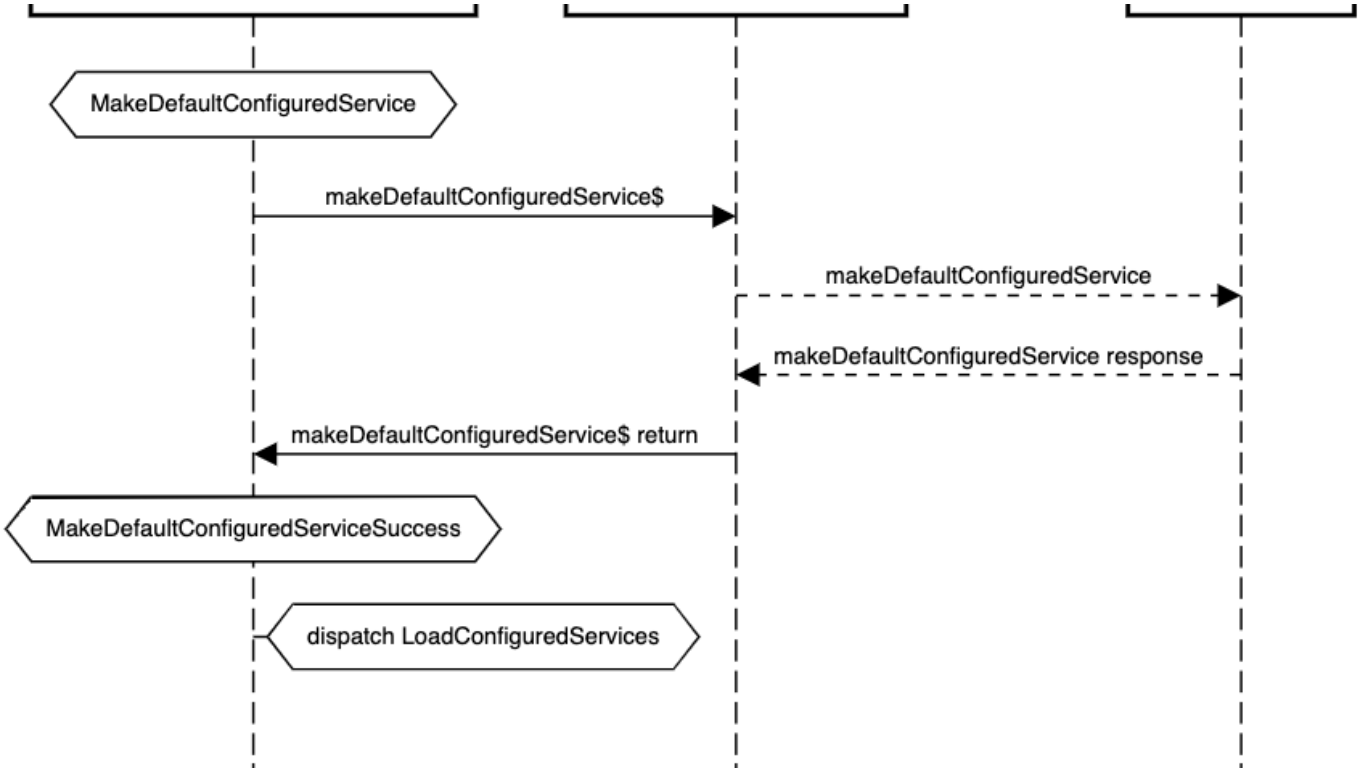
RemoveConfiguredService Action Flow



MakeDefaultConfiguredService

MakeDefaultConfiguredService Action Flow





Linc Editor State (Deprecated)

Overview

The linc editor state is used to store objects needed to run the deprecated linc editor module, including the local working copy of the linc config for the assistant.

Navigation State

Overview

The navigation state is responsible for storing information about the top nav UI.

State Structure

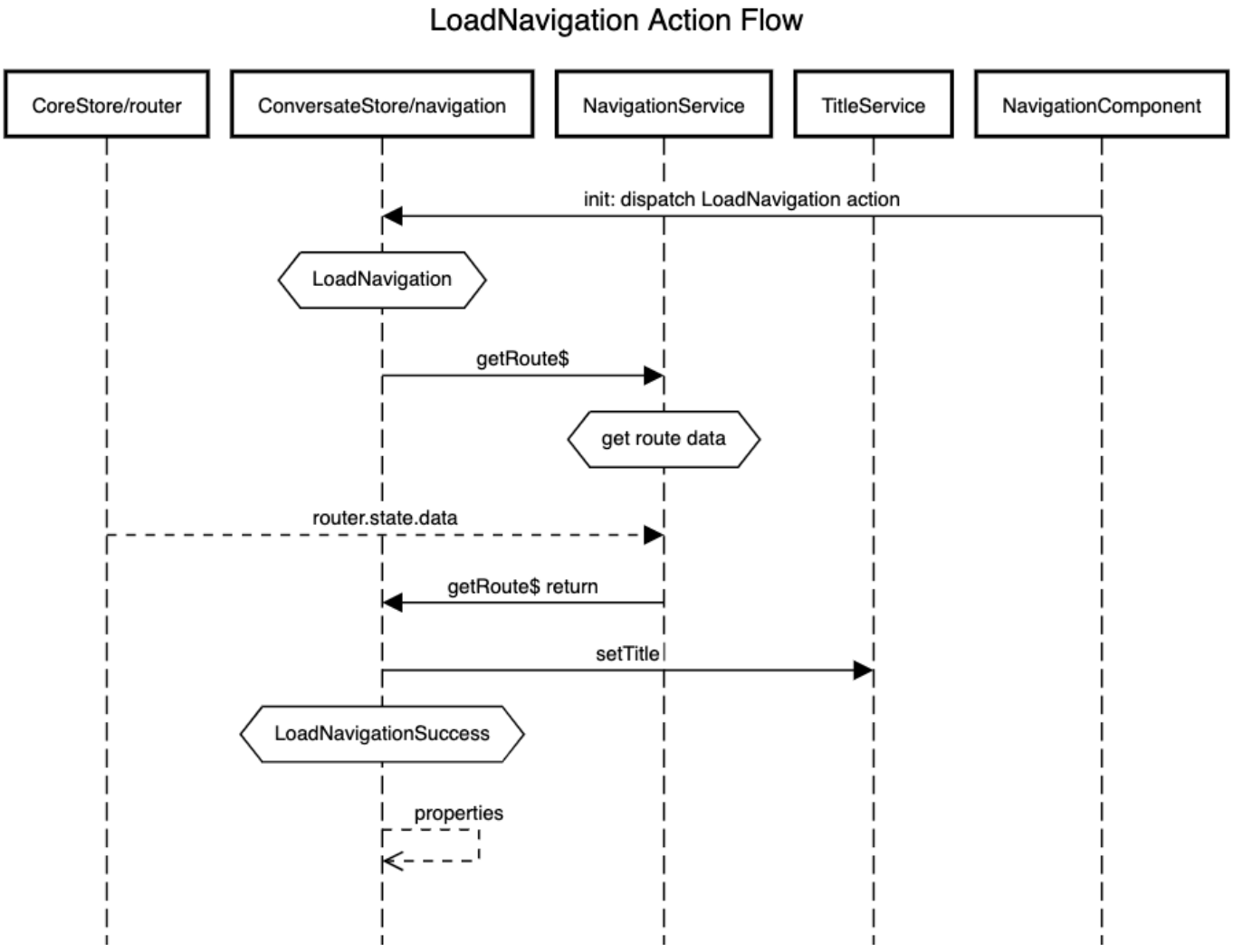
```
{
  user: {
    id: UUID;
    account: UUID;
    namespace: string;
    email: string;
    exp: number;
    hasLoggedOut: boolean;
    isLockedOut: boolean;
    error: RESTError | null;
  } | null;
  properties: {
    subNavItems: {
      label: string;
      link: ChangeTabString;
      base?: SubNavBase;
```

```
      renderConditionFn?: (ftService: FeatureToggleService) => boolean;
    }[];
    showNavigation: boolean;
    showSubNav: boolean;
    showAgents: boolean;
    activeMenu: string;
  } | null;
  loaded: boolean;
  loading: boolean;
}
```

Sequence Flows

LoadNavigation

used to hydrate store variables used in the navigation component. Navigation Component dispatches this action on it's init process



Supervisor State

Overview

The supervisor state houses data needed to run the Supervisor module, including conversations, and the active conversation.

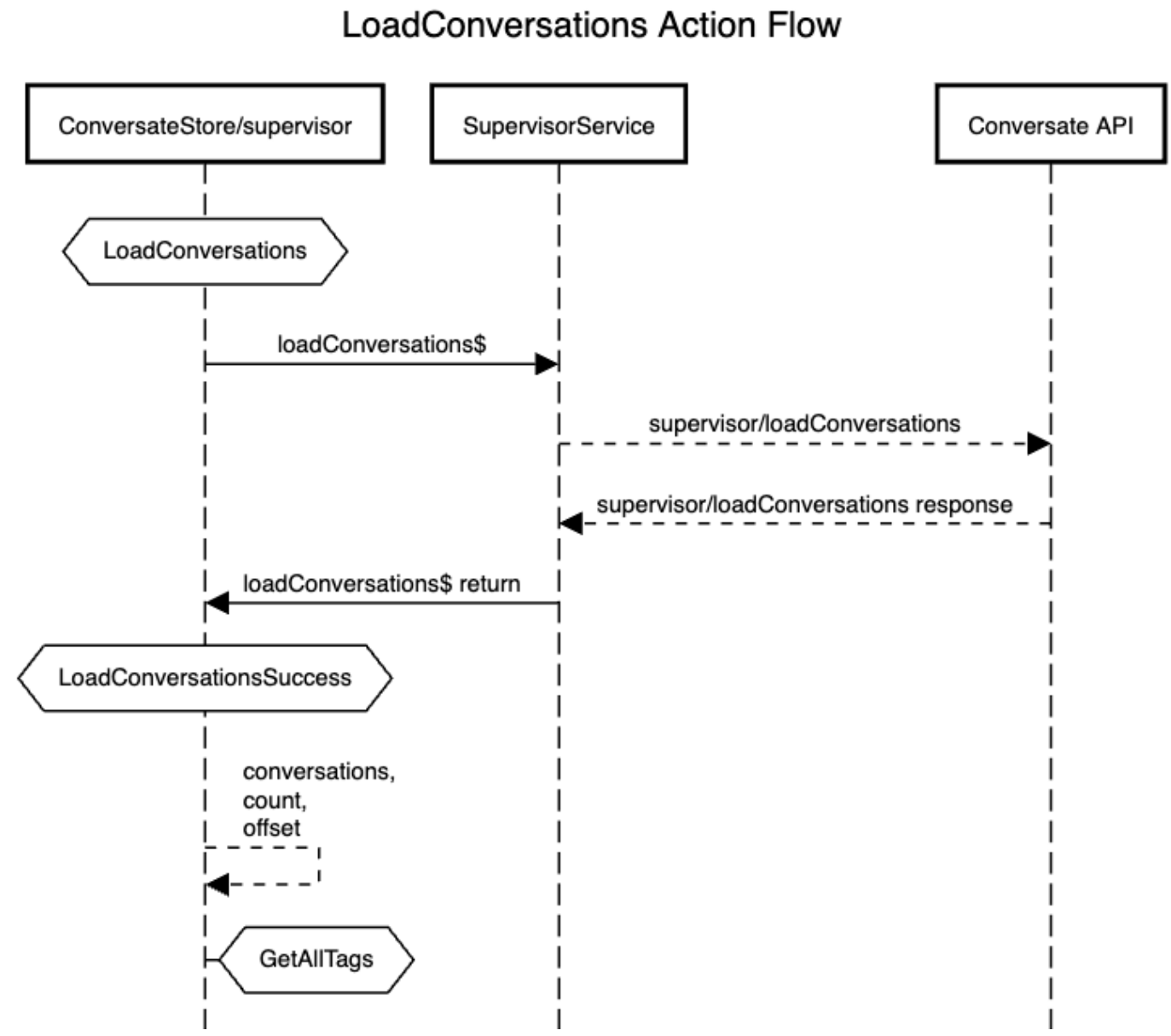
State Structure

```
{
  conversations: Array<SupervisorConversation> | null;
  selectedConversation: SupervisorConversation | null;
  count: number;
  offset: number;
  loaded: boolean;
  loading: boolean;
  messageLogs: SupervisorMessageLogs;
  currentMessage: ConversationMessage | null;
  tags: Array<string>;
}
```

Sequence Flows

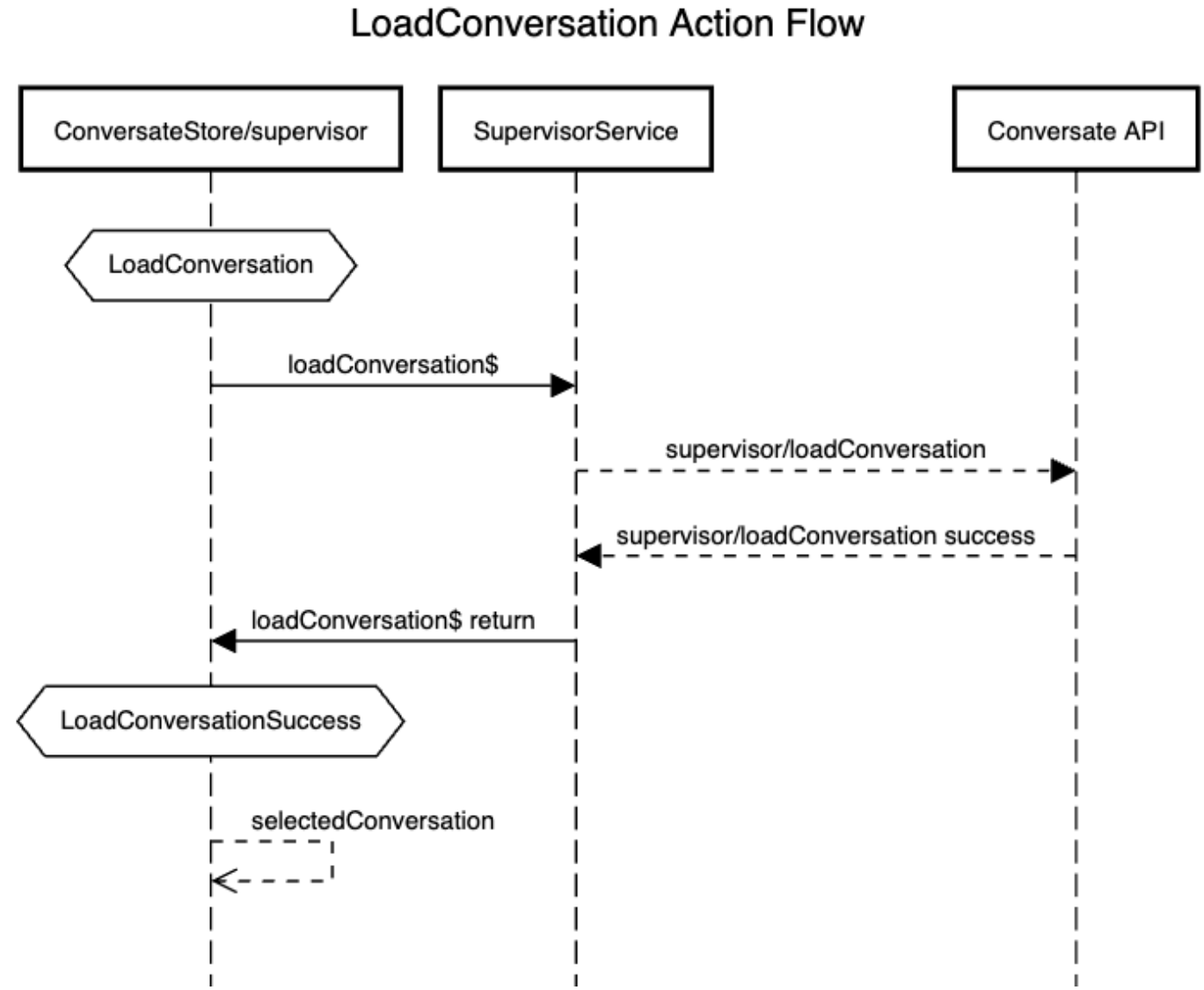
LoadConversations

retrieves conversations for the assistant from Conversate API



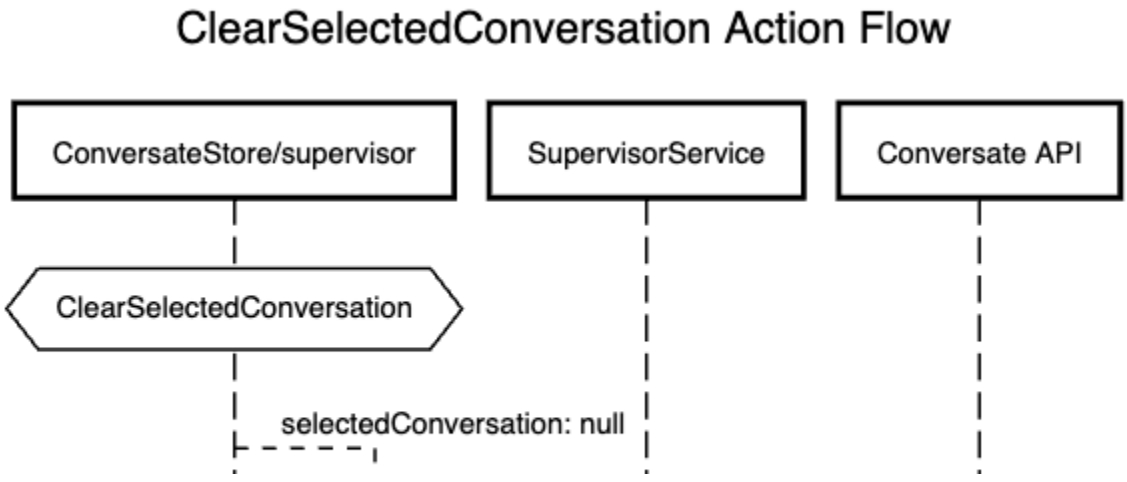
LoadConversation

load a single conversation's messages, and make it the active conversation in the viewer.



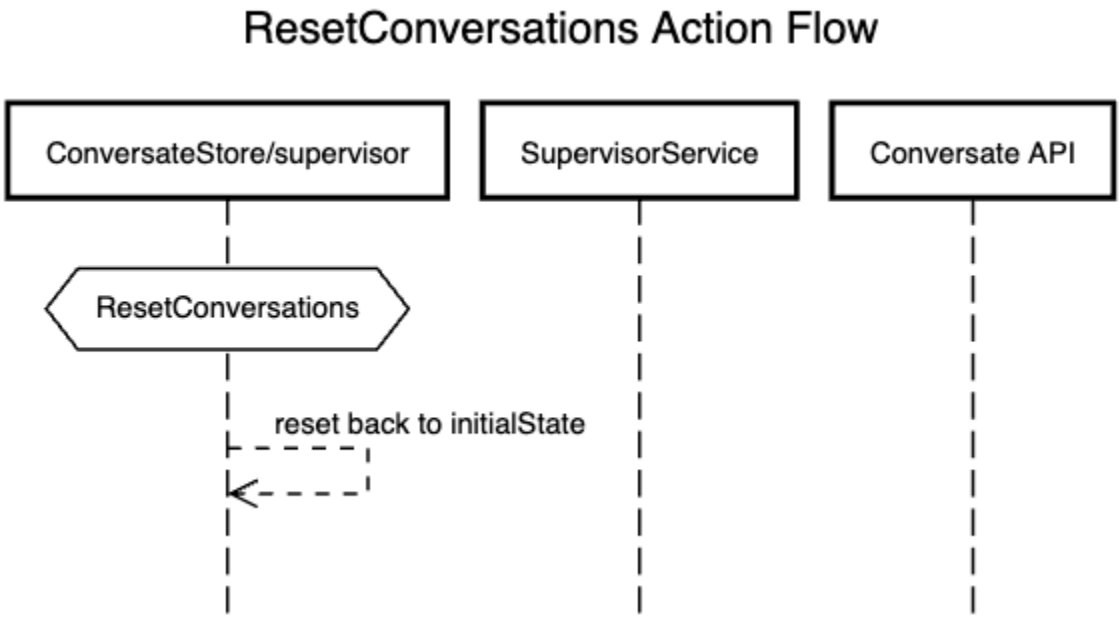
ClearSelectedConversation

clears the active conversation



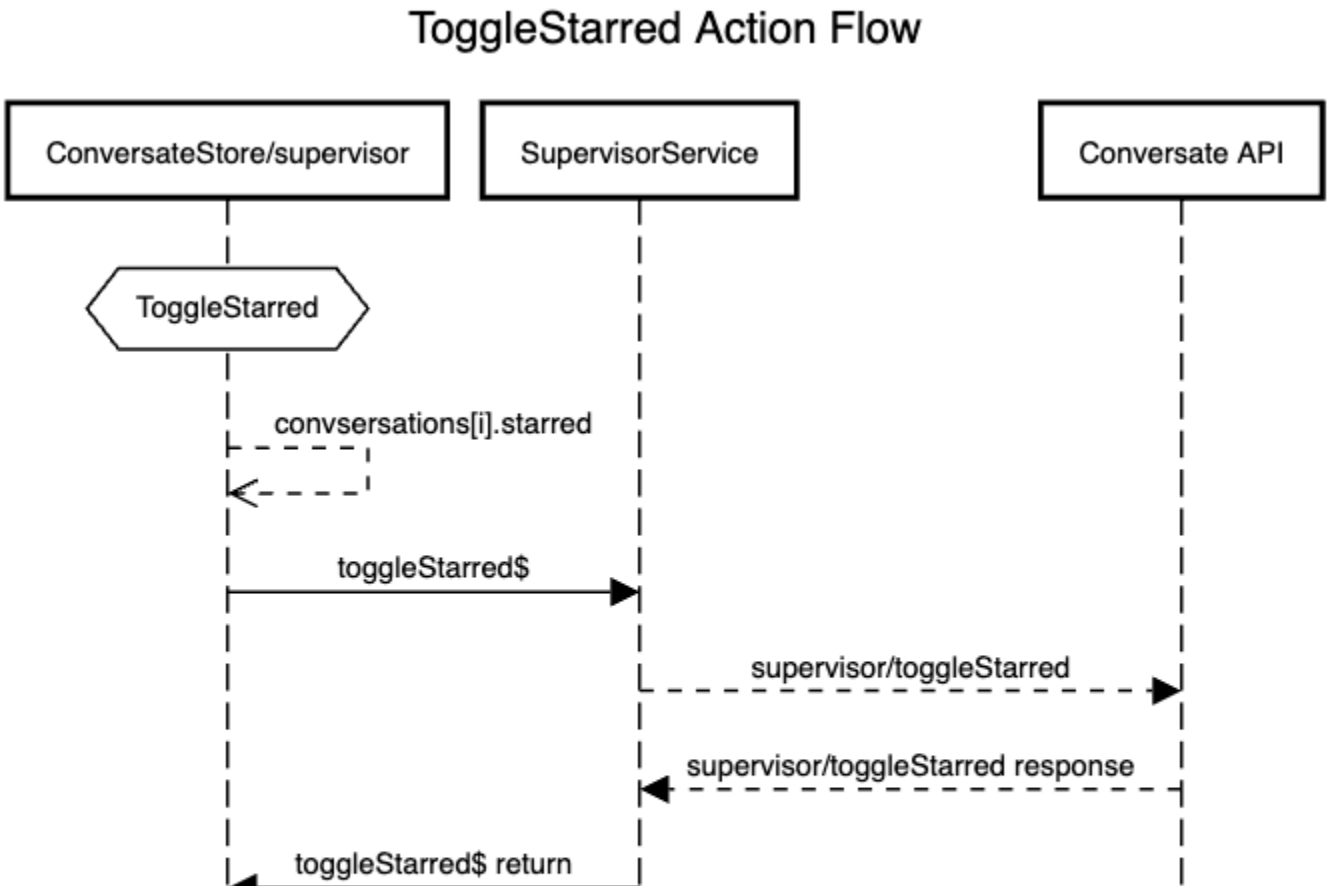
ResetConversations

Set conversations list back to empty. Used when changing active assistant. This action really resets the entire Supervisor state back to it's defined initial state, making it ready to re-hydrate.



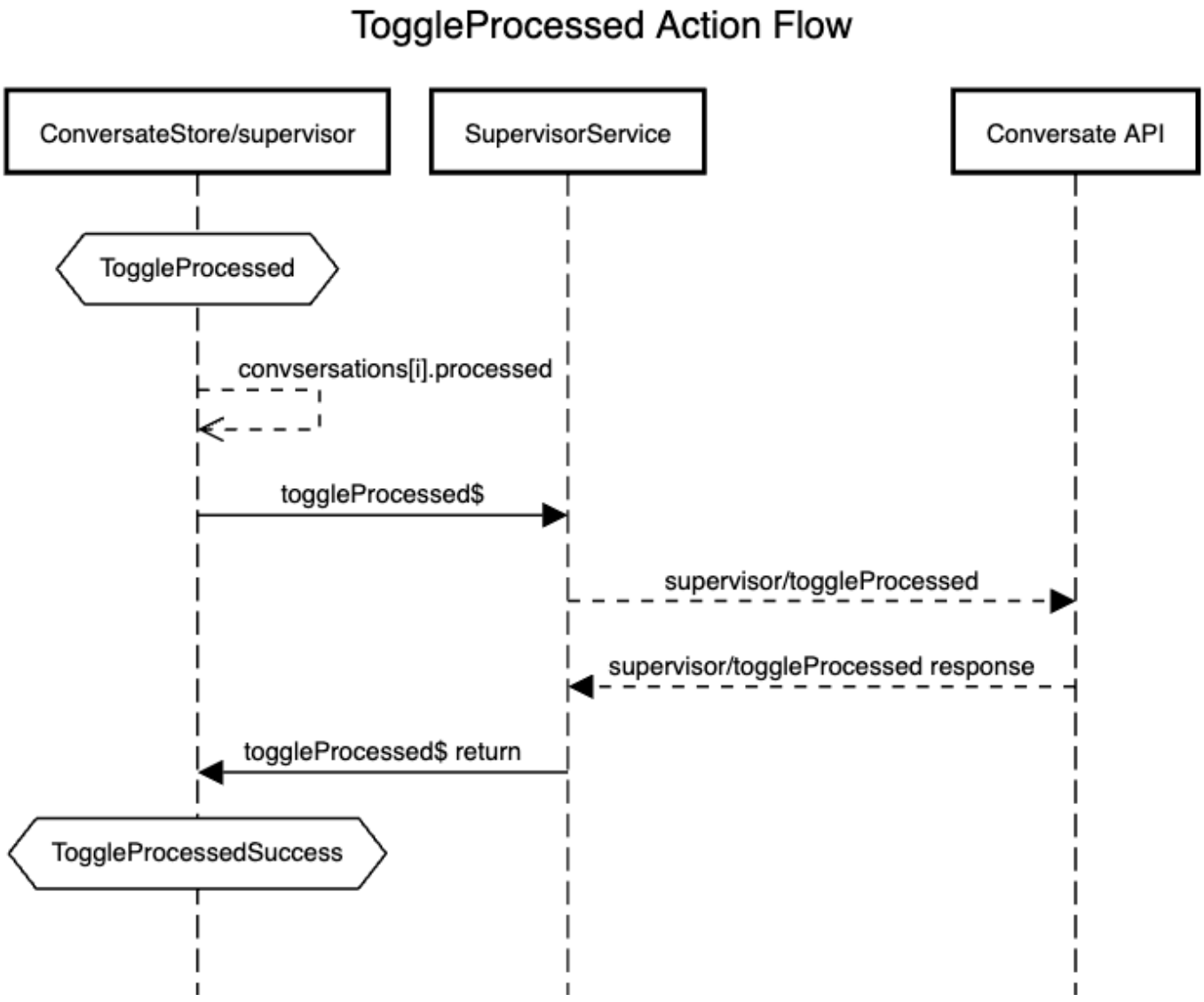
ToggleStarred

Toggles a conversation as starred/un-starred.



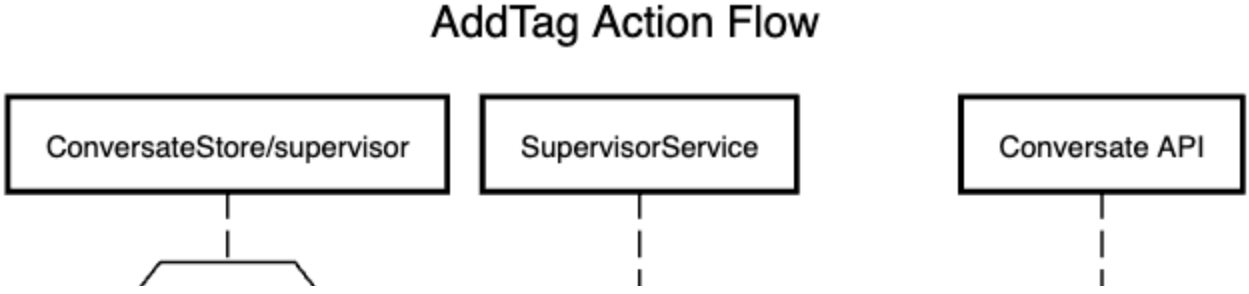
ToggleProcessed

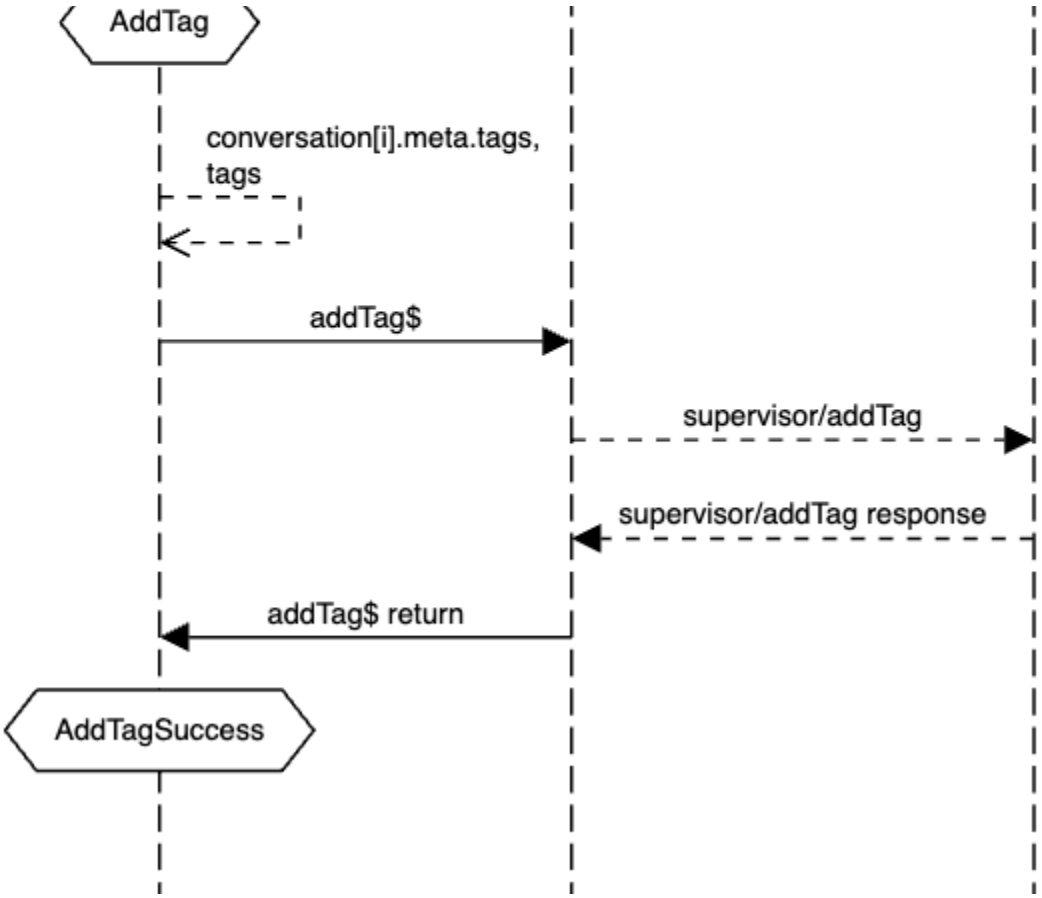
Toggles a conversation as processed/un-processed.



AddTag

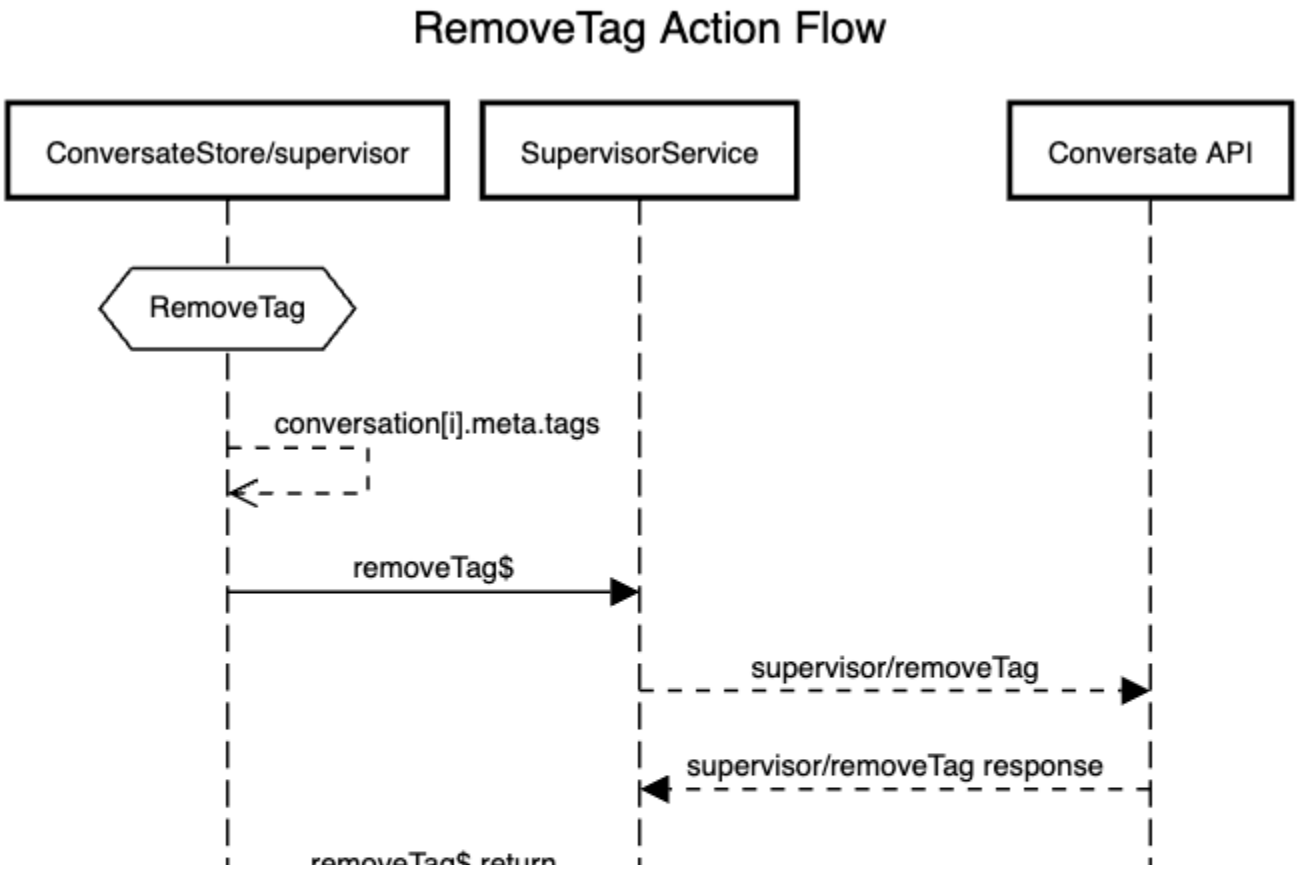
Adds a tag to a conversation

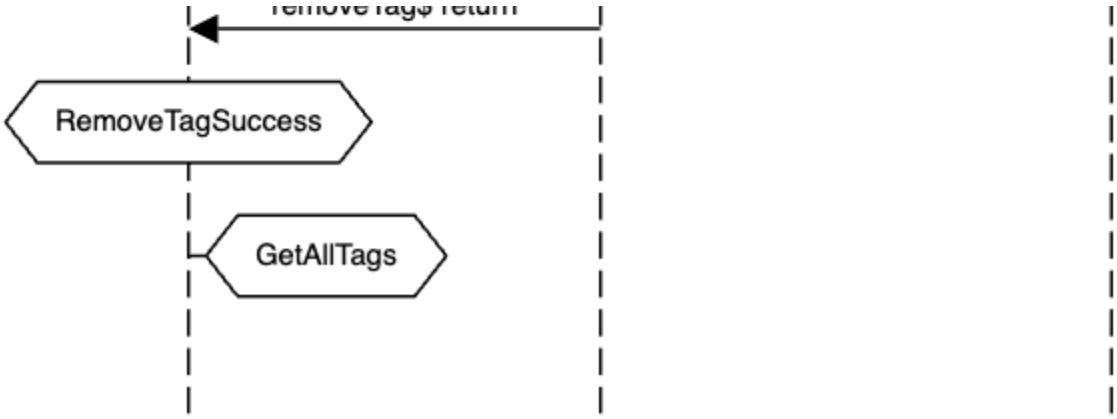




RemoveTag

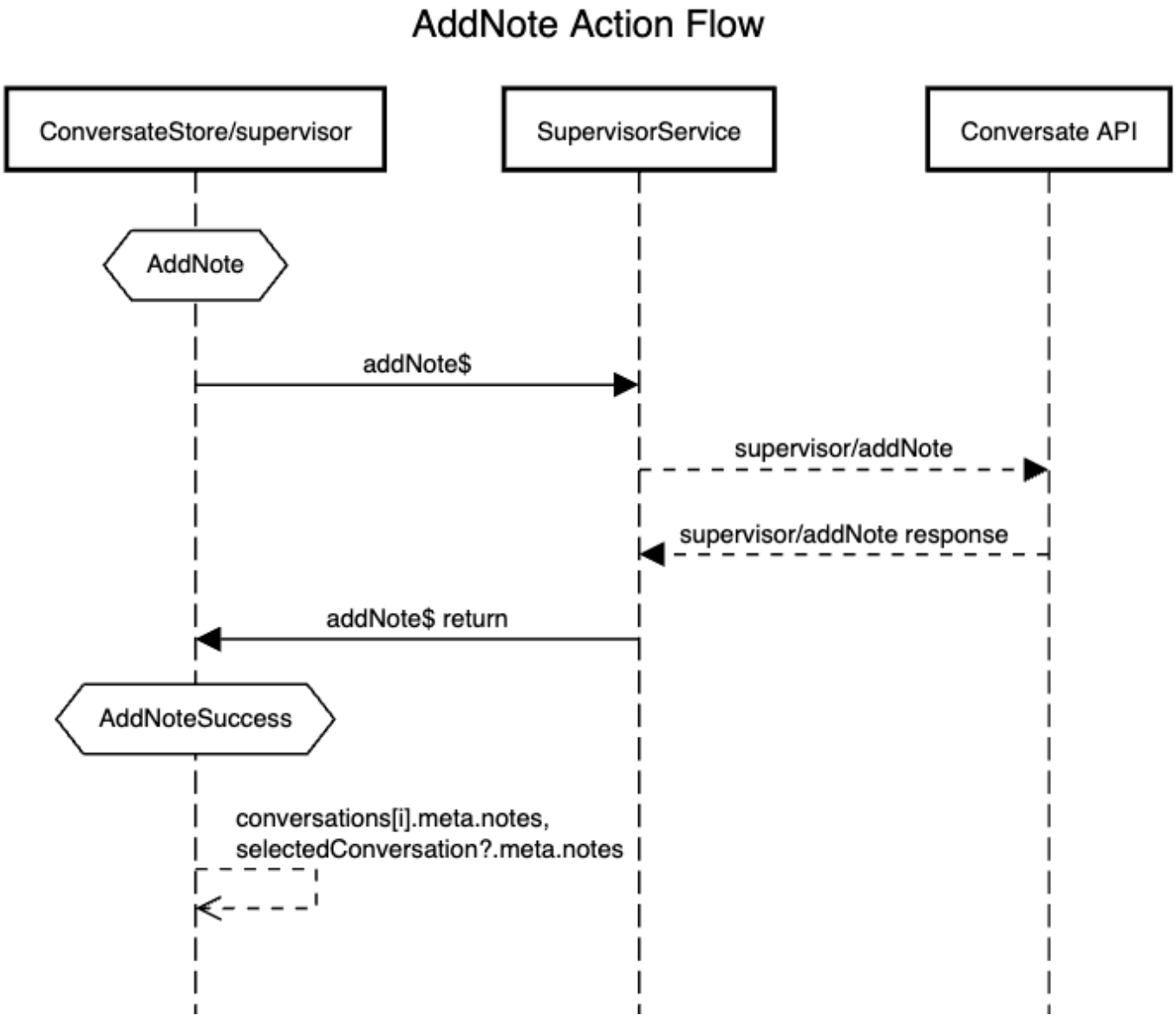
removes an existing tag from a conversation





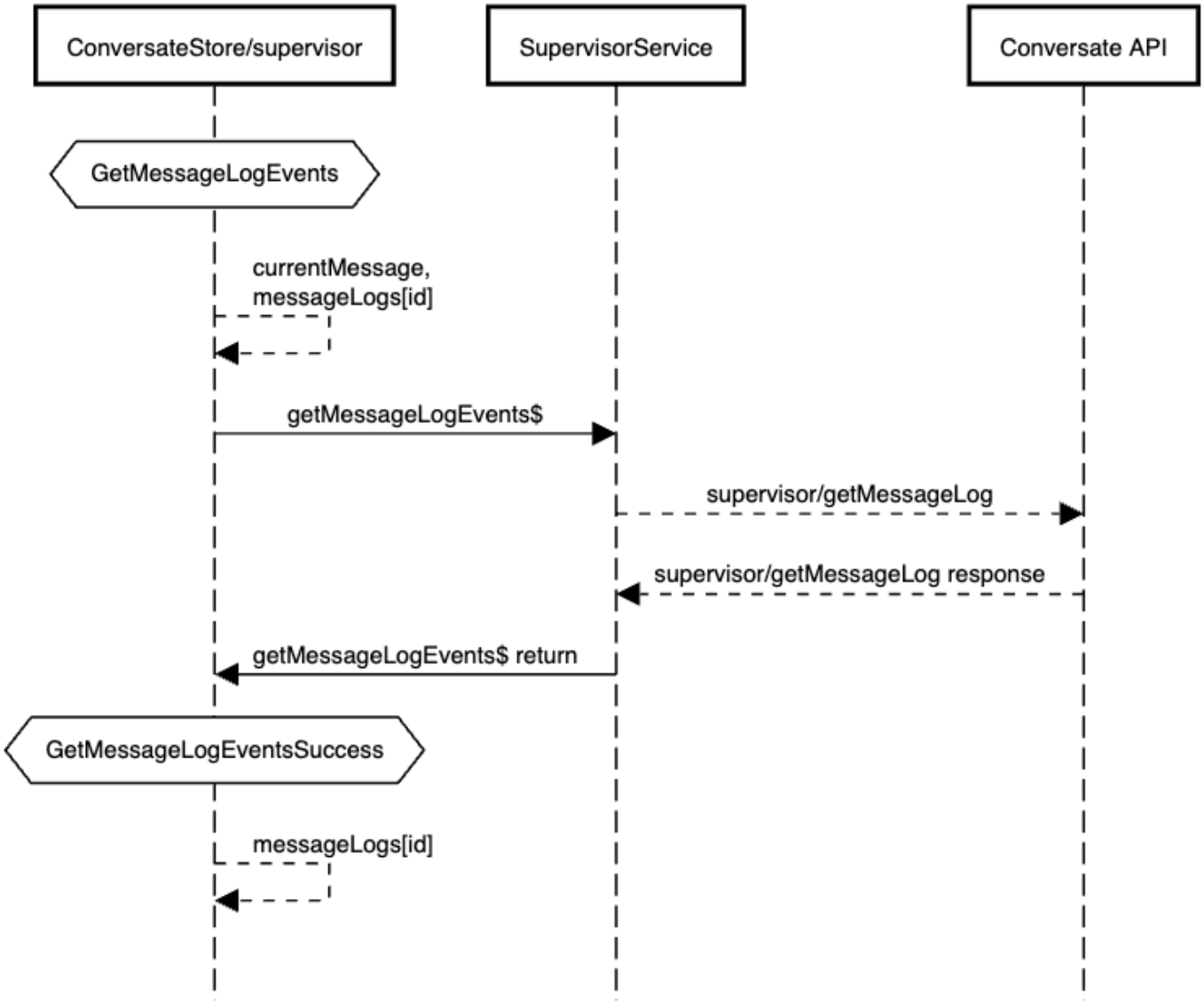
AddNote

Adds a note to the conversation



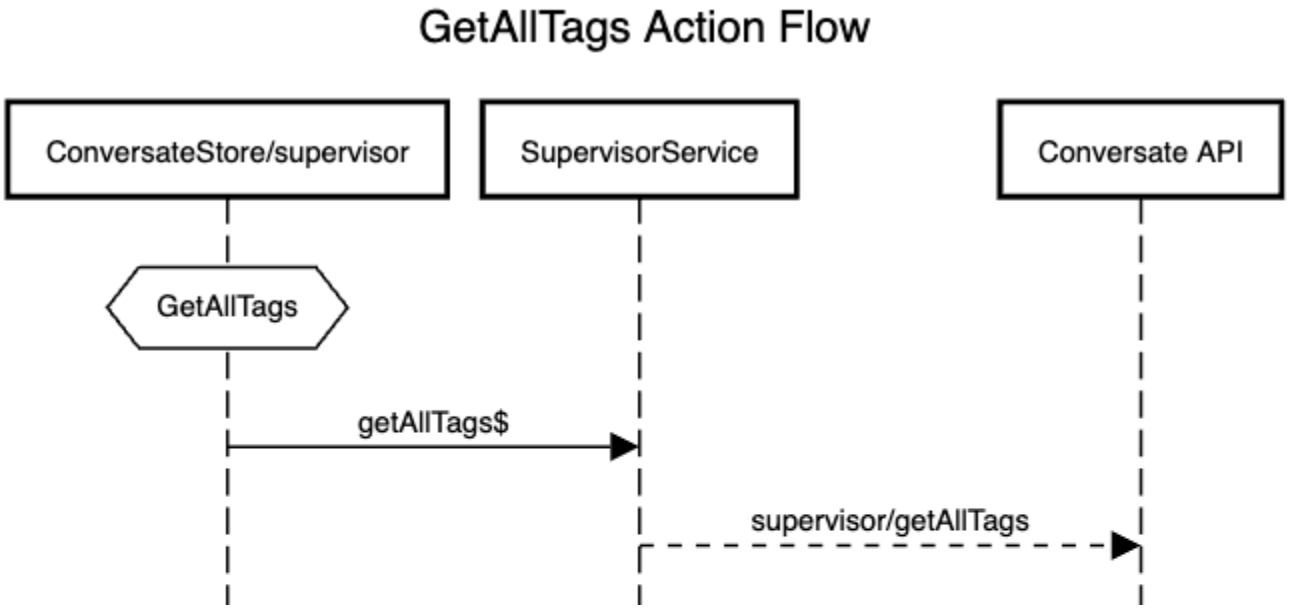
GetMessageLogEvents

triggered when user requests logs for a specific message on a conversation (by correlation ID)



GetAllTags

retrieves all unique tags used across the assistant



Stores

Overview

CUI uses a library called NGRX (version 8) to enable Redux pattern in angular. It is set up to house 3 store: Core, Conversate, and Router. You can read more about each store below.

NGRX?

Stores in NGRX are essentially a collection of data that can be accessed anywhere in the app. They can hold a singular state, or multiple different states. Each state is responsible for its own mutation via a reducer function. Additionally, each state can have its own actions, and effects and selectors.

Terminology

NGRX Action

A special class in NGRX that acts as an event.Once triggered, actions can have mutations to the store, or other side effects, even triggering other actions.

NGRX Effect

a special class in NGRX that allows actions to have additional side effects, whether it be an asynchronous data request, or to trigger additional ngrx actions.

NGRX Reducer

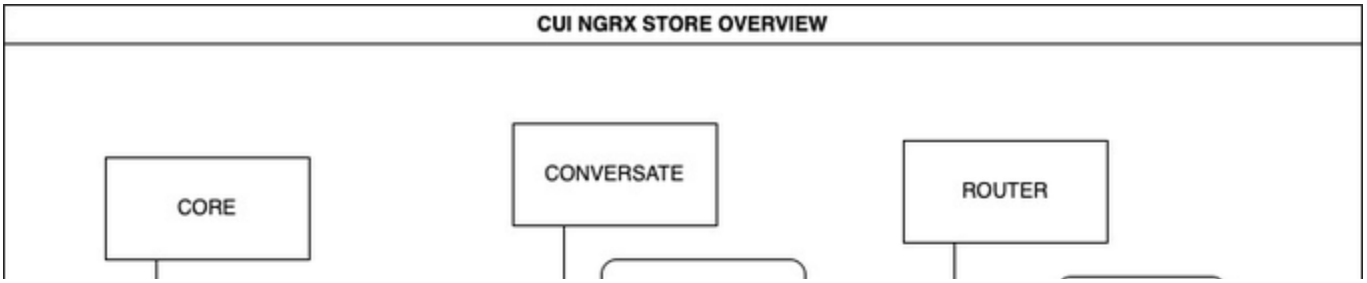
A special class or function that subscribes to the entire NGRX pipeline, and sets up listeners for certain actions on certain states, and each listener is responsible for logic that can generate a new version of the state it is tied to.

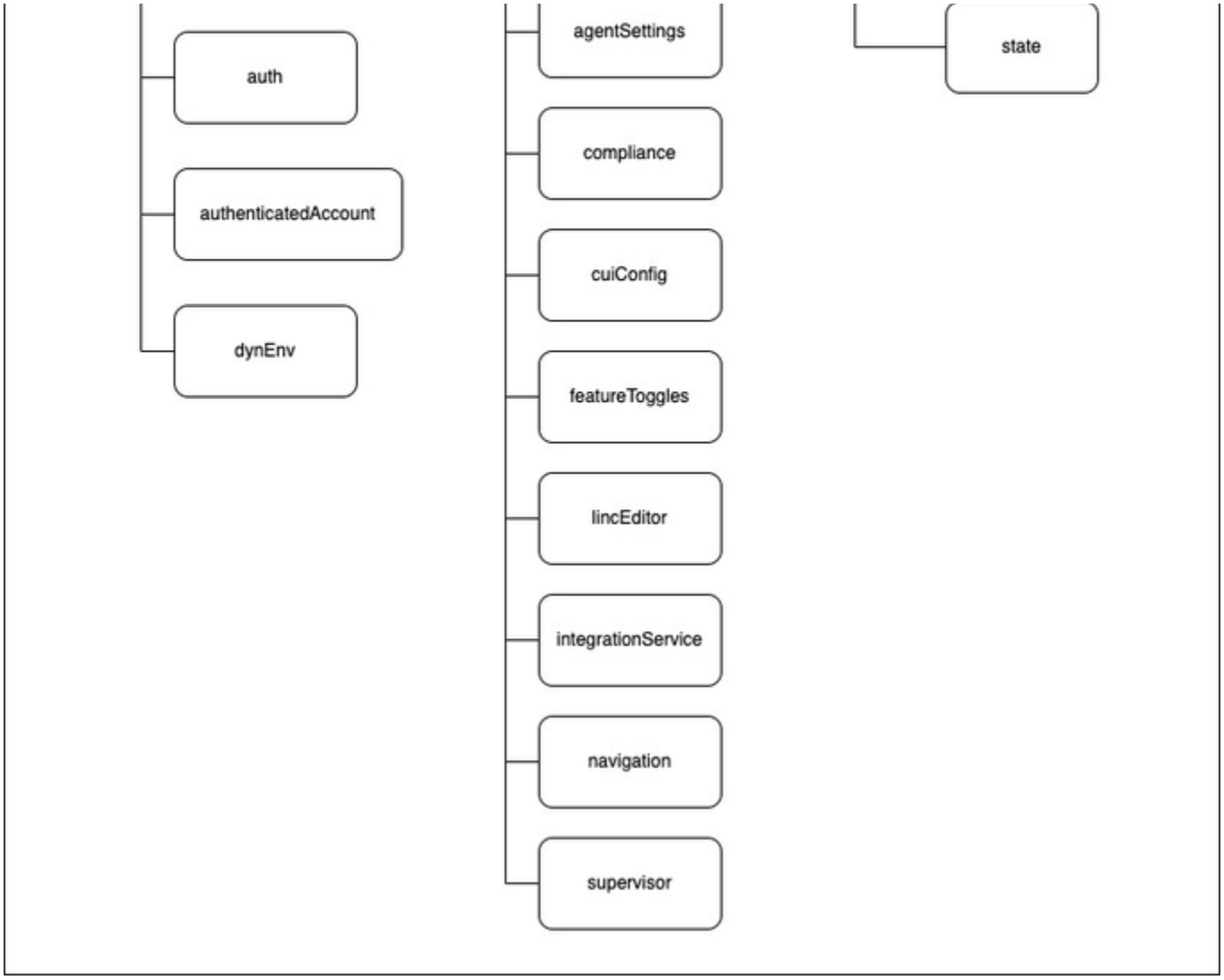
NGRX Selector

A special class that allows components, services and modules, to subscribe to specific data via observables.

You can read more about NGRX from thier documentation here.

Structure





Stores

Core Store

Core store is the main store in CUI, or the “forRoot” store. It is intended for housing information used across multiple apps.

Conversate Store

conversate store is a store that contains data only used in conversate and it’s sub modules. It used to be that all sub modules owned their own state reducers and actions, but now Conversate Module owns that code directly.

Router Store

Tech Stack Details

CUI is built on the Angular framework, using version 10.

NGRX

CUI uses NGRX to utilize Redux pattern, and we have 3 stores set up: Core, and 2 feature stores; router, and Conversate. Learn more about each store in it’s related section.

Socket-IO

CUI communicates with the Conversate API via web-socket, and we use the socket-io library to establish the connection. The web-socket events are used more like REST HTTP requests, rather than typical BE FE broadcast events, there is a clear request response pattern used.

DynEnv

CUI utilizes DynEnv, to load its environment variables, rather than the environment typescript files typically used.

Raven

We utilize Raven.js to connect to Sentry for logging of any console errors.

PUG and SCSS

For rendering our components, we utilize PUG, the HTML pre-processor, and SCSS, the CSS pre-processor. This allows templates, and thier related styling, to be streamlined for devs to write and maintain. PUG needs to be parsed into standard HTML whenever a change is made to a PUG file, and we utilize Gulp to implement that pipeline.

Terminology

Intent

An intent is a package object consisting of a utterance configuration, response configuration, authentication requirements, and integration service method connections (intents that are hooked up to a IS are considered “data-driven intents”

Utterance

An utterance is a single string used as training data for its parent intent. Intents have a one to many relationship with utterances, and they are not shared. They support dynamic variables to be configured within the utterance content.

Response

A response object in CUI is a configuration of how the A.I. should respond when an utterance is matched to an intent. Response’s can have different message content based on the channel used to trigger the intent. They also support dynamic logic within the content via handlebars syntax compatibility. Responses also can take in markup syntax for things like hyperlinks.

NGRX Action

A special class in NGRX that acts as an event.Once triggered, actions can have mutations to the store, or other side effects, even triggering other actions.

NGRX Effect

a special class in NGRX that allows actions to have additional side effects, whether it be an asynchronous data request, or to trigger additional ngrx actions.

NGRX Reducer

A special class or function that subscribes to the entire NGRX pipeline, and sets up listeners for certain actions on certain states, and each listener is responsible for logic that can generate a new version of the state it is tied to.

NGRX Selector

A special class that allows components, services and modules, to subscribe to specific data via observables.

Observables

A special class that can broadcast data to entities that are subscribed to it, and those entites will be kept up to date with any new data passed through.

Embedded Web

Embedded Web is a embeddable chat widget, that provides a connection to Abe A.I.'s paltform, in order to allow end user’s to have conversations with A.I. assistants, as well as human agents.

- Features
- Tech Details
- Terms
  - Utterance
  - Intent
  - Message
- UI/UX

- Widget Icon
- Widget Rollout Text
- Welcome Panel
- Conversation Panel
- User Input
- Quick Replies

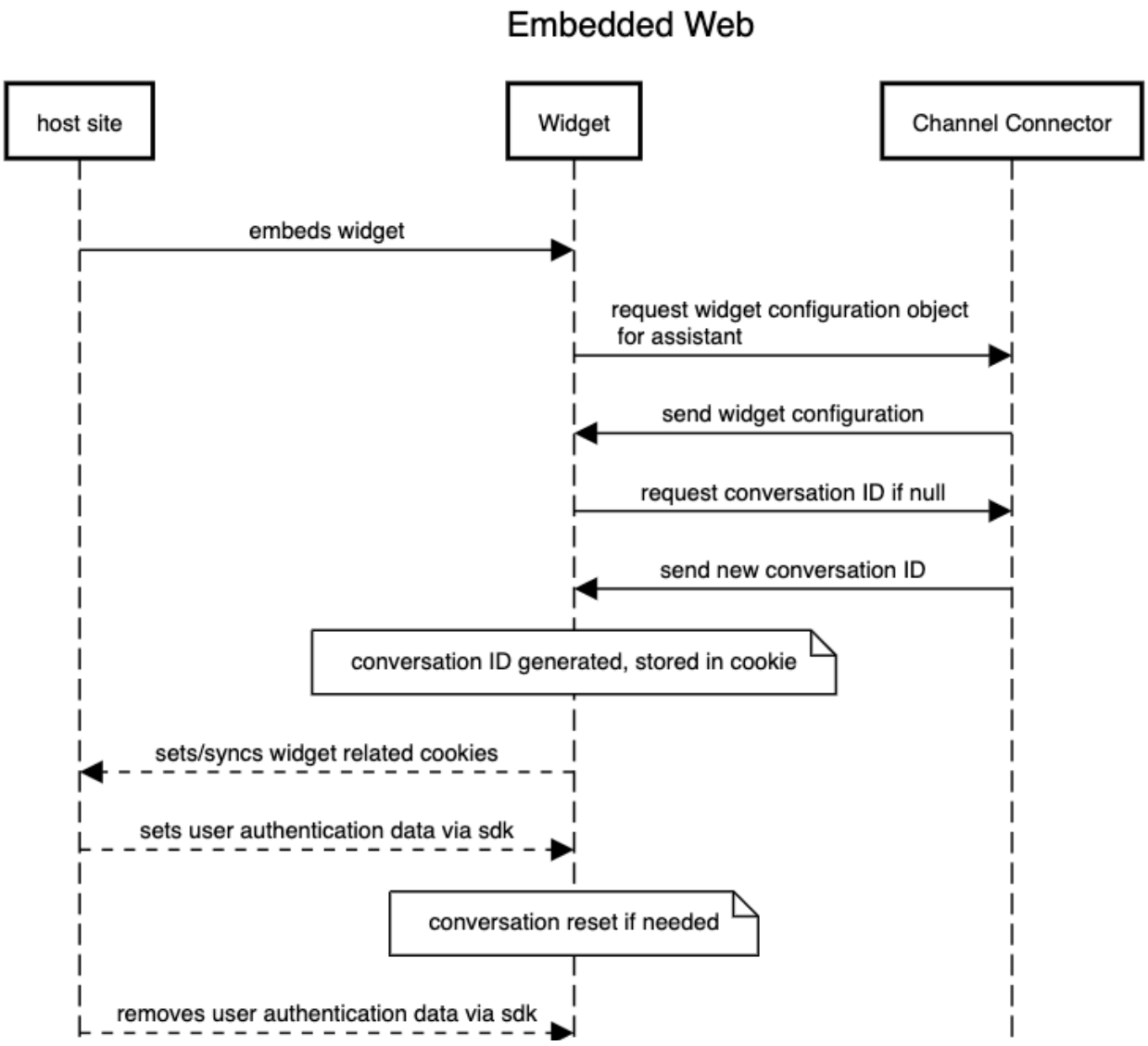
- Human Agent Interaction

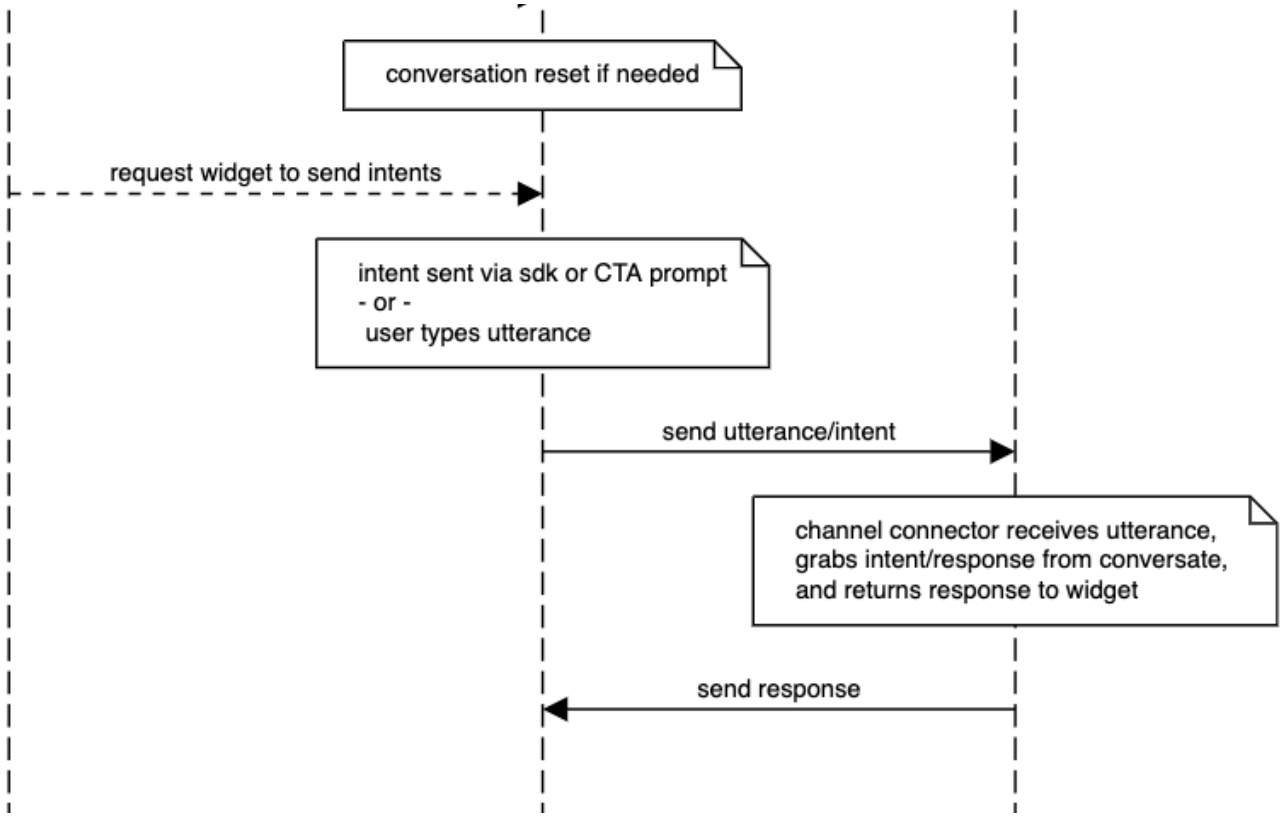
Features

- A.I. Assistant communication
- quick replies
- Preloaded FAQ intents
- Embedded SDK for clients to use (for auth or to trigger custom intents)
- speak with a human agent via Human Handoff
- screen sharing with human agent
- video/audio call with human agent
- Sequences
- Campaigns
- Customization via CUI settings
- Contact Center integration

Tech Details

Embedded Web is built on Vue.js version 2 and is converted to a web component during the build process. The web component is then uploaded to an Amazon AWS CDN where it can be used/imported as a library to embed on client's web pages. It utilizes web-sockets to communicate with the assistant/human via the Channel Connector service.





Terms

Utterance

A utterance is a string, input by an end-user. It is used within conversate/pynlu to determine the proper response the assistant should respond with

Intent

A intent is a string, that is a direct pointer to a mapped intent object on the assistant. It is used as a key in conversate's brain to return a response.

Message

A message is a response from channel connector. It gets mapped to individual response message bubbles.

UI/UX

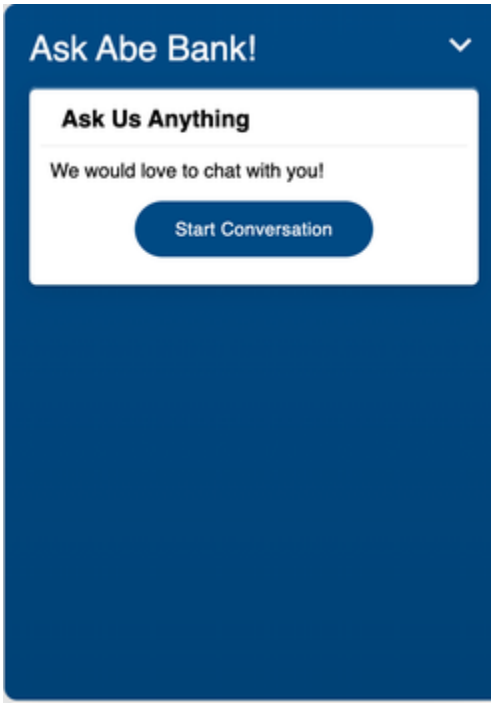
Widget Icon

- created in widget component
- pulls image and name from the tenants config
- on click invokes CTA to toggle the dialogue open

Widget Rollout Text

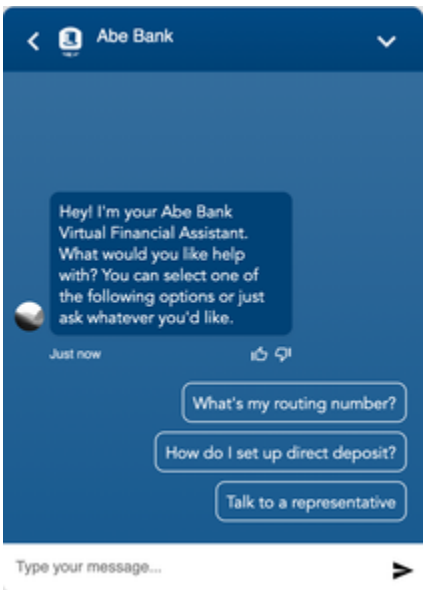
- set in widget component
- rolls out when hovering over the widget icon
- can display a default rollout text using the tenants assistant name or override text that comes from app state

Welcome Panel



- created in start-conversation component
- pops up when the widget icon has been clicked for the first time
- gets the title and content from the tenants chat panel config
- can hold FAQ's that will trigger quick replies when clicked

Conversation Panel



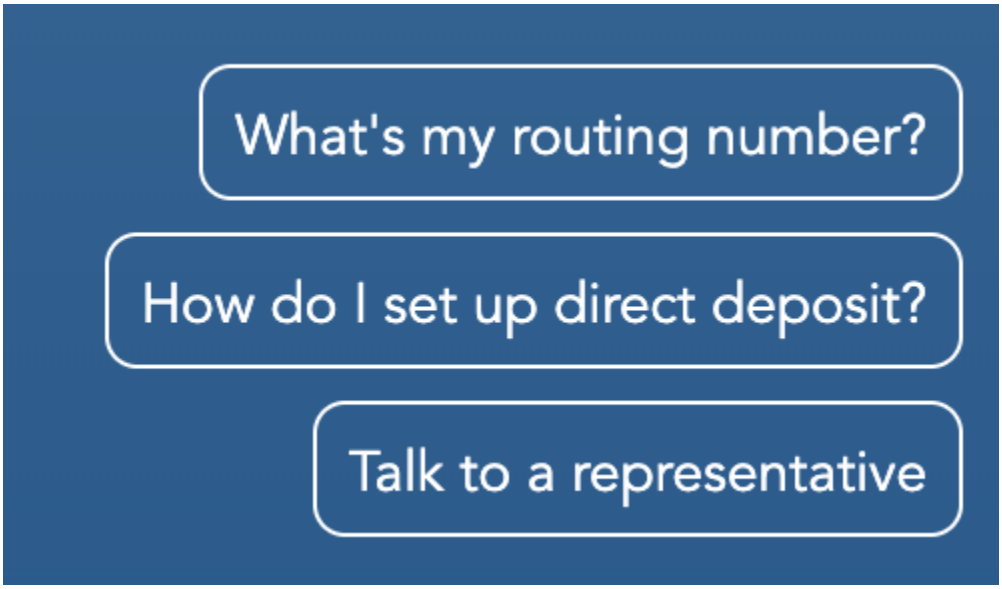
- created in conversation component

- gets assistant name and image from tenants config
- allows the user to interact with the AI and human agent
- allows the user to give feedback on responses
- can hold quick replies

User Input

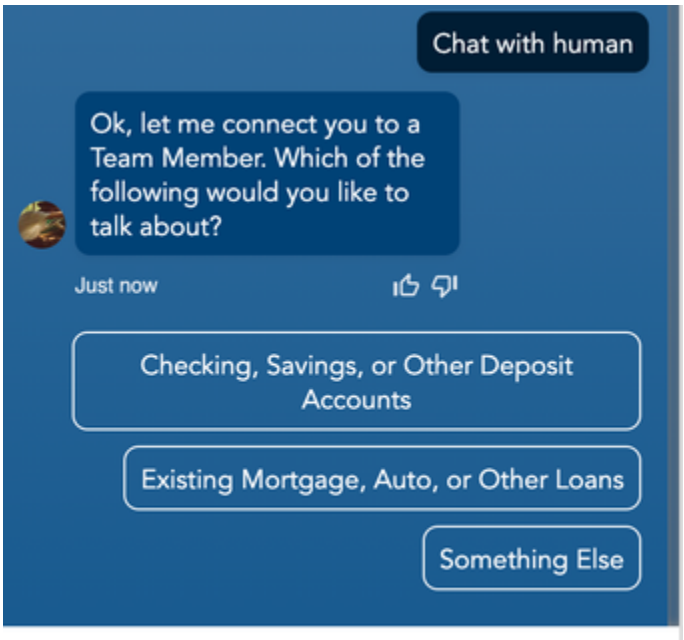
- created in message-input component
- keeps track if the user is currently typing
- sending a message triggers the SEND_MESSAGE action

Quick Replies

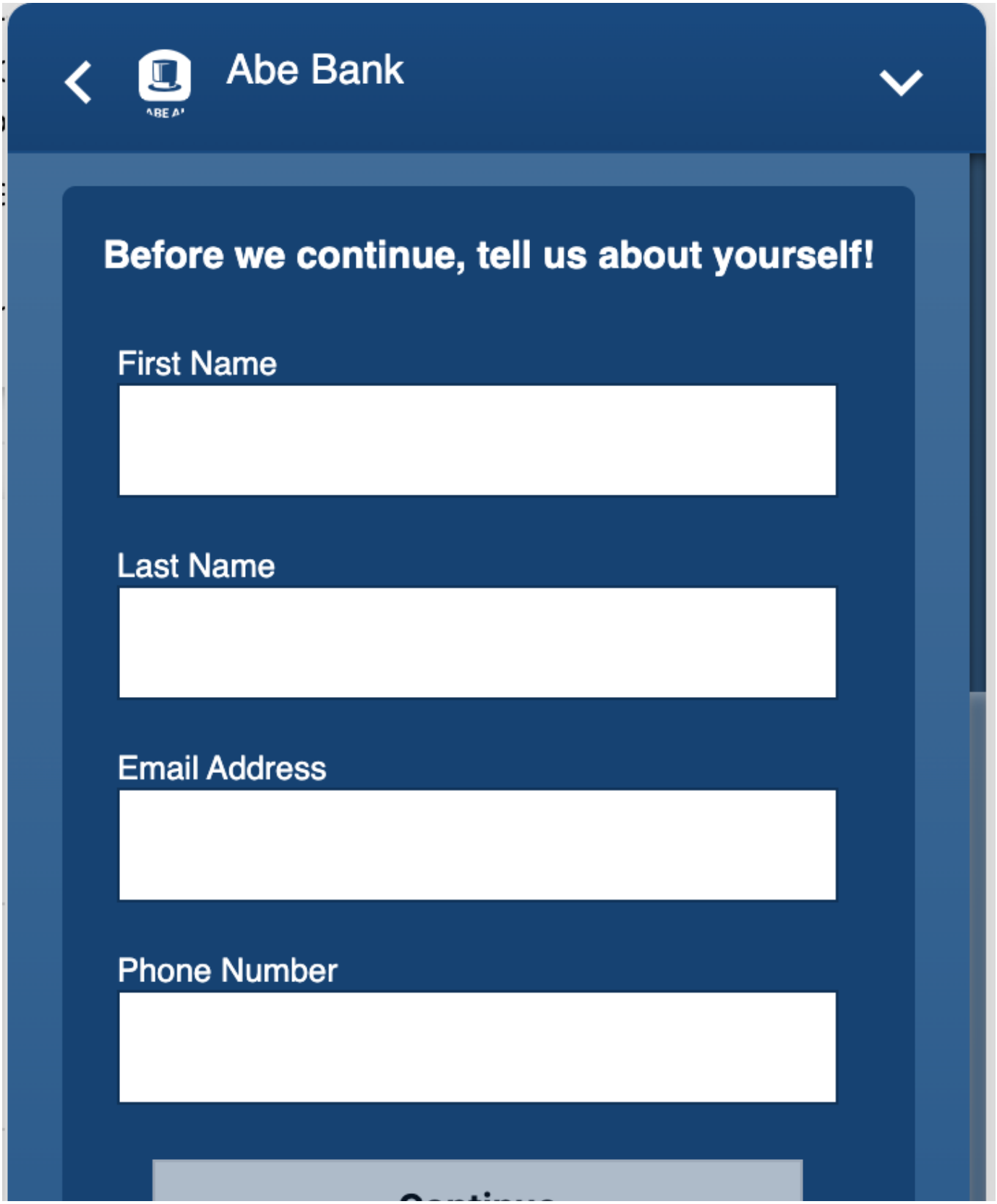


- created in quick-replies component
- gets the quick replies from the message data that is passed from conversion component
- clicking a quick reply will trigger the connecting intent

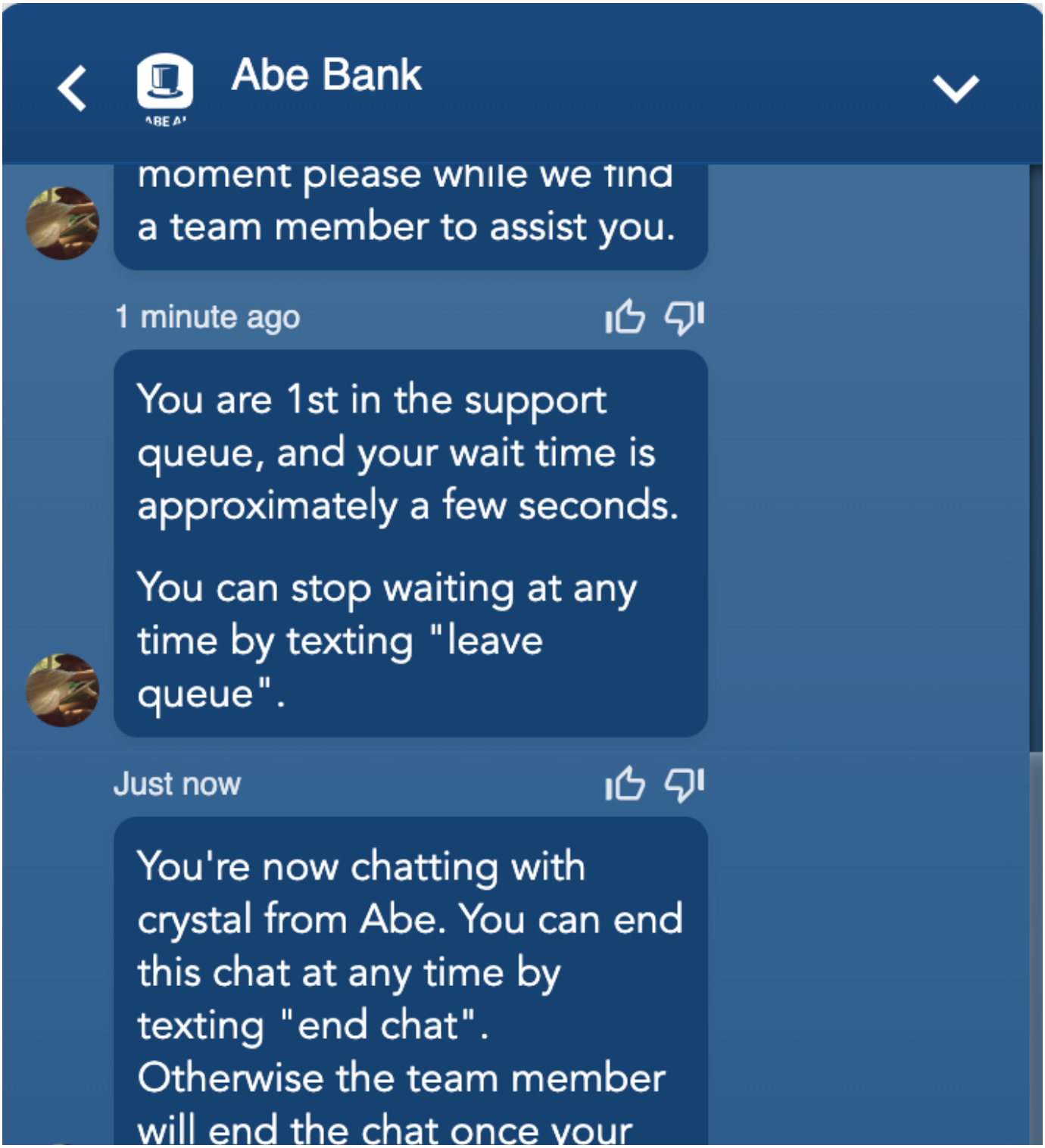
Human Agent Interaction

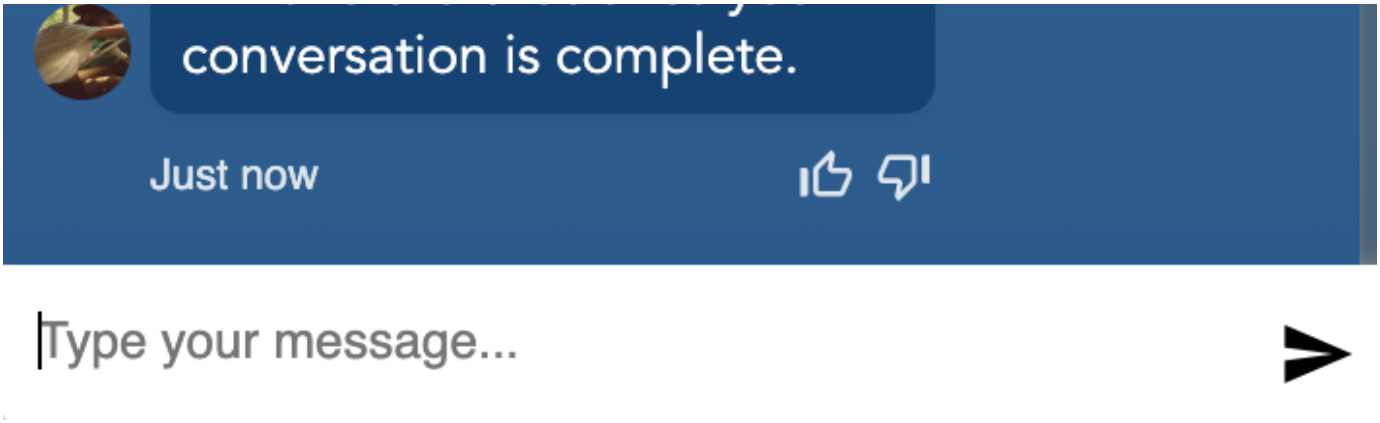


- The user can request to speak to a human agent anytime durning a conversation with the AI.
- After requesting a human agent, the AI can ask what topic the user will like to talk about so they can be redirected to the connecting team in Contact Center.

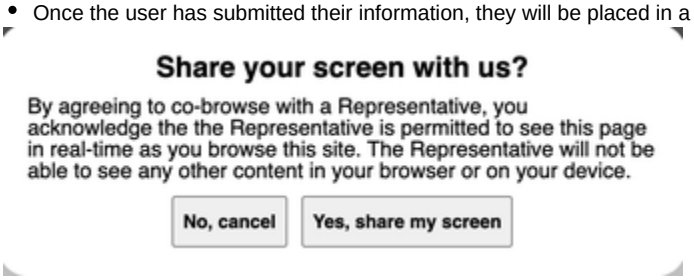


- After choosing a topic, the user will be asked to fill out information.
- Each field can be set to be required or not



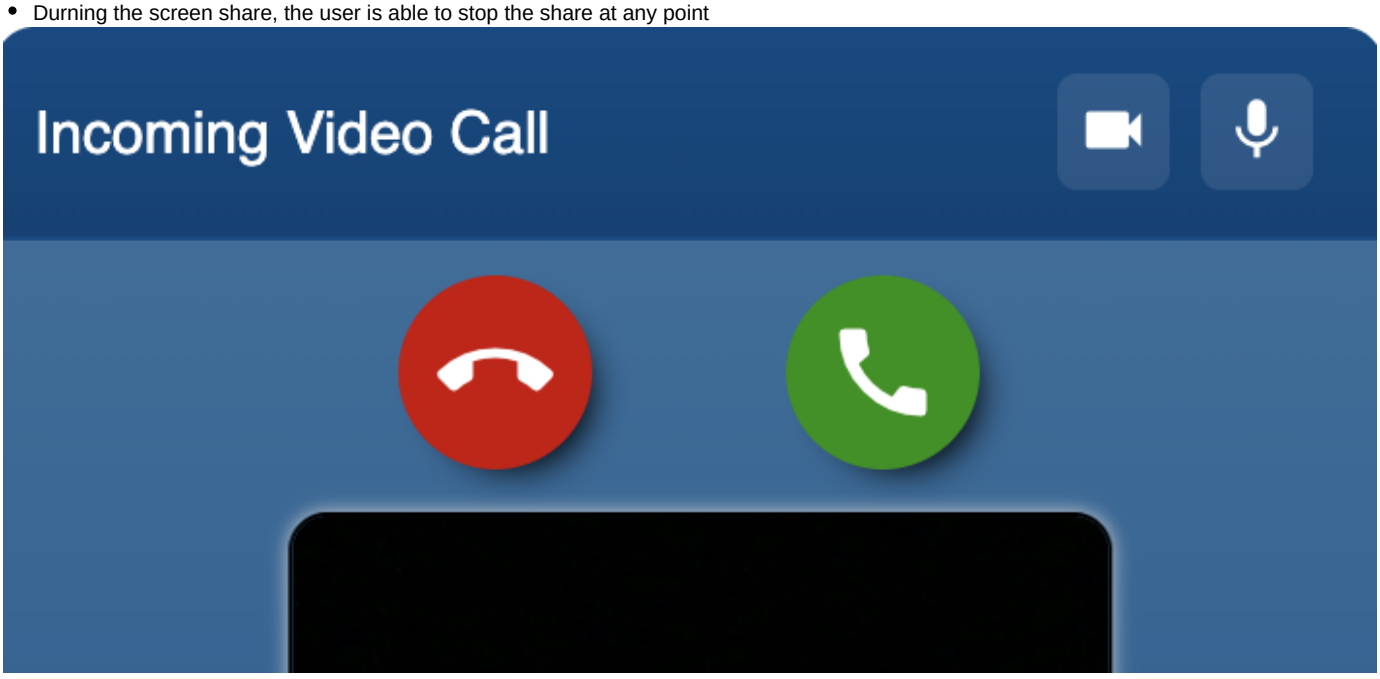


- Once the user has submitted their information, they will be placed in a queue until they are connected with a agent from Contact Center

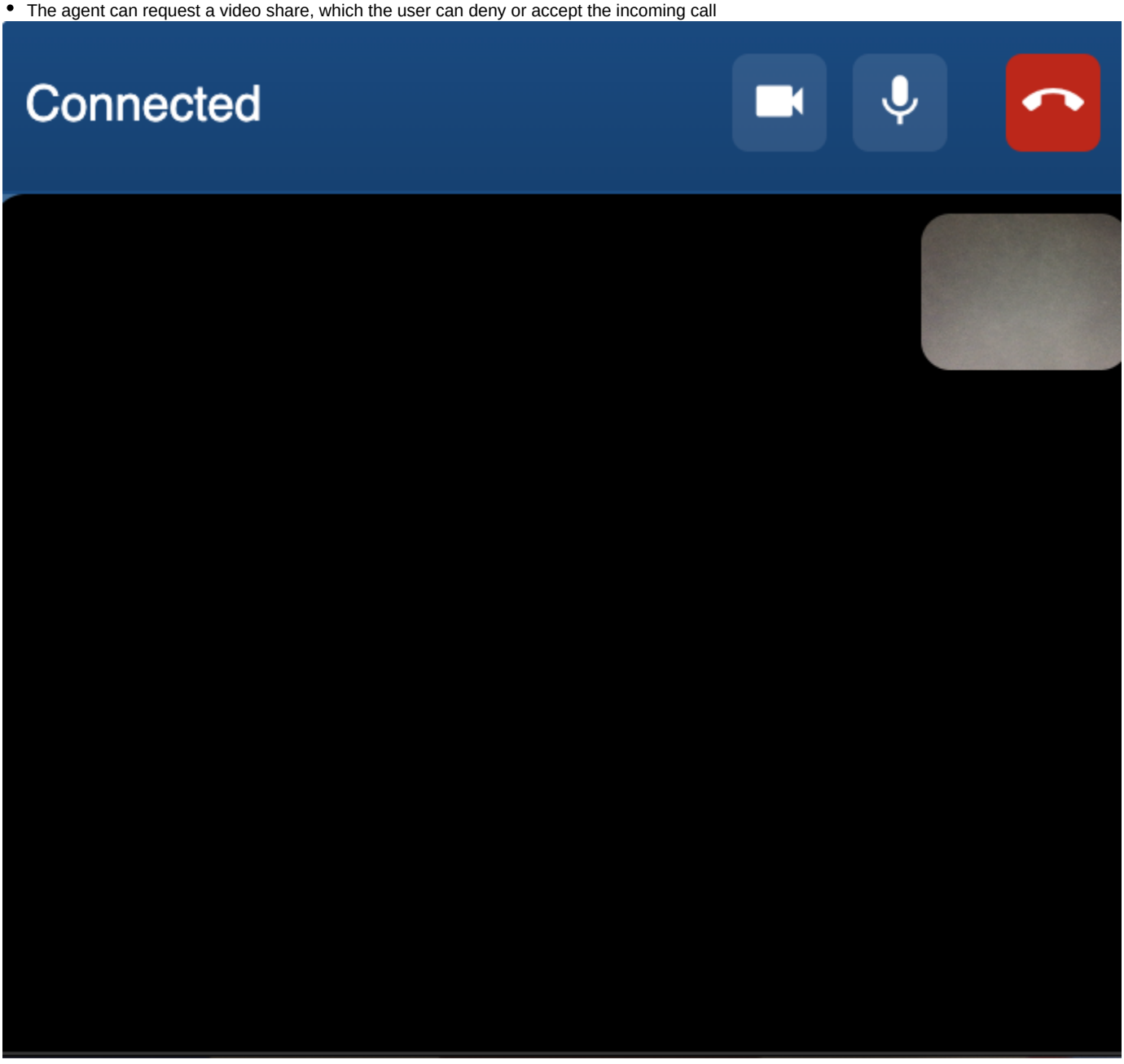


- The agent is able to request a screen share, which will prompt the user if they approve the co-browse

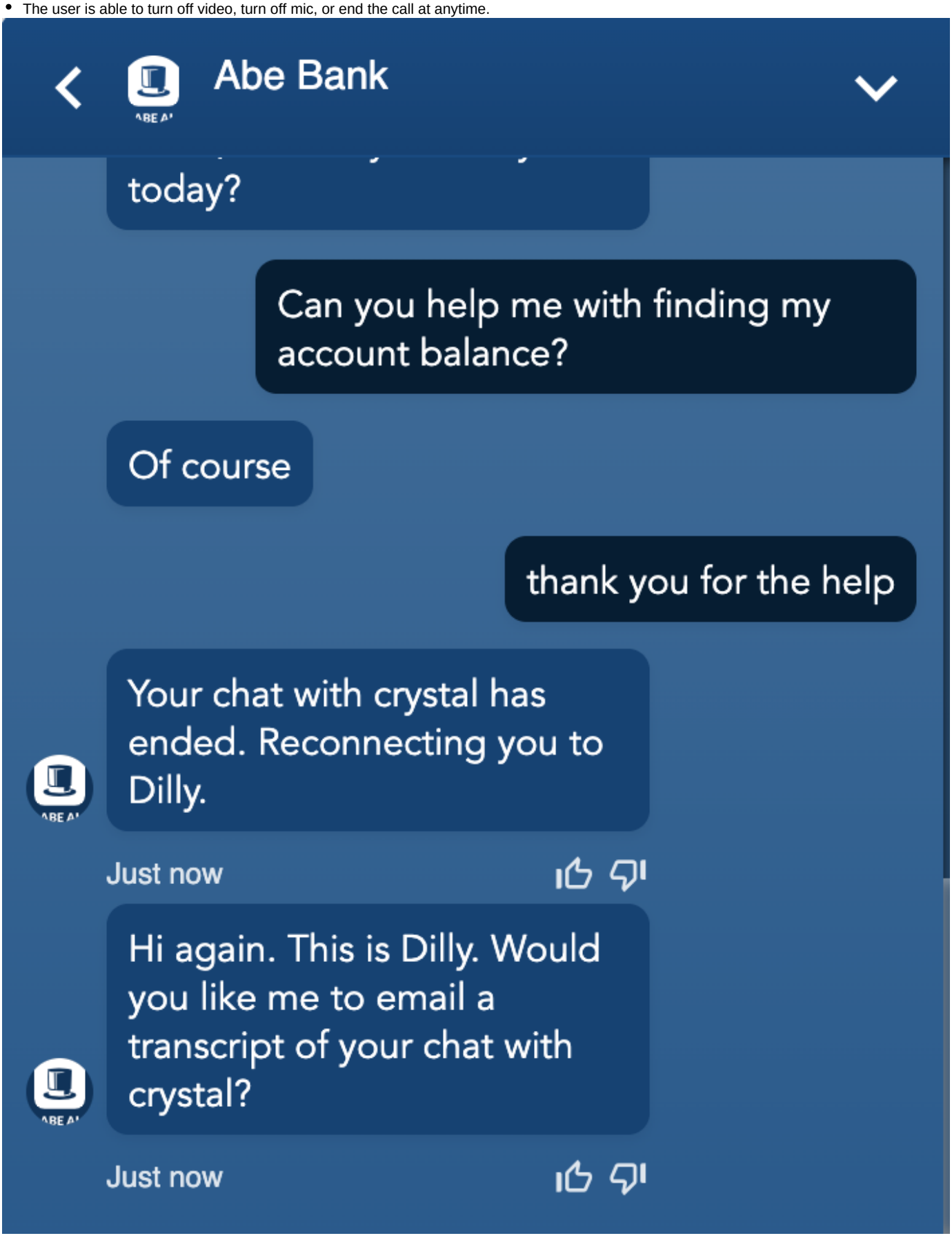
- Durning the screen share, the user is able to stop the share at any point



- The agent can request a video share, which the user can deny or accept the incoming call



- Durning the video call, the agent video will take up the whole black screen while the user video will be in the top right corner
- The user is able to turn off video, turn off mic, or end the call at anytime.



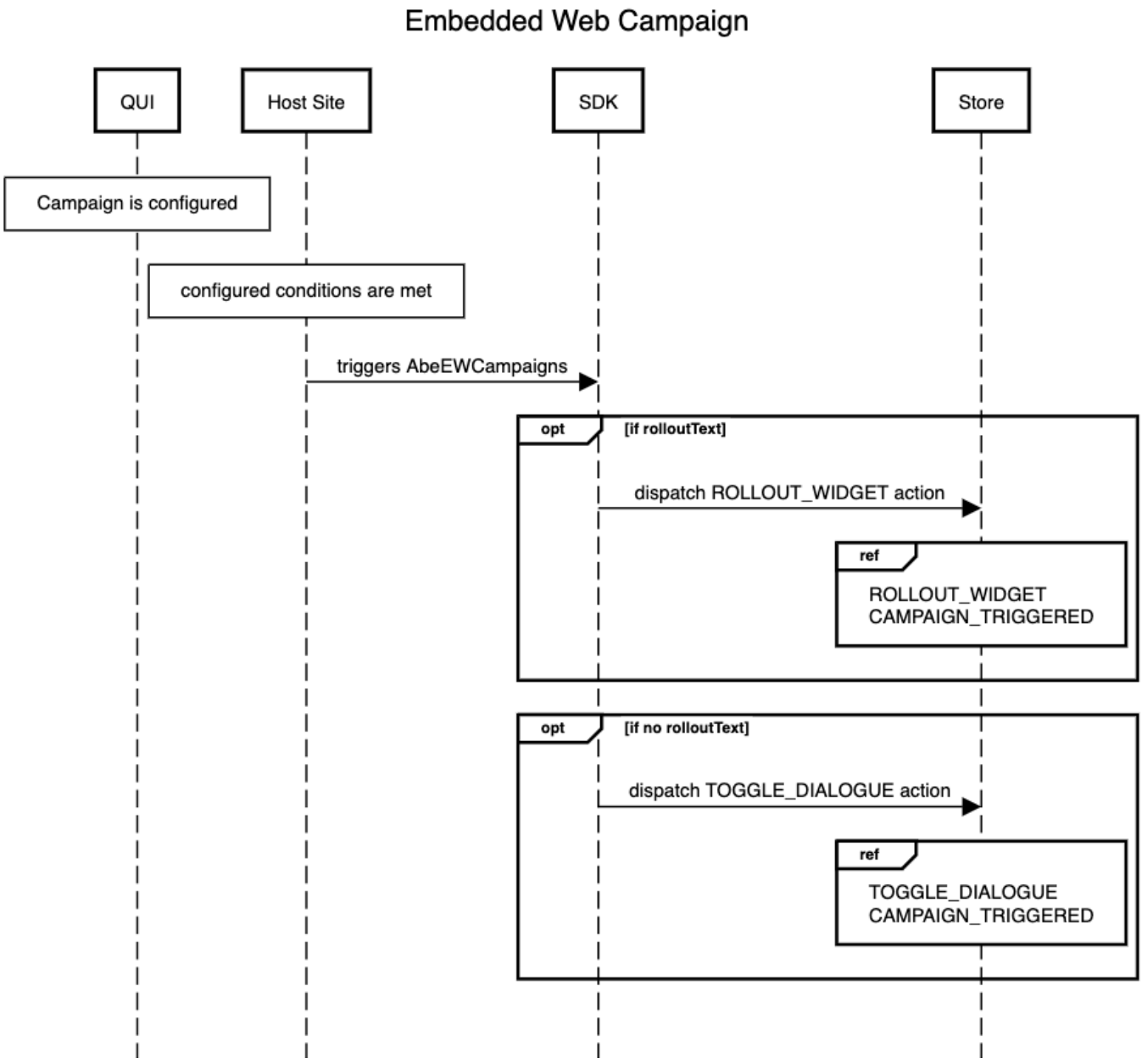
- Once a conversation has been ended by the human agent, the user will be prompted if they would like a email of the chat

Embedded Web Campaigns

Overview

Embedded Web Campaigns are similar to CTA's and are configured in QUI by anyone on the business side of the product. Unlike CTA, Campaigns are given a start and end date for how long it will be active. When all conditions are valid it will trigger either a partial or full CTA that can contain custom text, hyperlinks, or an intent key matching the content. The FI is responsible for configuring the triggers, which can include scrolling past a certain point on a webpage, clicking a link, or visiting a specific page.

Campaigns Flow



Campaigns Flow Diagram

Embedded Web CTA

Overview

Embedded Web custom Call to Action (CTA) is triggered by an intent or custom text that is configured by the FI and called from the SDK. There are two types of CTA, a full and a partial. Full CTA appears as a message in the chat box that can contain custom text, hyperlinks, or an intent key matching the content. Partial CTA appears as a prompt in the the button that opens the chat box.

CTA Characteristics

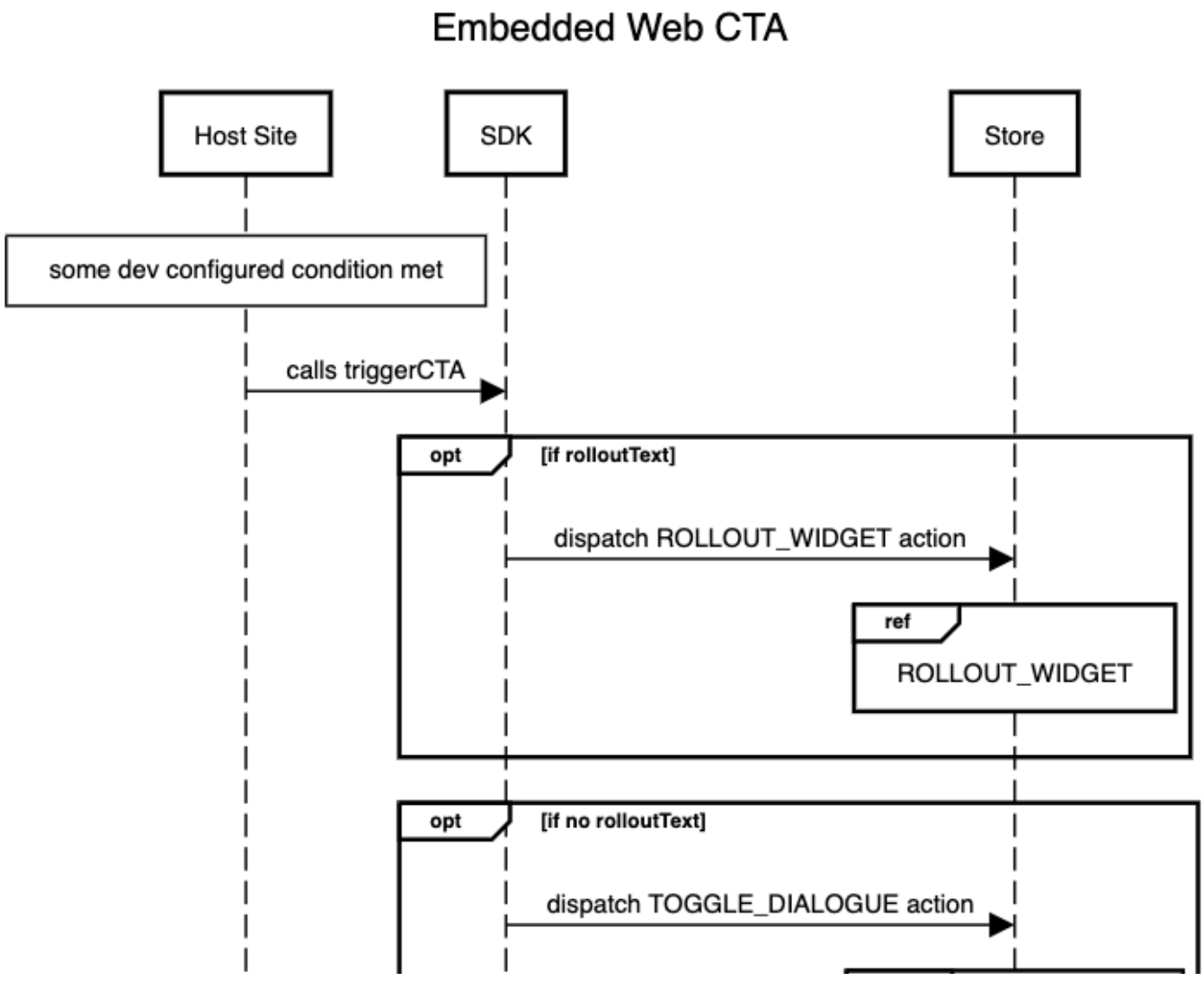
- Full CTA
  - will not trigger if the chat box open
  - can contain custom text, hyperlinks, or intent key matching the content
- Partial CTA
  - will not trigger if the chat box is open
  - will only appear for 5 seconds
  - disappears when clicked
  - has no maximum characters
  - can contain custom text, hyperlinks, or intent key matching the content

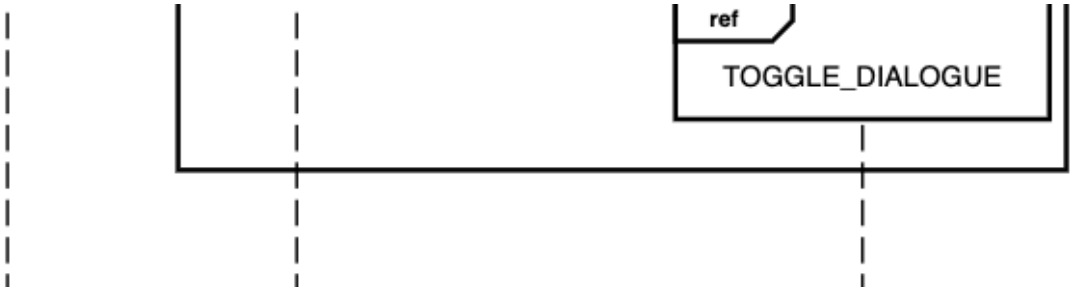
CTA Triggers

The FI is responsible for configuring and calling the triggerCTA event through the SDK, some trigger examples are:

- user visits a specific page
- user clicks on a link
- user scrolls past a certain point on the webpage

CTA Flow





Embedded Web CTA Diagram

View referenced actions flow in more detail in App State for ROLLOUT_WIDGET and TOGGLE_DIALOGUE

Embedded Web Sequence

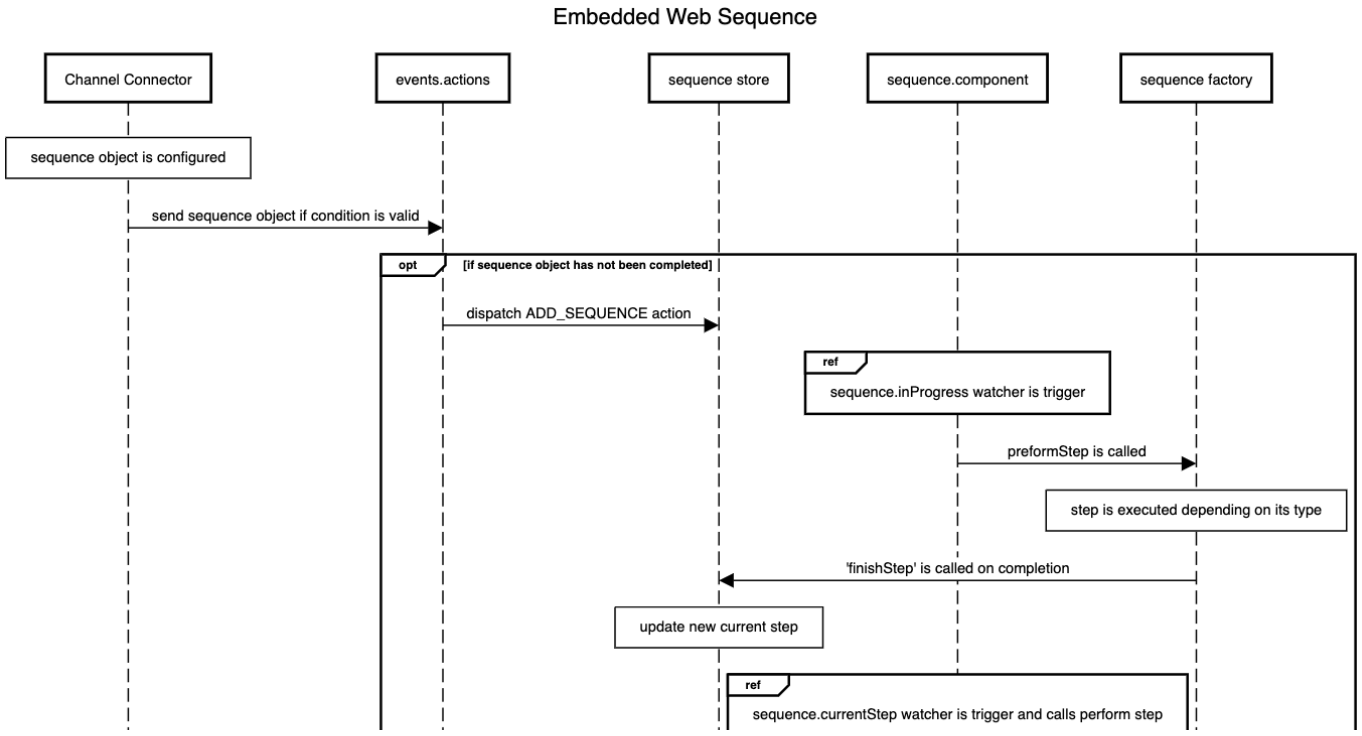
Overview

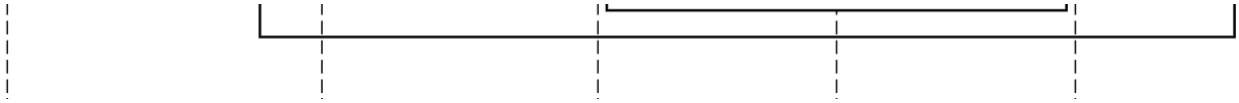
Embedded Web Sequence allows anyone on the business side of the product to create a Sequence Object containing a list of steps that will be executed when the event is triggered from Channel Connector. These steps can help guide the user throughout the website to assist with finding out more information on a selected topic.

Sequence Steps

- Click
  - performs a click event on a selected element
- Highlight Text
  - highlights a selected line or multiple lines of text
- Highlight Element
  - highlights a selected element
- Move Cursor
  - moves the mock cursor to a selected element
- Toggle Dialogue
  - toggles the widget dialogue
- Navigate
  - navigates the website to a selected URL
- Pause
  - waits for the given duration time before moving to another step

Sequence Flow





Embedded Web Sequence Flow

SDK

Overview

Embedded Web installs a SDK on the window for clients to use, with some standard methods exposed

Methods

authenticateUser

Calling this method will allow the widget to pass authentication data with utterances/intents. Clients need to invoke this method when a user authenticates via their web app. Once this is set, intents that rely on authentication will return proper responses, instead of an unauthorized response.

unauthenticateUser

Calling this method will remove the authentication data from the widget's store, and subsequent utterances/intents. It will also clear out the current conversation, and start a new conversation if any conversation has been started.

triggerCTA

Calling this method will allow client's webpages that are embedding the widget the ability to trigger the widget to send an intent.

Factories

Overview

In Embedded Web we utilize a few base functions in a common directory we call `Factories`. These functions do not necessarily follow the "factory pattern" in traditional programming, as some operate more like a utility class.

Factories

API

The API factory is a utility class, and the file exports a singleton instance of the API class, providing the ability to connect to Channel Connector via HTTP requests. It has methods for `POST`, `GET`, and `PUT` , and also handles adding our standard `ABE-AI-EW-VERSION` header to calls.

AutoResize

Resizes a custom target depending on its scroll height and padding.

Cobrowsing

Evaluates multiple cases that comes from the live web-socket message to keep track if it is ready, started, reconnected, ended, agent drawing, redrawing, or clear agent's drawing.

Color

Exports methods that can darken or lighten a given color by a custom percentage or convert an RGB to HEX or HEX to RGB.

DotNotation

The DotNotation file exports 2 functions, `assign` and `get`. These functions are able to set/return a value to a property nested somewhere in a root object's tree. This is similar to the get/set functions found in lodash. These functions are mostly used within state mutations, most notably as part of the `oneToOneMutations` that can be found in a few states. These one to one mutations are just simple mappings with no mutation logic.

GenerateSignature

Generates a new signature key using hash-based message authentication codes (HMAC) from create-hmac. This is used to when calling `POST` on `API`, to create a signature key we store in the `X-Hub-Signature` header.

Links

Exports methods that will change the current window location to the given link or create a new tab from a given external link.

Live

The Live factory is tied in with the cobrowse and video factory files. It is responsible for establishing a websocket connection to the `cobrowseWebsocketUrl`, sending events over the socket, as well as handling incoming events that are sent over the socket.

MessageTextToHtml

Uses Remarkable 'linkify' to convert the given text to HTML. We mostly use this to enable markdown style links in our templates to be converted to html anchor tags.

Sequence

Holds the logic to perform each unique step from a given sequence object.

SharedStorage

Exports methods that can get or set data from the document cookies that uses one of the shared custom storage keys.

SoundData

Exports the sound data URI for when a message is received in the dialogue.

Utils

Holds the logic that gets and sets the configuration channel id and its storage data.

Video

Holds the logic that gets permissions for a live video chat as well as handle what happens when a call is ringing, started, or ended.

Websocket

the Websocket factory file exports a class `WebSocketInstance`. This class can be instantiated with a endpoint as it's constuctor argument, and will initialize a connection over websocket protocol to the endpoint. This file also has a dictionary object of open websockets.

Store

Overview

Embedded Web uses a redux pattern implementation. Vue's implementation of store is non-traditional. Mutations and Actions are not coupled, meaning state mutations can occur without an action being fired, and can be triggered by components themselves. Additionally, Vue states are mutable, and do not need to supply a new state object during mutation, nor are they frozen objects.

States

App State

The App state is responsible for housing information related to how the widget works, such as, instance id, callback url, verify token, device id, widget rollout config, campaigns, websocket status, network status, user info, and user typing status.

Config State

The Config state houses information about the widget's configuration for the specific assistant.

Events State

The Events state houses only one piece of information, the nextCursor. The nextCursor is used when requesting new events, the API will only return events that the widget has not already seen.

Live State

The Live state houses information related to Embedded Web's ability to connect to our Twilio Flex plugin Contact Center.

Messages State

The messages state houses information about the conversation's messages.

SDK State

The SDK state houses authentication data once host sites call SDK authentication related methods

Survey State

The Survey state houses information for surveys after a human handoff session ends.

App State

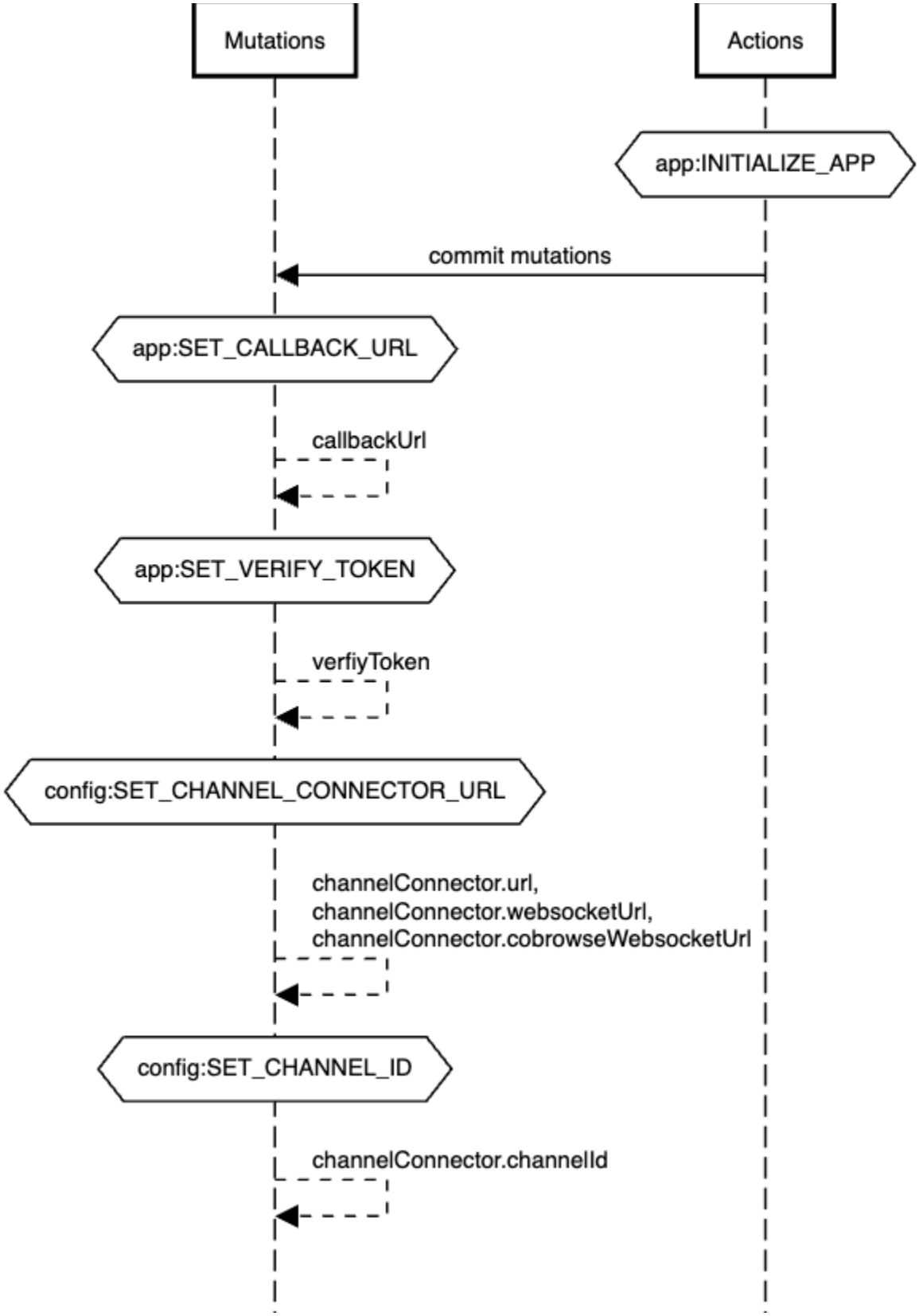
Overview
The App state is responsible for housing information related to how the widget works, such as, instance id, callback url, verify token, device id, widget rollout config, campaigns, websocket status, network status, user info, and user typing status.

Initial State

```
{
  instanceId: uuidv4(),
  leaderInstanceId: null,
  callbackUrl: null,
  verifyToken: null,
  deviceId: null,
  widget: {
    firstLoad: true,
    open: false,
    rolloutActive: false,
    rolloutTextOverride: null,
    intentOnRolloutClick: null,
    view: WIDGET_VIEW_PANELS,
    triggeredCampaigns: [],
  },
  userTyping: false,
  websocket: {
    status: WEBSOCKET_STATUS_DISCONNECTED,
  },
  network: {
    status: NETWORK_STATUS_CONNECTED,
    errorCount: 0,
  },
  userInfoCollectionData: {
    firstName: '',
    lastName: '',
    email: '',
    phone: '',
  },
  modal: null,
}
```

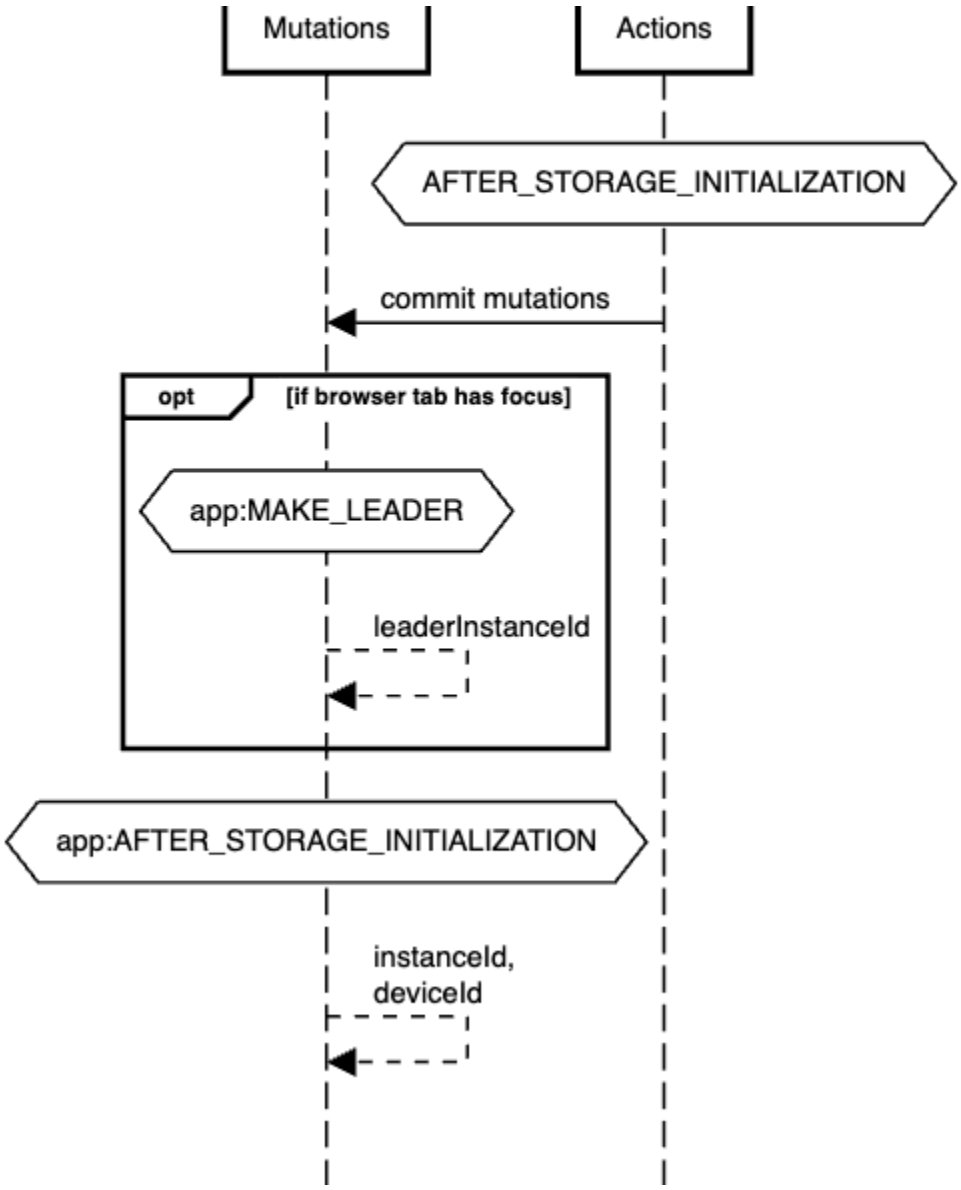
Actions

INITIALIZE_APP



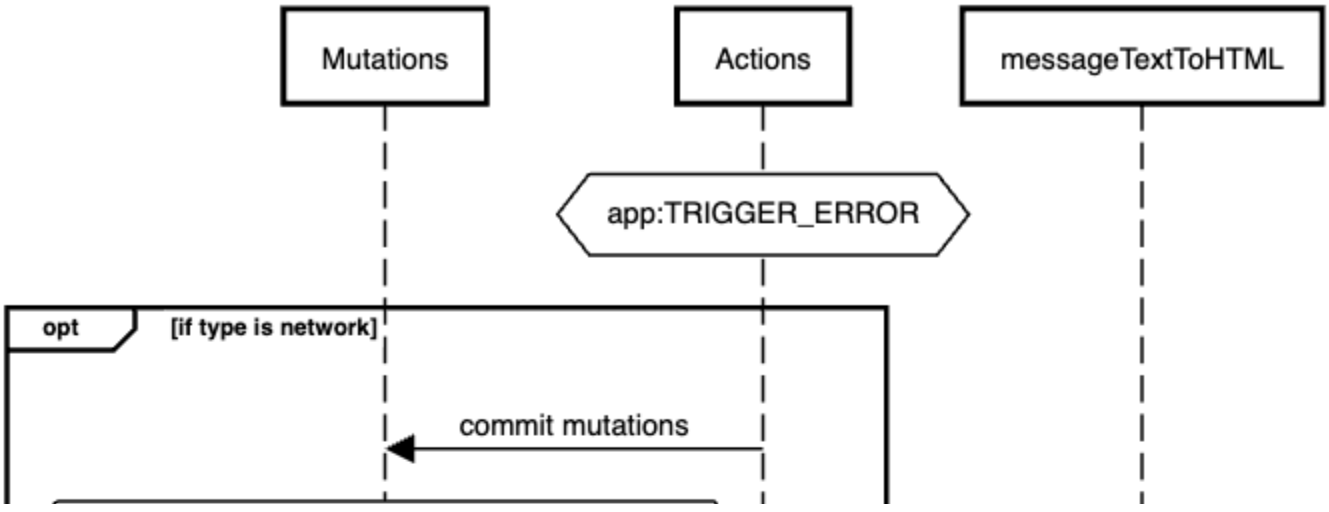
AFTER_STORAGE_INITIALIZATION

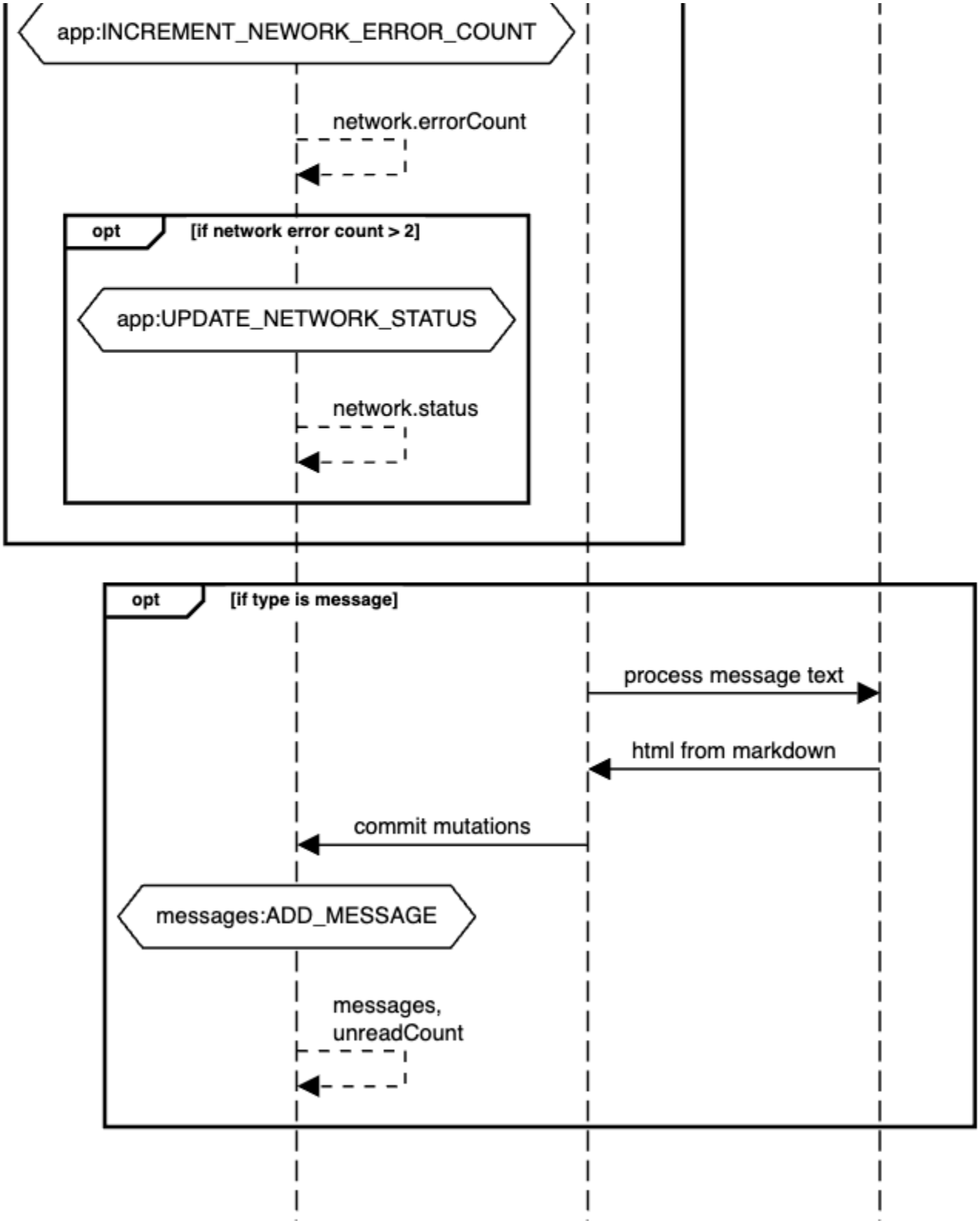
EW App AFTER_STORAGE_INITIALIZATION



TRIGGER_ERROR

EW TRIGGER_ERROR Action Flow



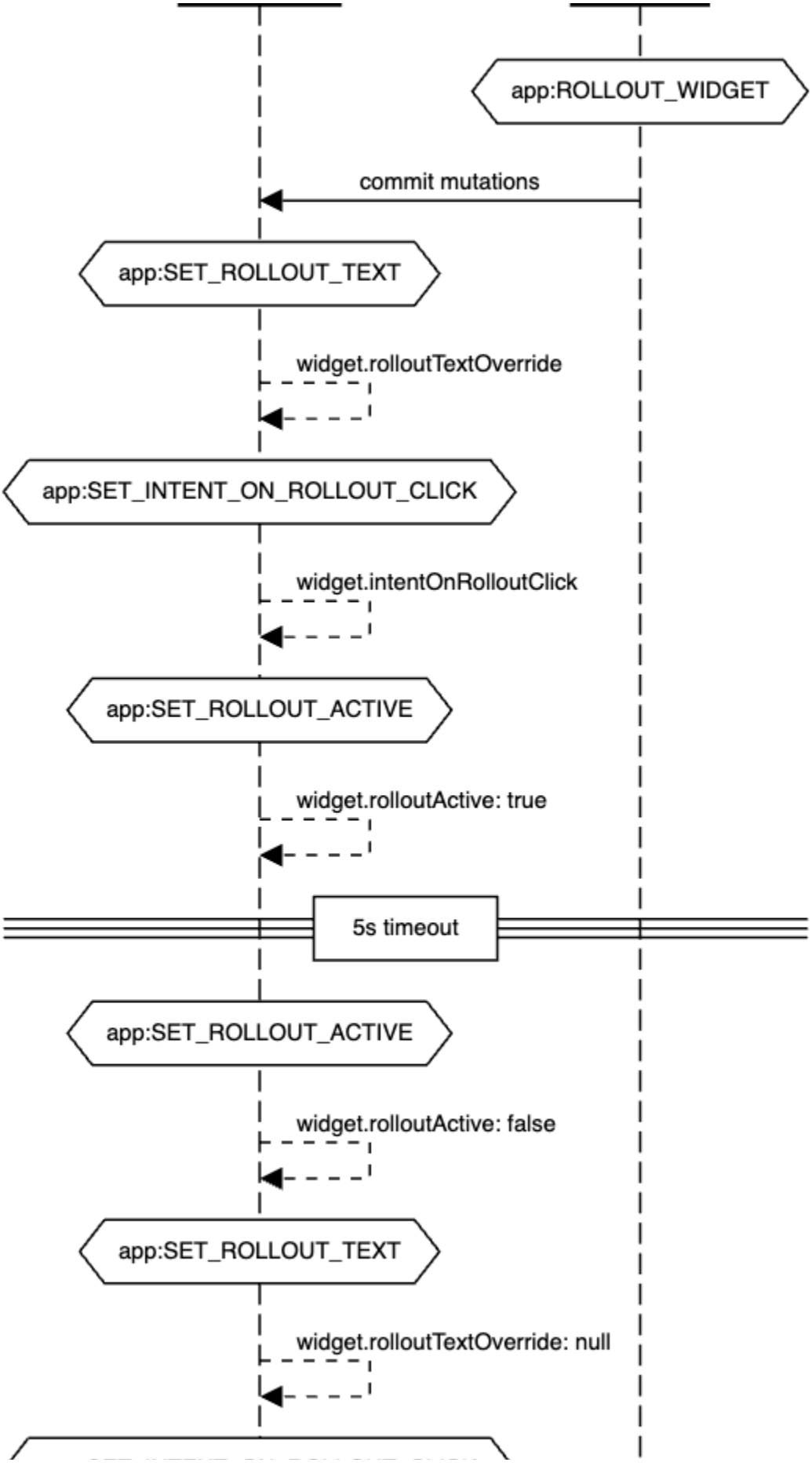


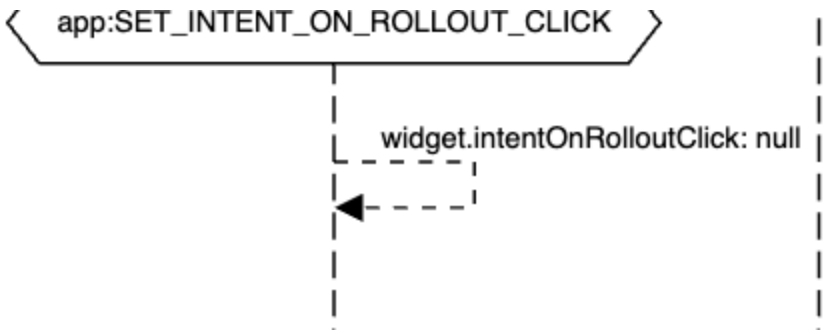
ROLLOUT_WIDGET

EW ROLLOUT_WIDGET Action Flow

Mutations

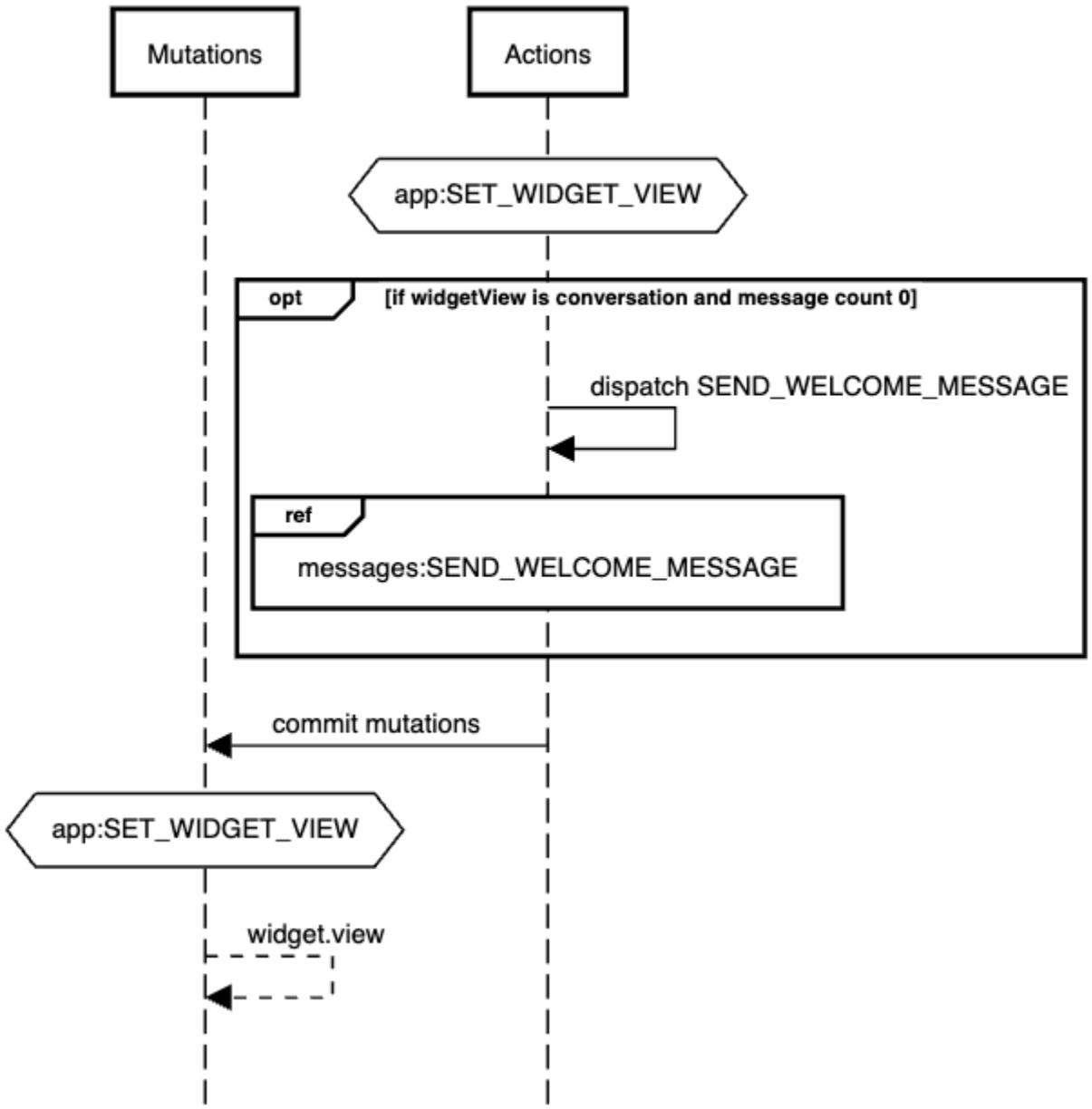
Actions


app:ROLLOUT_WIDGET
commit mutations
app:SET_ROLLOUT_TEXT
widget.rolloutTextOverride
app:SET_INTENT_ON_ROLLOUT_CLICK
widget.intentOnRolloutClick
app:SET_ROLLOUT_ACTIVE
widget.rolloutActive: true
5s timeout
app:SET_ROLLOUT_ACTIVE
widget.rolloutActive: false
app:SET_ROLLOUT_TEXT
widget.rolloutTextOverride: null



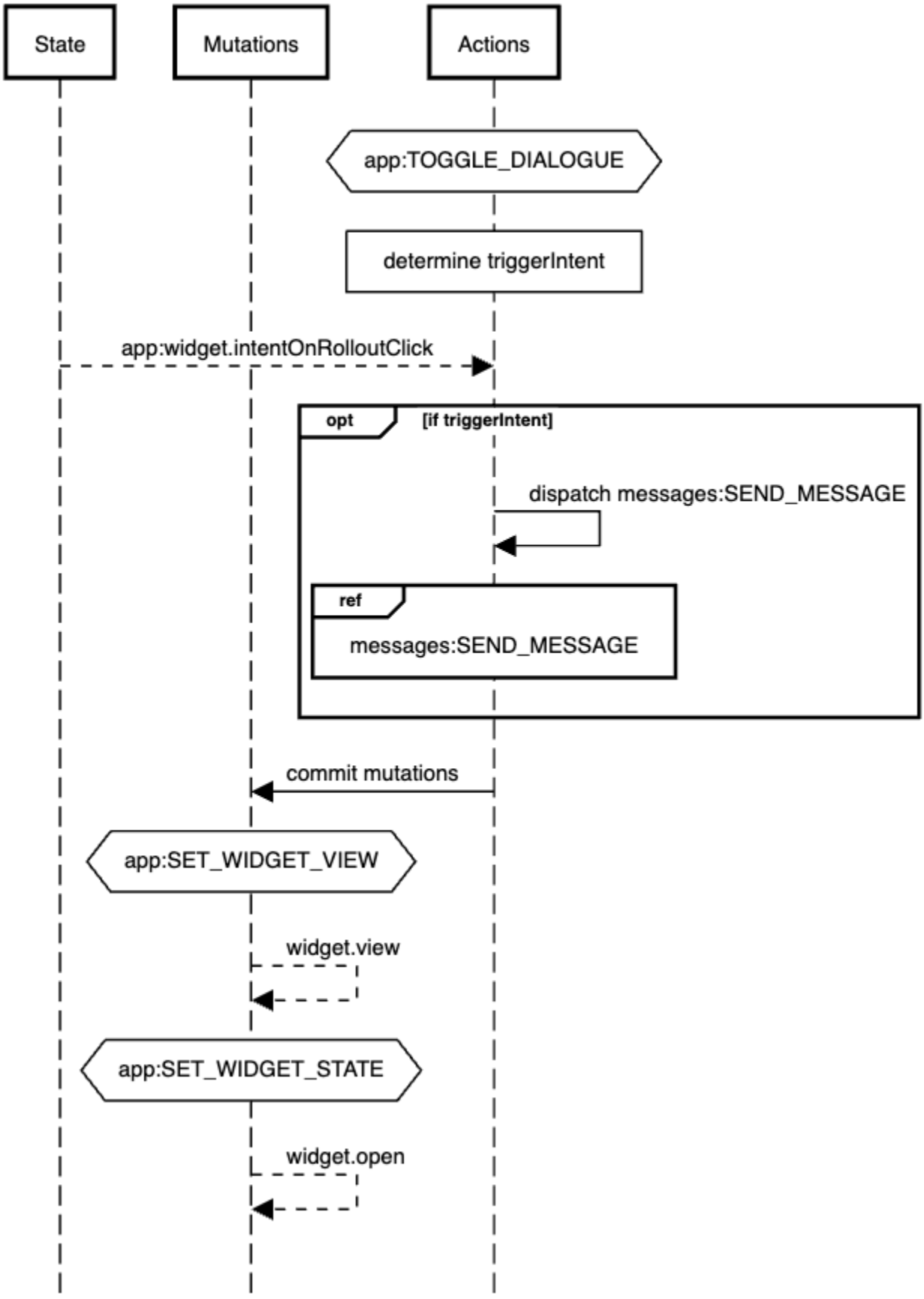
SET_WIDGET_VIEW

EW SET_WIDGET_VIEW Action Flow

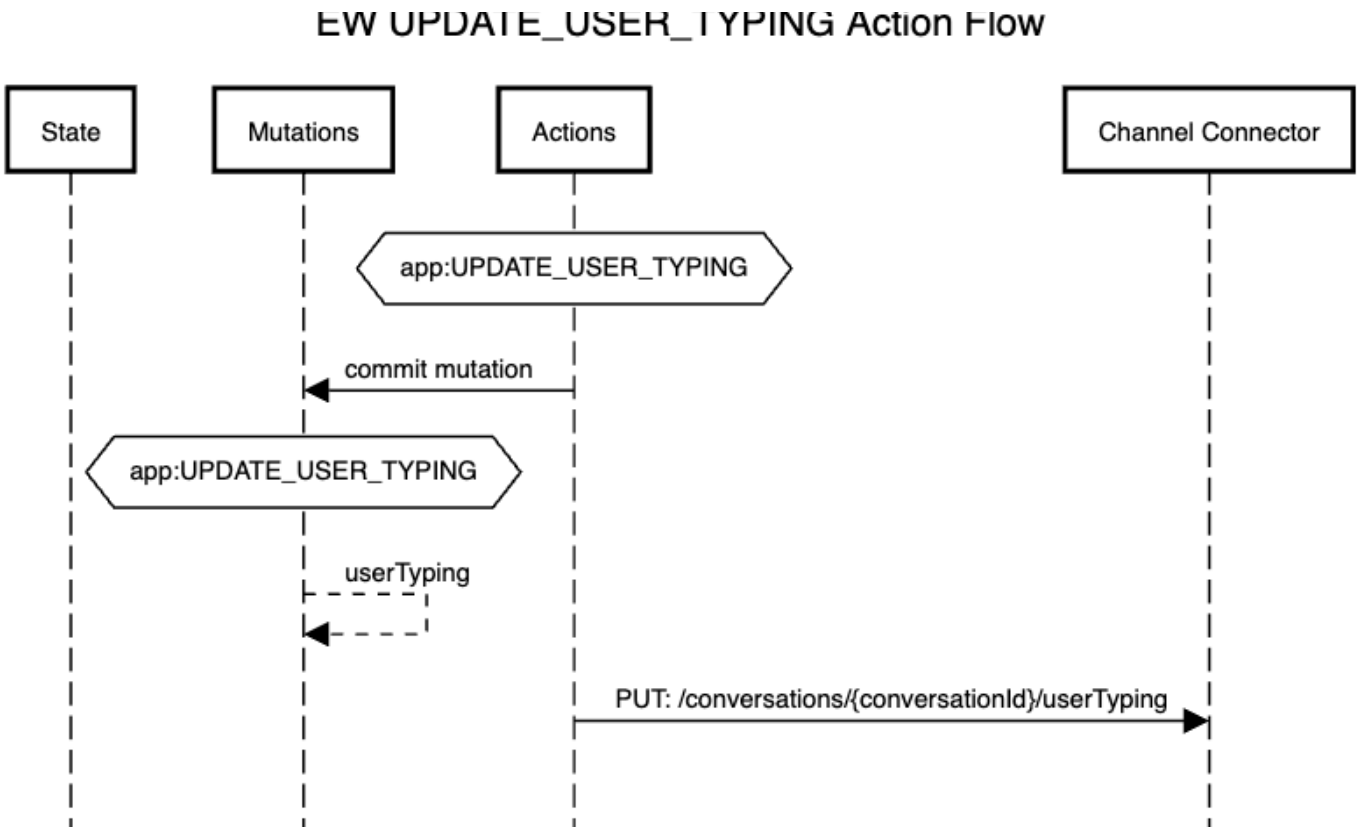


TOGGLE_DIALOGUE

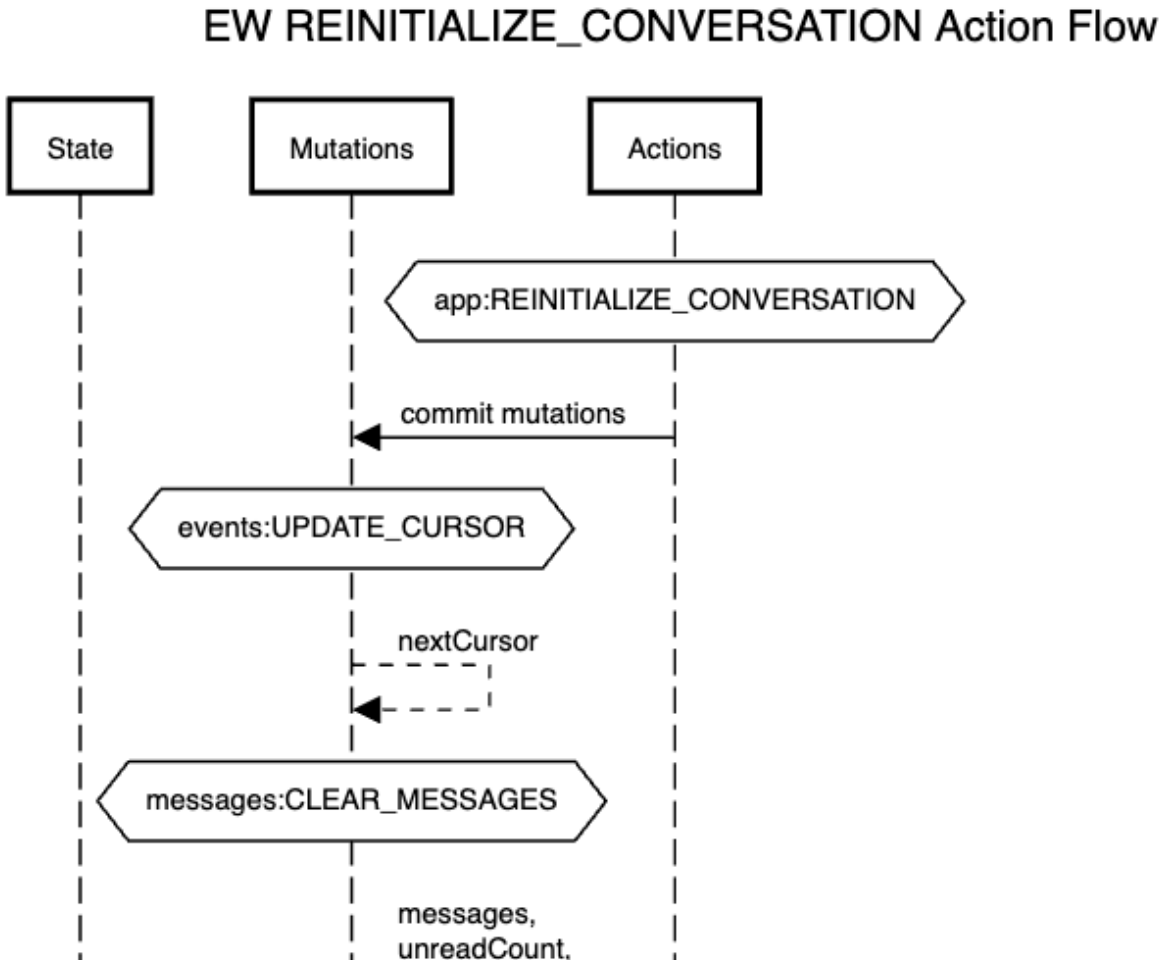
EW TOGGLE_DIALOGUE Action Flow

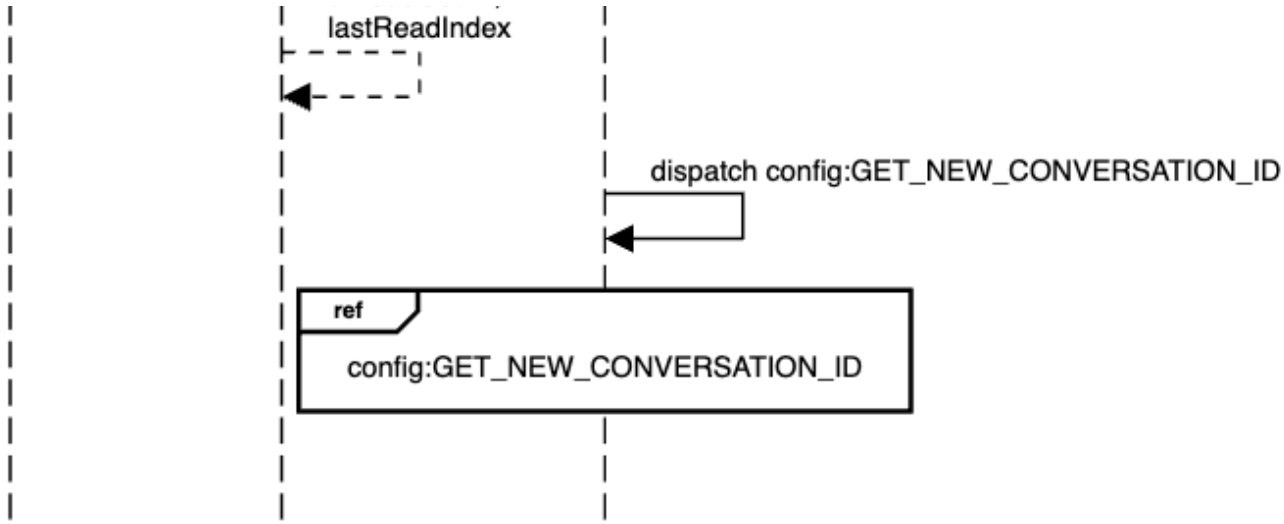


UPDATE_USER_TYPING



REINITIALIZE_CONVERSATION





Config State

Overview

The Config state houses information about the widget's configuration for the specific assistant.

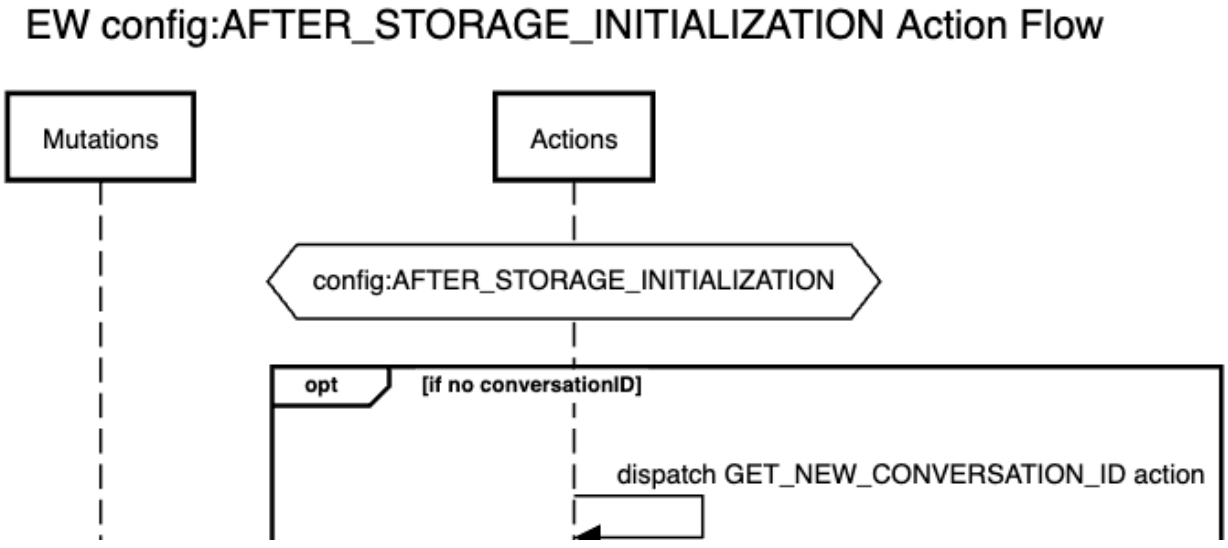
Initial State

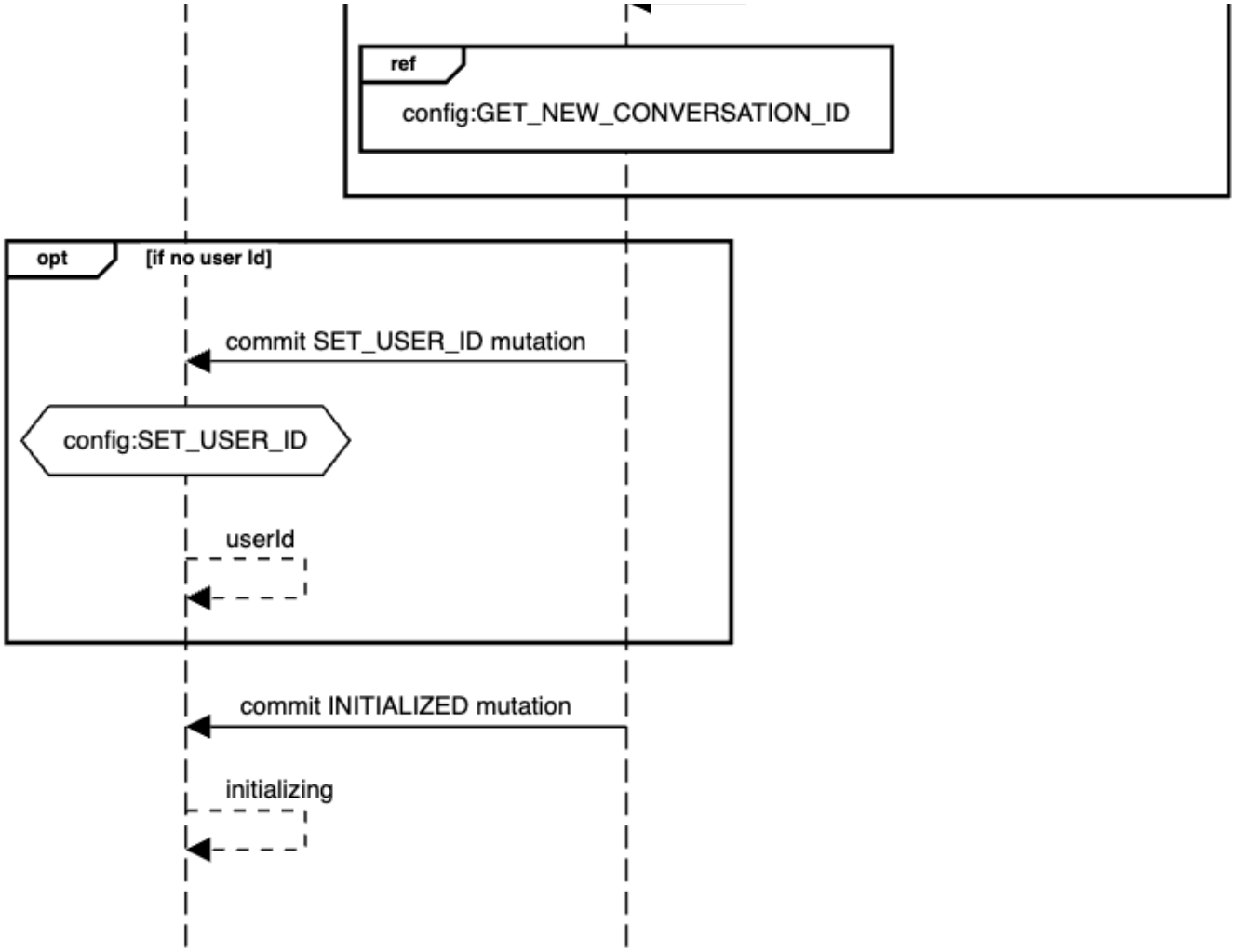
```
{
  initializing: true,
  channelConnector: {
    cobrowseWebsocketUrl: null,
    websocketUrl: null,
    url: null,
    channelId: null,
  },
  conversationId: null,
  userId: null,
  tenant: {
    loaded: false,
    campaigns: {
      data: null,
    },
    brand: {
      color: {
        hex: null,
        rgb: null,
        contrast: {
          hex: null,
          rgb: null,
        },
      },
      cta: {
        color: {
          hex: null,
          rgb: null,
```

```
        },
      },
      logo: null,
      heading: null,
      tagline: null,
    },
    assistant: {
      name: null,
      tagline: null,
    },
    chatPanel: {
      heading: null,
      content: null,
    },
    conversation: {
      inputPlaceholder: null,
    },
    widget: {
      rollOut: {
        onHover: null,
        onFirstLoad: false,
      },
      triggerQueryString: null,
      hide: false,
    },
    faq: {
      enabled: false,
      slots: [],
    },
  },
}
```

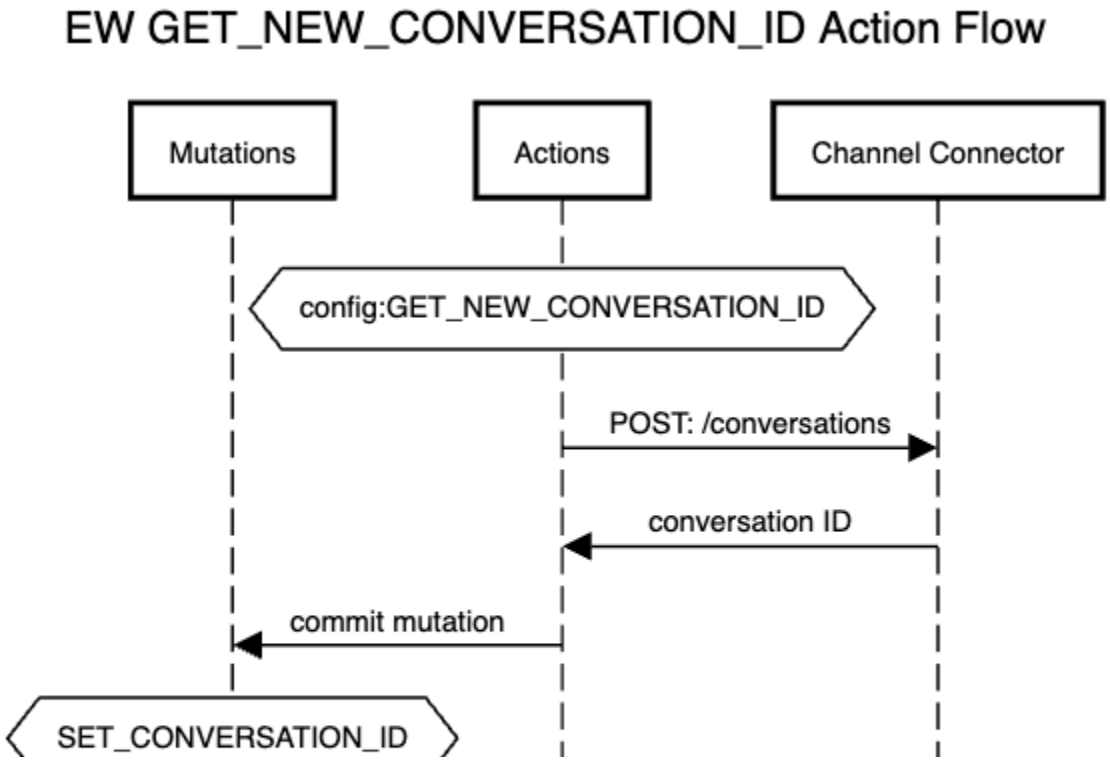
Actions

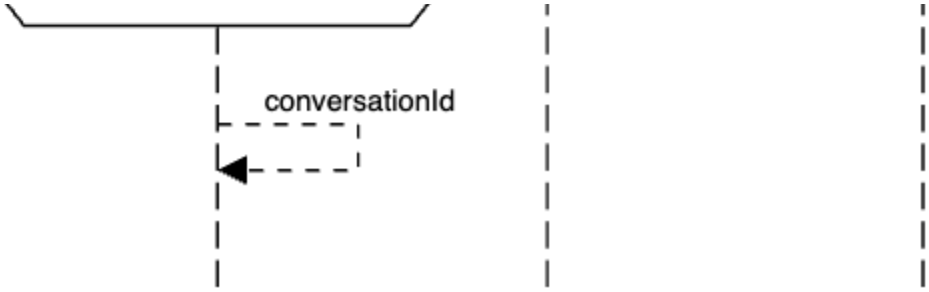
AFTER_STORAGE_INITIALIZATION



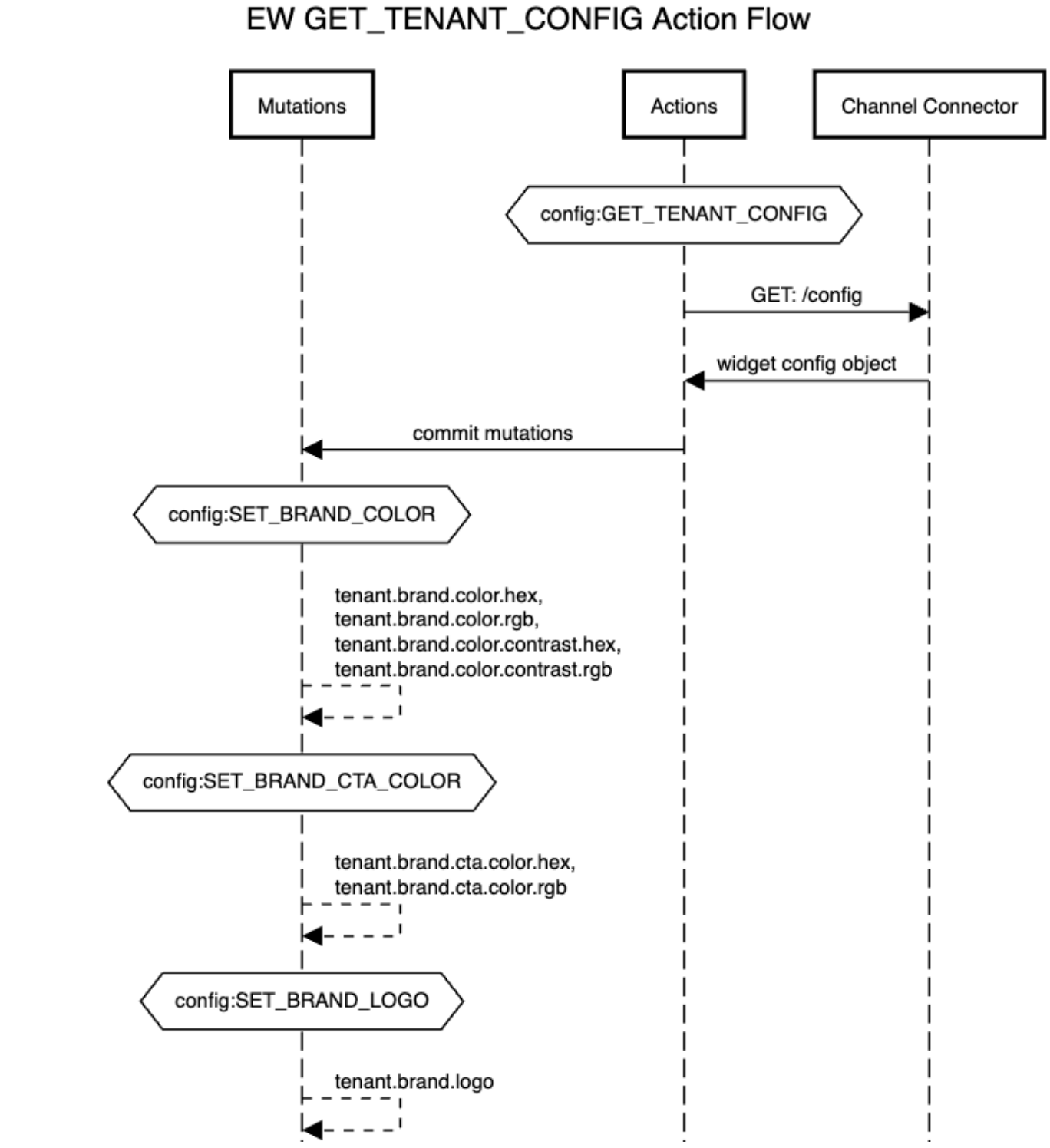


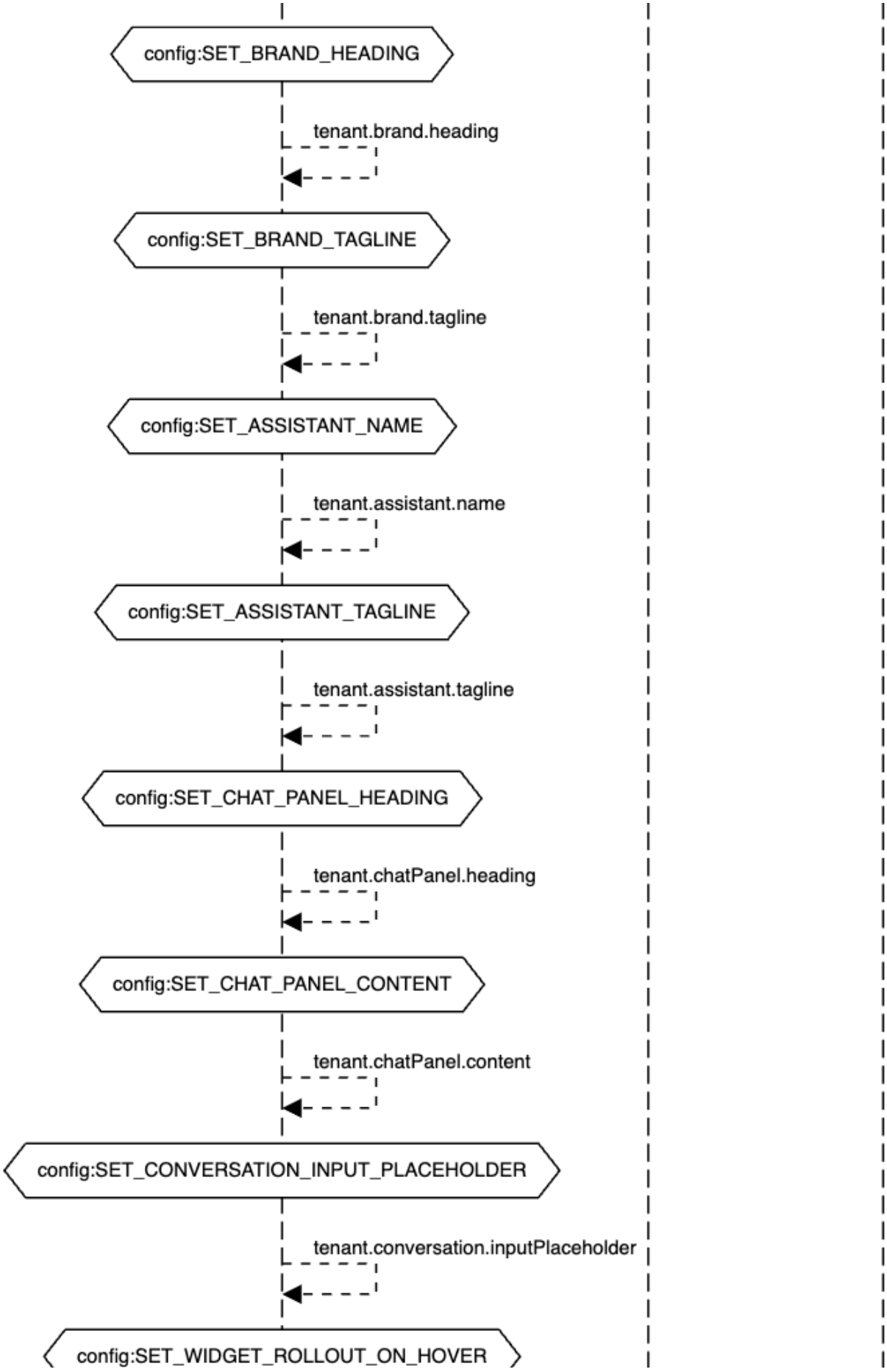
GET_NEW_CONVERSATION_ID



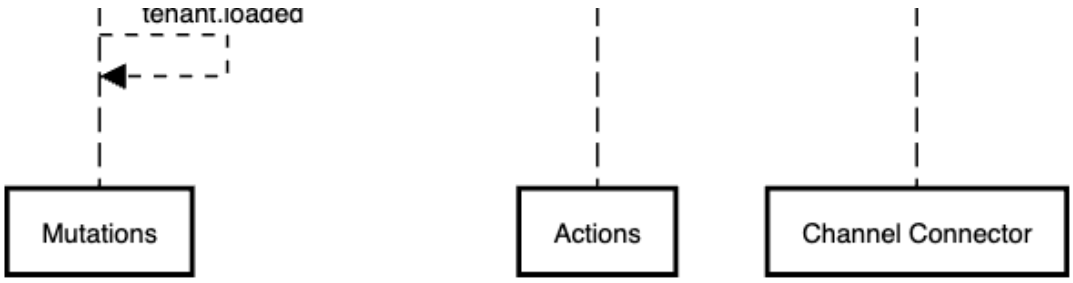


GET_TENANT_CONFIG




config:SET_BRAND_HEADING
tenant.brand.heading
config:SET_BRAND_TAGLINE
tenant.brand.tagline
config:SET_ASSISTANT_NAME
tenant.assistant.name
config:SET_ASSISTANT_TAGLINE
tenant.assistant.tagline
config:SET_CHAT_PANEL_HEADING
tenant.chatPanel.heading
config:SET_CHAT_PANEL_CONTENT
tenant.chatPanel.content
config:SET_CONVERSATION_INPUT_PLACEHOLDER
tenant.conversation.inputPlaceholder
config:SET_WIDGET_ROLLOUT_ON_HOVER


tenant.widget.rollOut.onHover
config:SET_WIDGET_ROLLOUT_ON_FIRST_LOAD
tenant.widget.rollOut.onFirstLoad
config:SET_WIDGET_TRIGGER_QUERY_STATE
tenant.widget.triggerQueryString
config:SET_WIDGET_HIDE
tenant.widget.hide
config:SET_CAMPAIGNS
tenant.campaigns.data
config:SET_FAQ_ENABLED
tenant.faq.enabled
config:SET_FAQ_SLOTS
tenant.faq.slots
config:SET_TENANT_LOADED



Events State

Overview

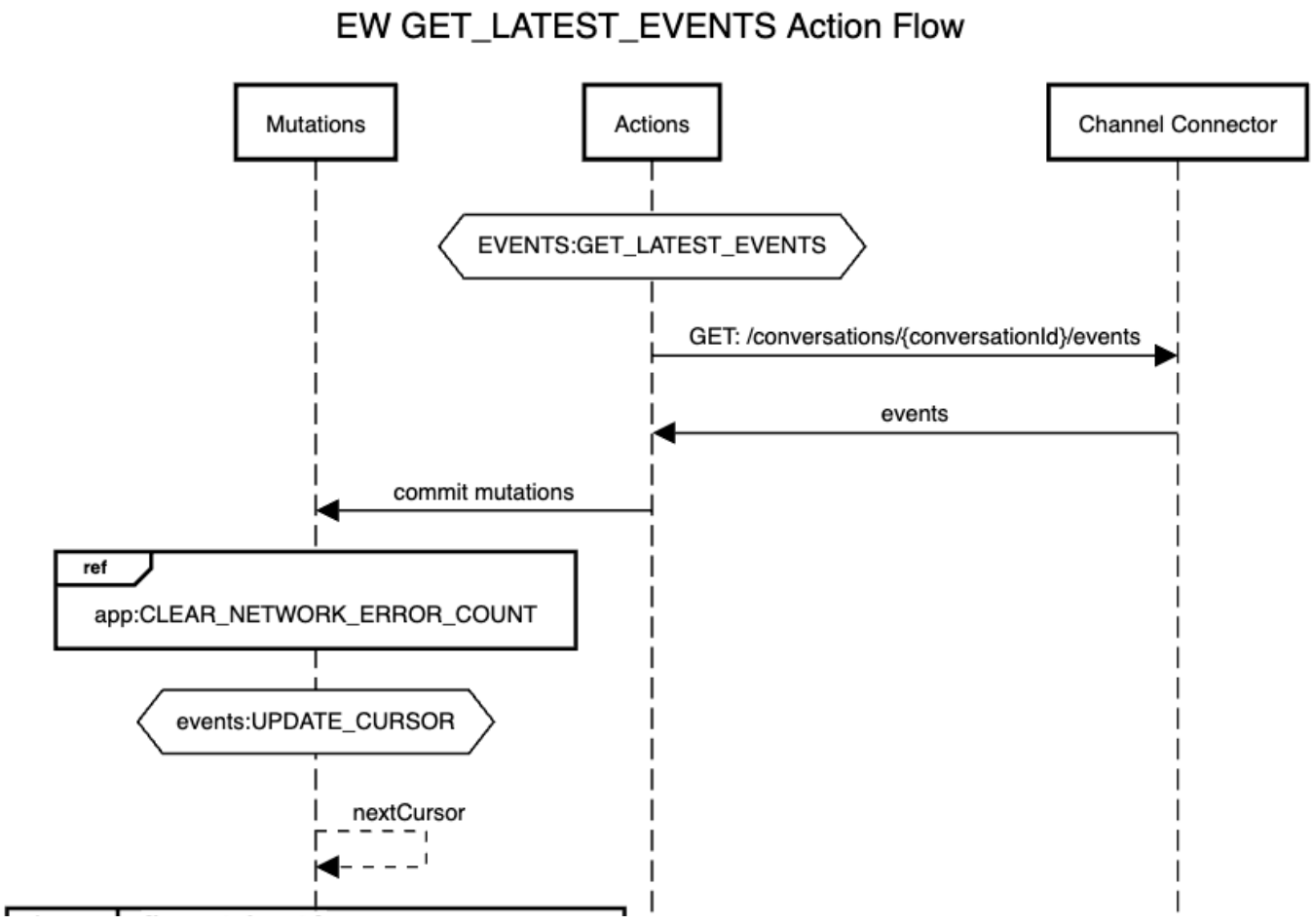
The Events state houses only one piece of information, the `nextCursor`. The nextCursor is used when requesting new events, the API will only return events that the widget has not already seen.

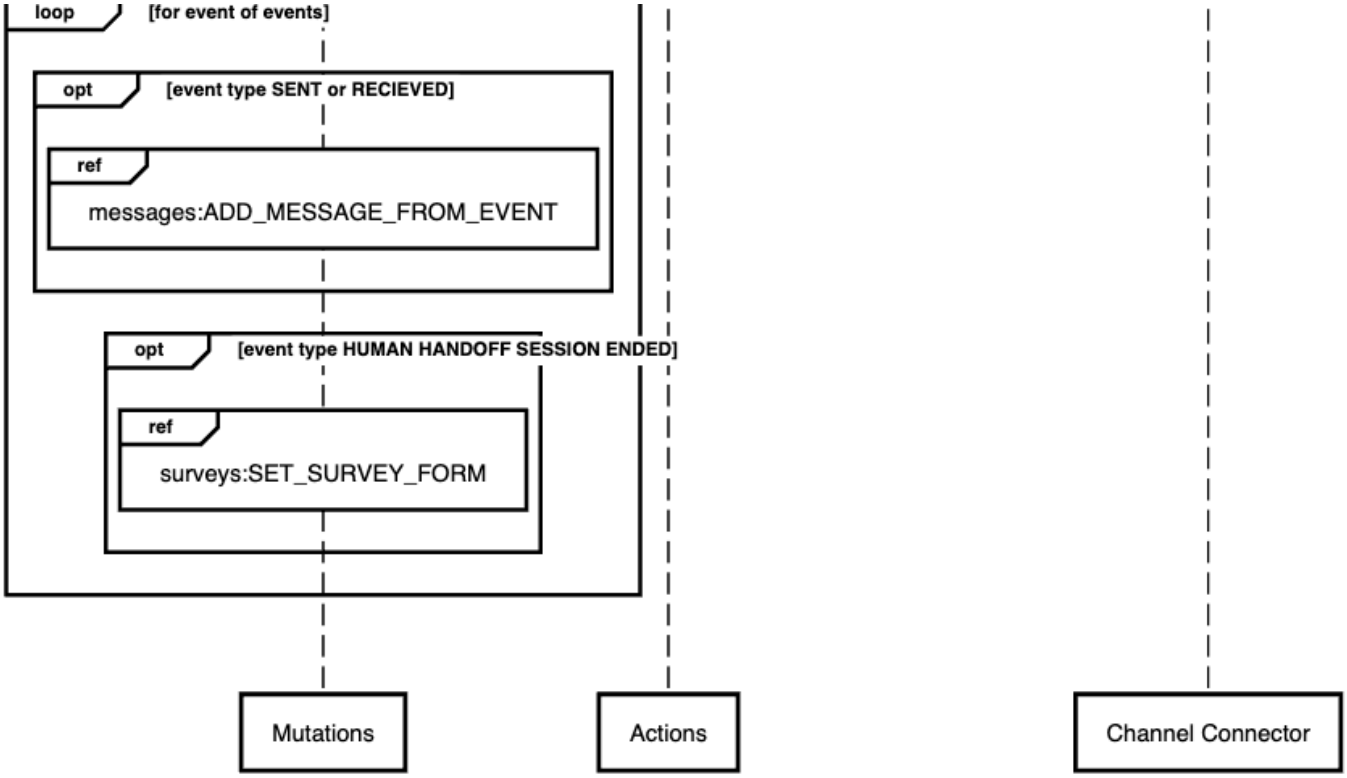
Initial State

```
{
  nextCursor: null,
}
```

Actions

GET_LATEST_EVENTS





Live State

Overview

The Live state houses information related to Embedded Web's ability to connect to our Twilio Flex plugin Contact Center.

This state, often has its mutations directly triggered from components or functions, circumventing actions.

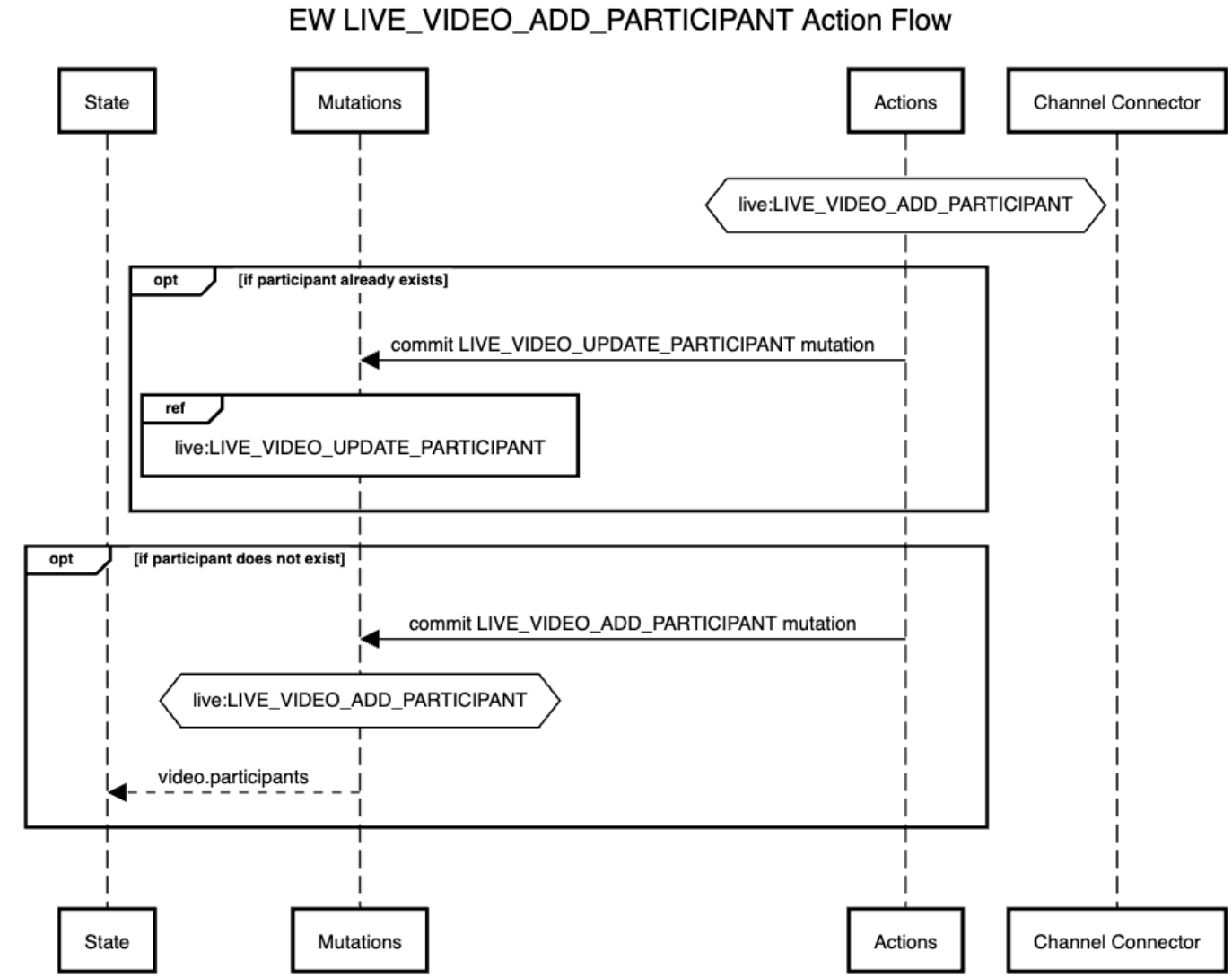
Initial State

```
{
  video: {
    status: LIVE_VIDEO_INACTIVE,
    token: null,
    type: null,
    camera: {
      enabled: true,
    },
    microphone: {
      enabled: true,
    },
    participants: [],
    participantsToRefresh: [],
    tracks: {
      local: {
        audio: null,
        video: null,
      },
    },
```

```
  },
  cobrowse: {
    status: LIVE_COBROWSE_INACTIVE,
  },
  instance: null,
}
```

Actions

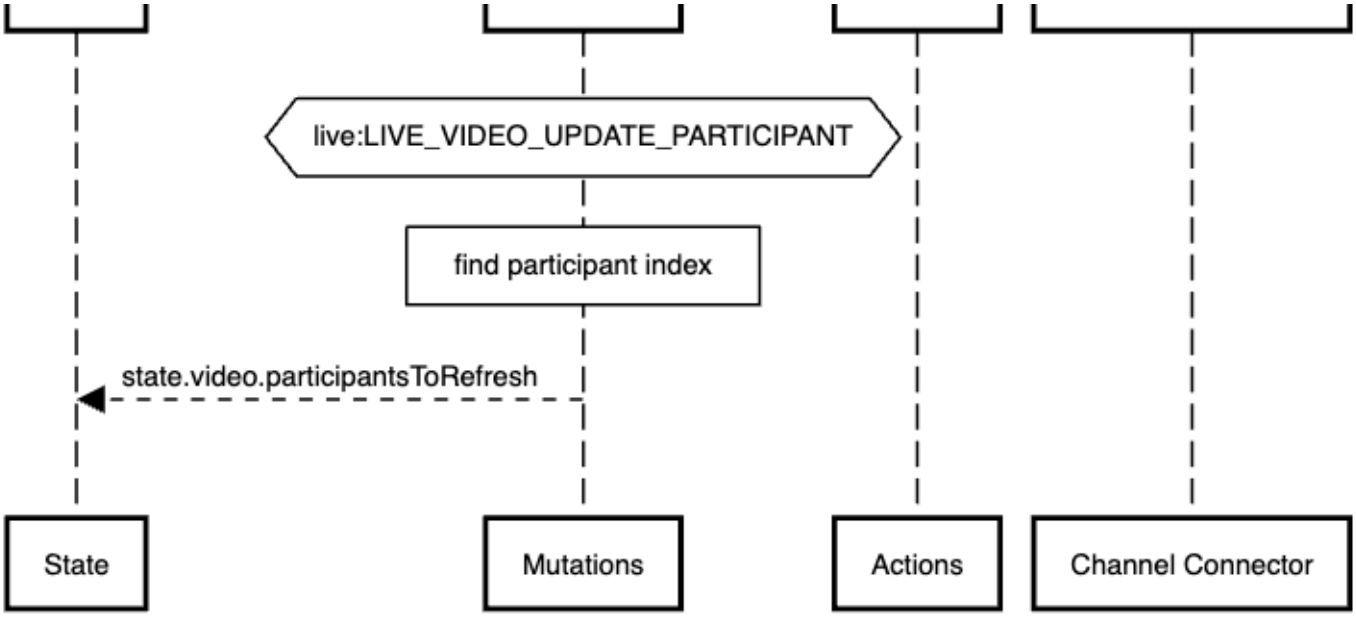
LIVE_VIDEO_ADD_PARTICIPANT



Mutations

These mutations do not occur in an action flow, instead being triggered by our Video class, so we will detail them there.

LIVE_VIDEO_UPDATE_PARTICIPANT



SET_LIVE_INSTANCE

updates `instance` on live state

SET_LIVE_VIDEO_TOKEN

updates `video.token` on live state

SET_LIVE_VIDEO_TYPE

updates `video.type` on live state

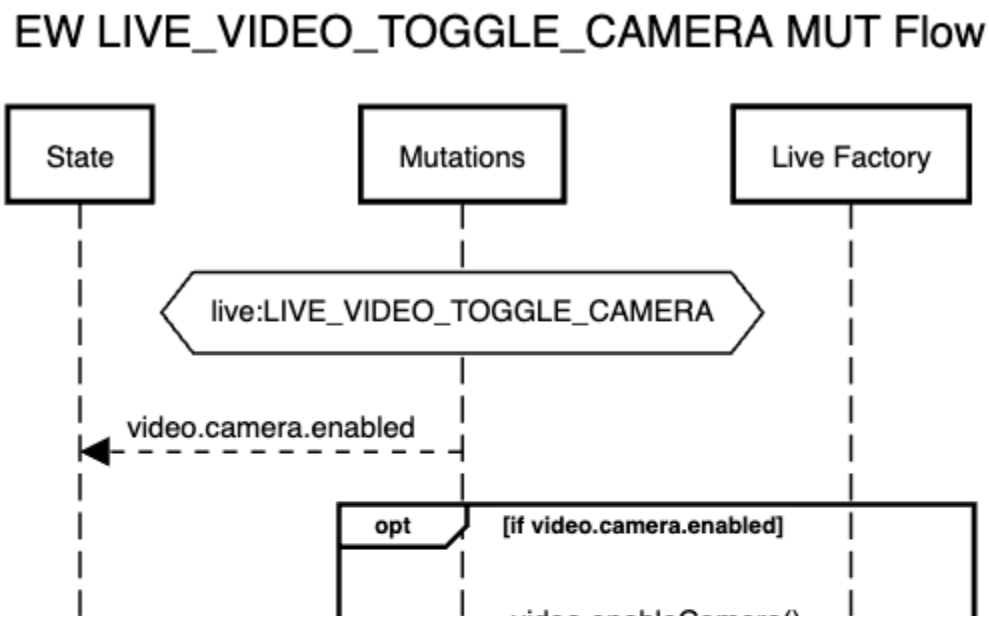
SET_LIVE_VIDEO_TRACK_LOCAL_VIDEO

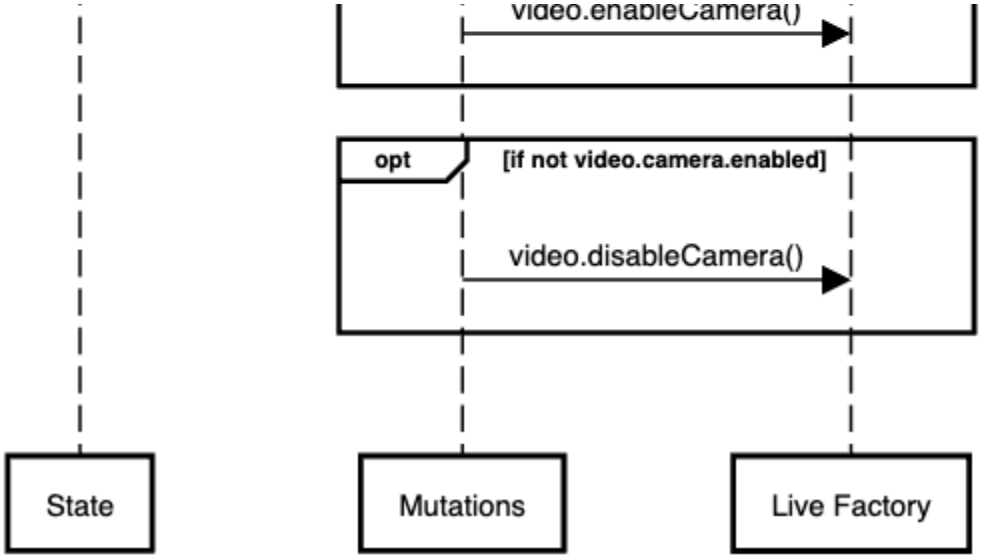
updates `video.tracks.local.video` on the live state

SET_LIVE_VIDEO_TRACK_LOCAL_AUDIO

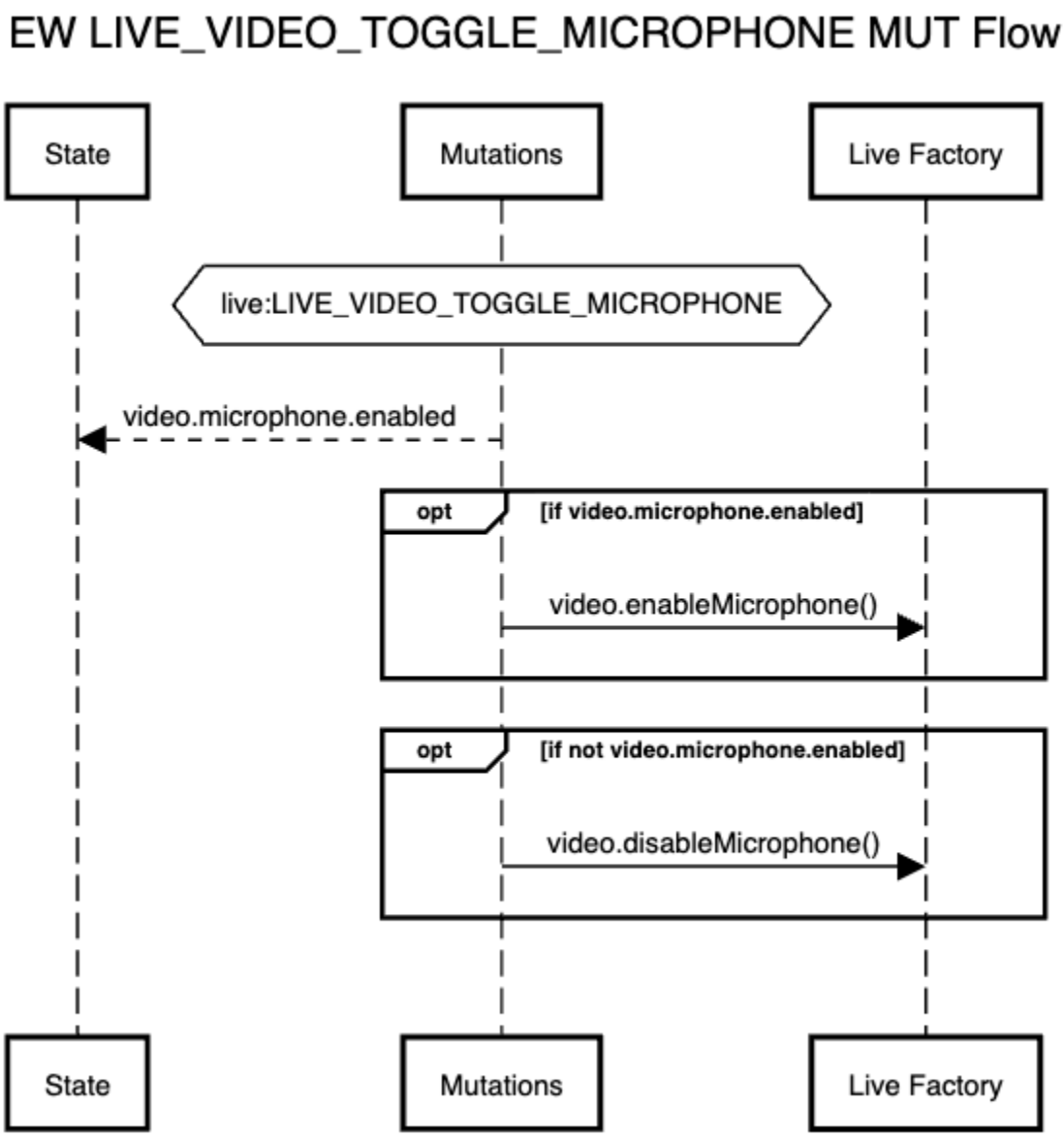
updates `video.tracks.local.audio` on the live state

LIVE_VIDEO_TOGGLE_CAMERA

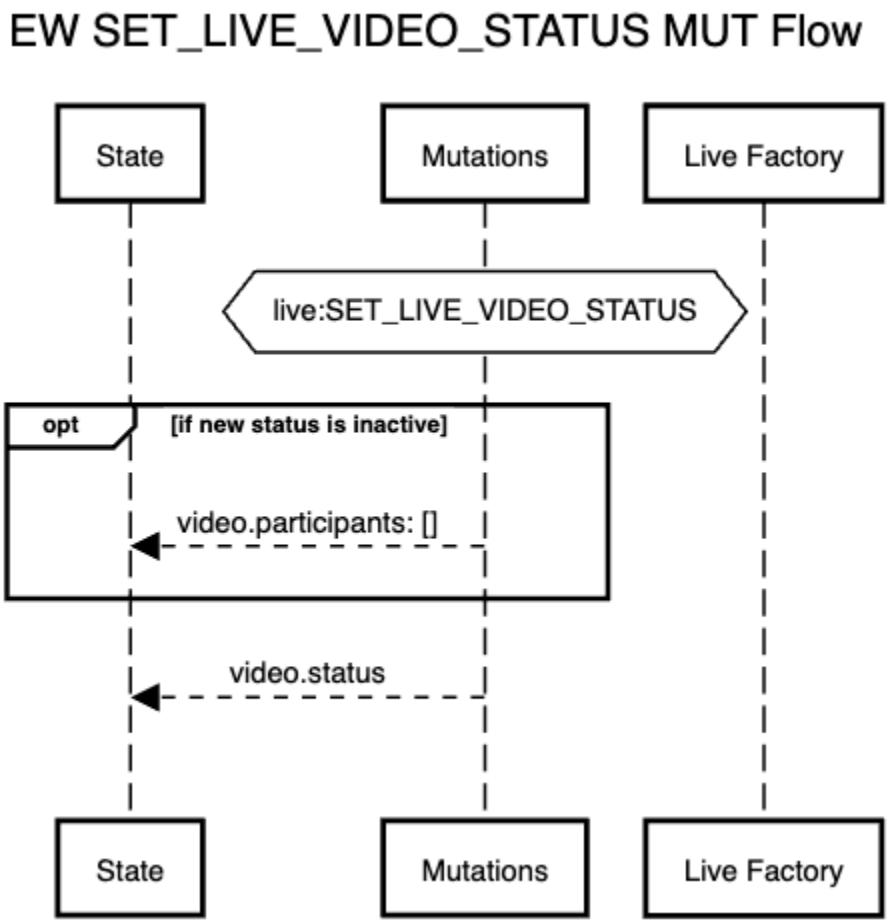




LIVE_VIDEO_TOGGLE_MICROPHONE



SET_LIVE_VIDEO_STATUS



SET_LIVE_COBROWSE_STATUS

updates `cobrowse.status` on live state

Messages State

Overview

The messages state houses information about the conversation's messages.

Initial State

```
{
  lastReadIndex: 0,
  messages: [],
  unreadCount: 0,
  quickReplyMap: {},
}
```

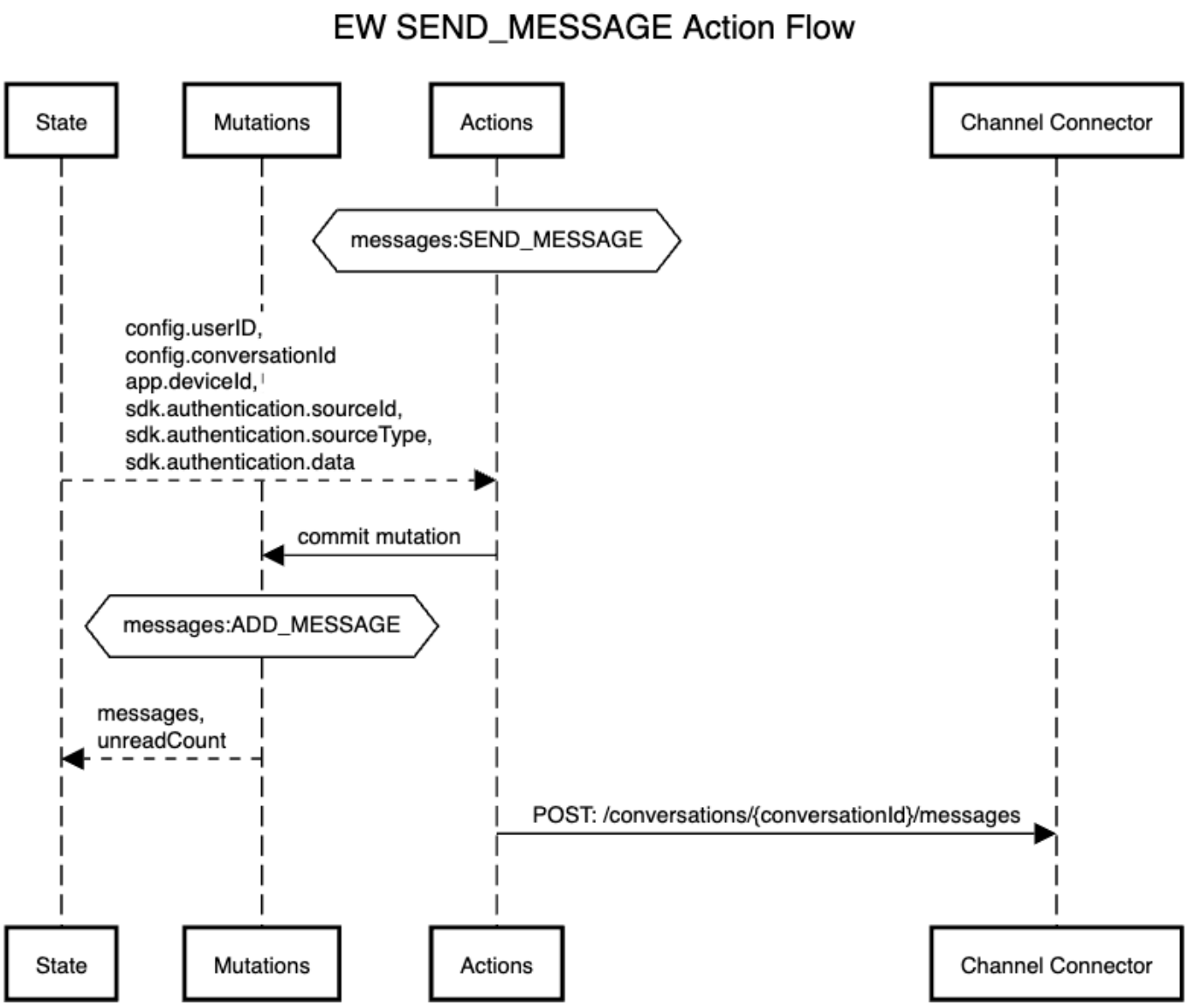
Actions

SEND_WELCOME_MESSAGE

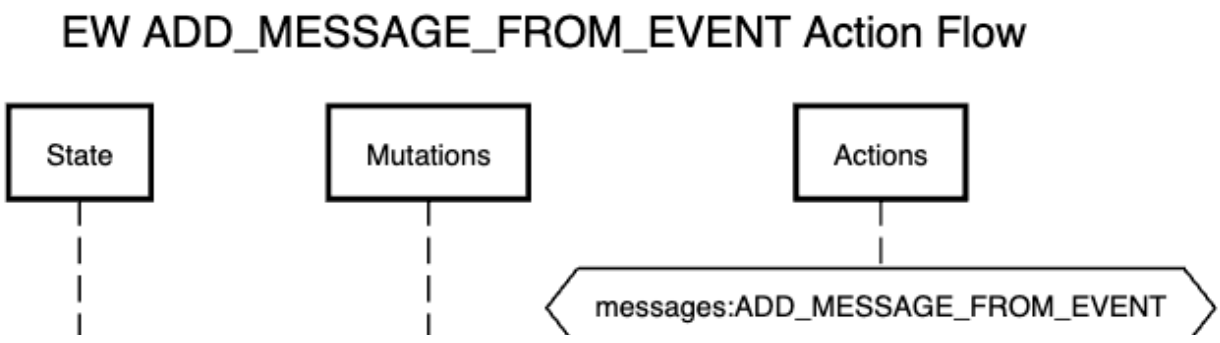
The SEND_WELCOME_MESSAGE action just dispatches a `SEND_MESSAGE` action with the following payload.

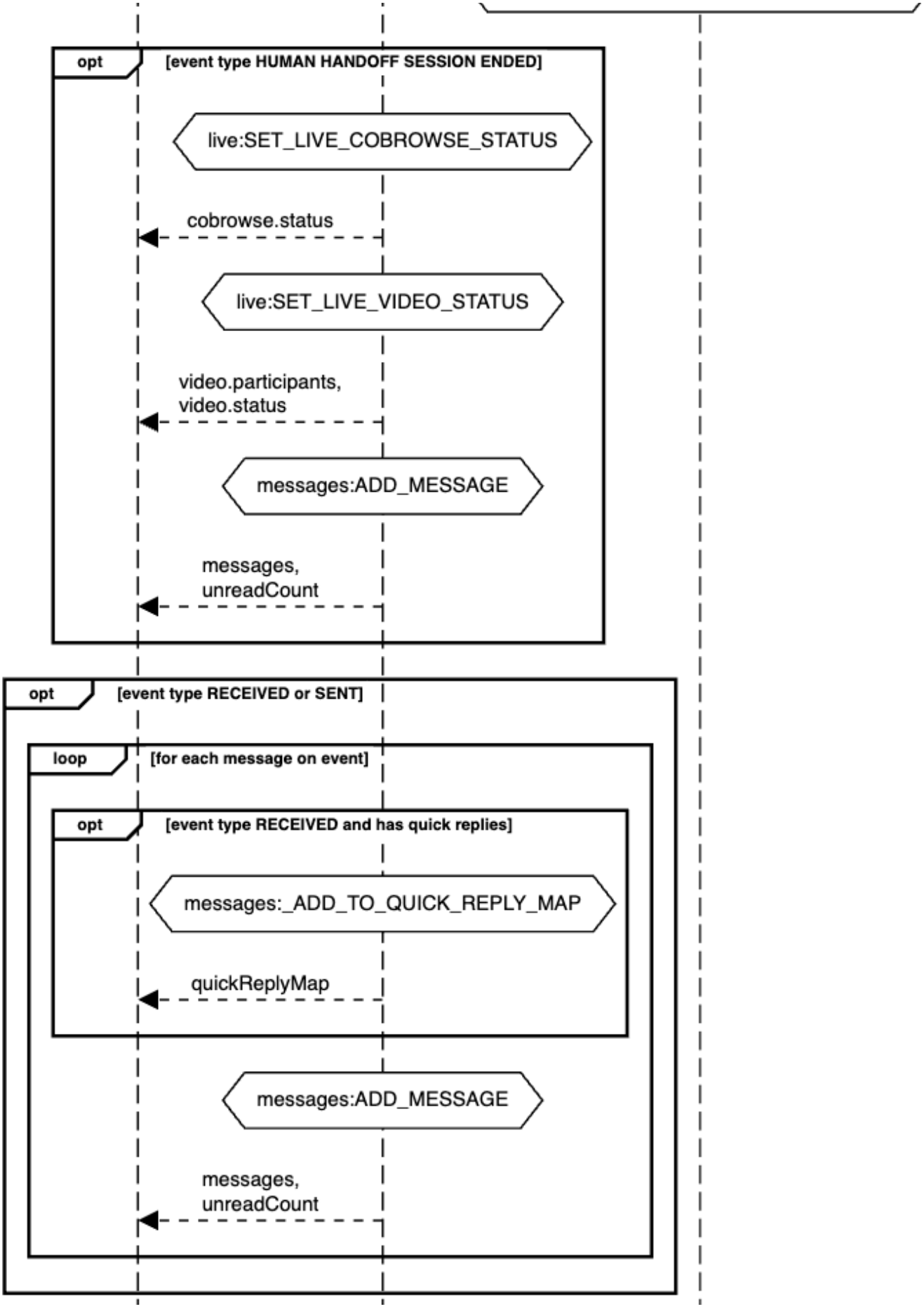
```
{
  intent: 'CONVERSATION_INITIALIZE'
}
```

SEND_MESSAGE

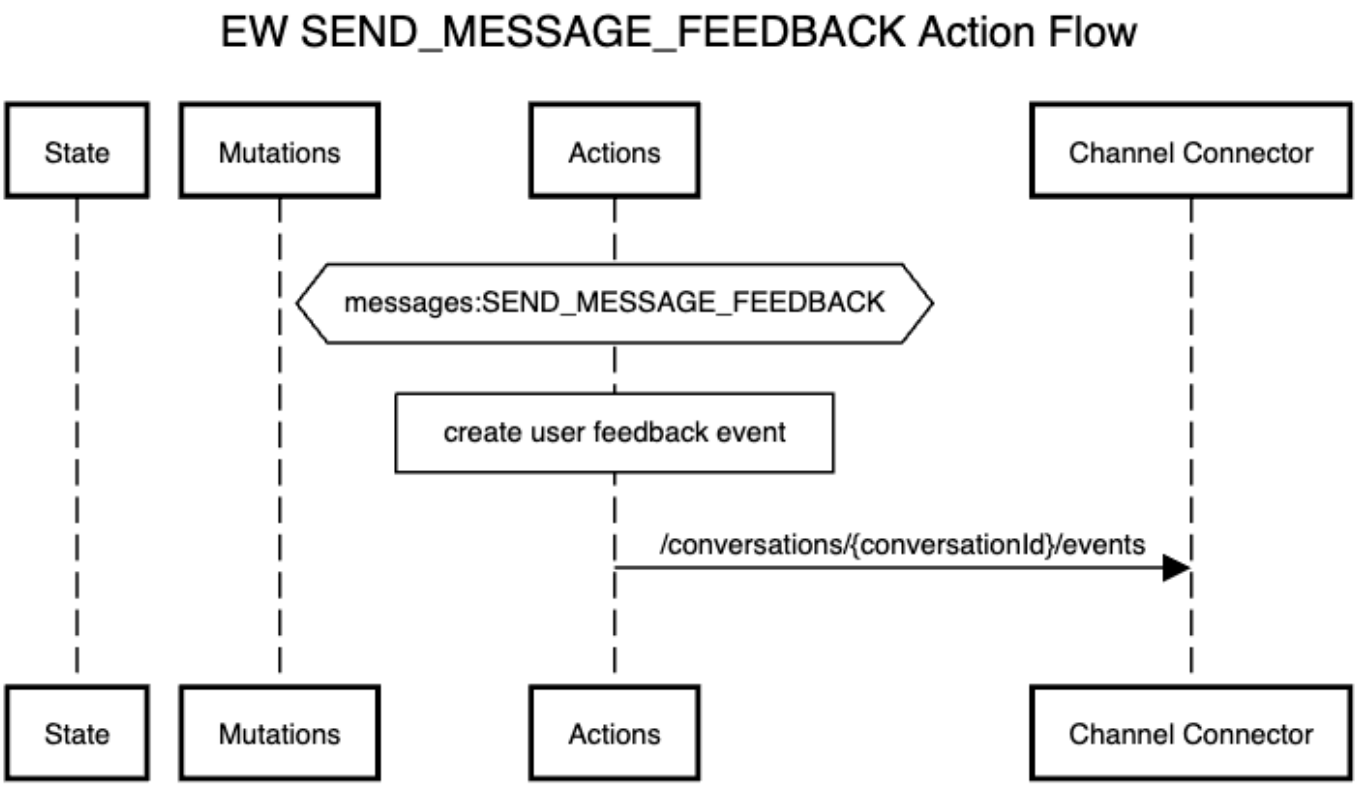


ADD_MESSAGE_FROM_EVENT




opt
[event type HUMAN HANDOFF SESSION ENDED]
live:SET_LIVE_COBROWSE_STATUS
cobrowse.status
live:SET_LIVE_VIDEO_STATUS
video.participants,
video.status
messages:ADD_MESSAGE
messages,
unreadCount
opt
[event type RECEIVED or SENT]
loop
[for each message on event]
opt
[event type RECEIVED and has quick replies]
messages:_ADD_TO_QUICK_REPLY_MAP
quickReplyMap
messages:ADD_MESSAGE
messages,
unreadCount

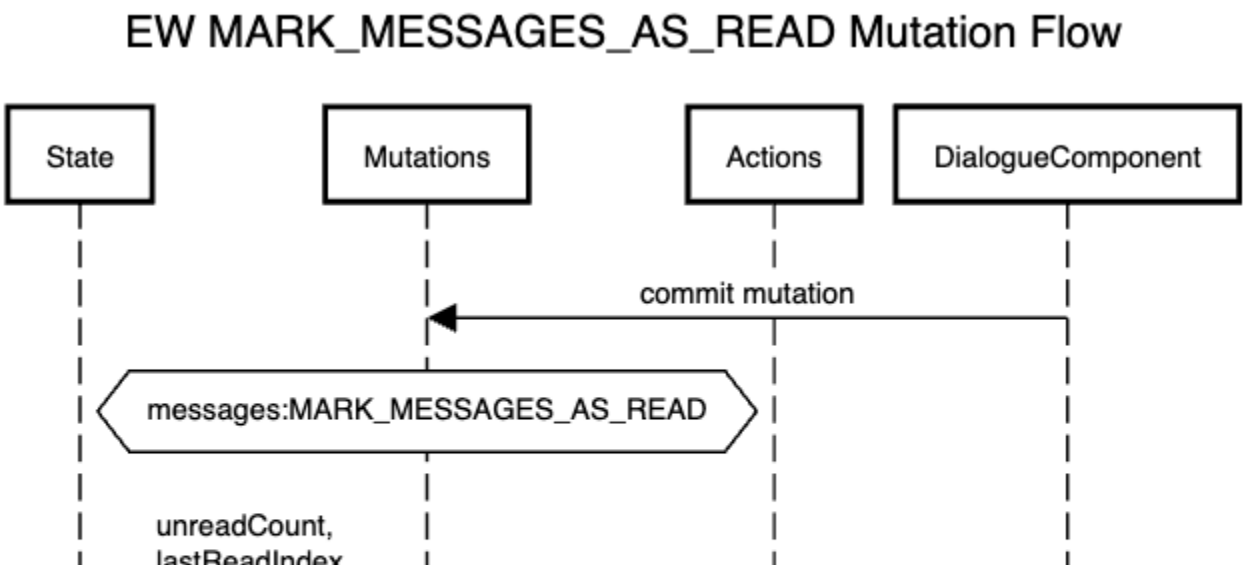
SEND_MESSAGE_FEEDBACK

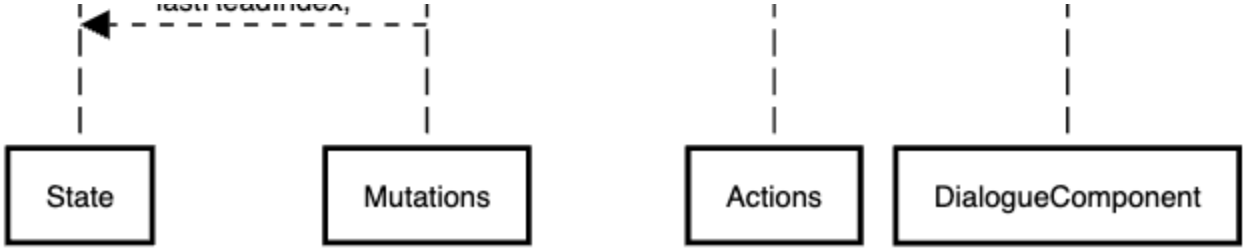


Mutations

These mutations are not accounted for in other action flows

MARK_MESSAGES_AS_READ





SDK State

Overview

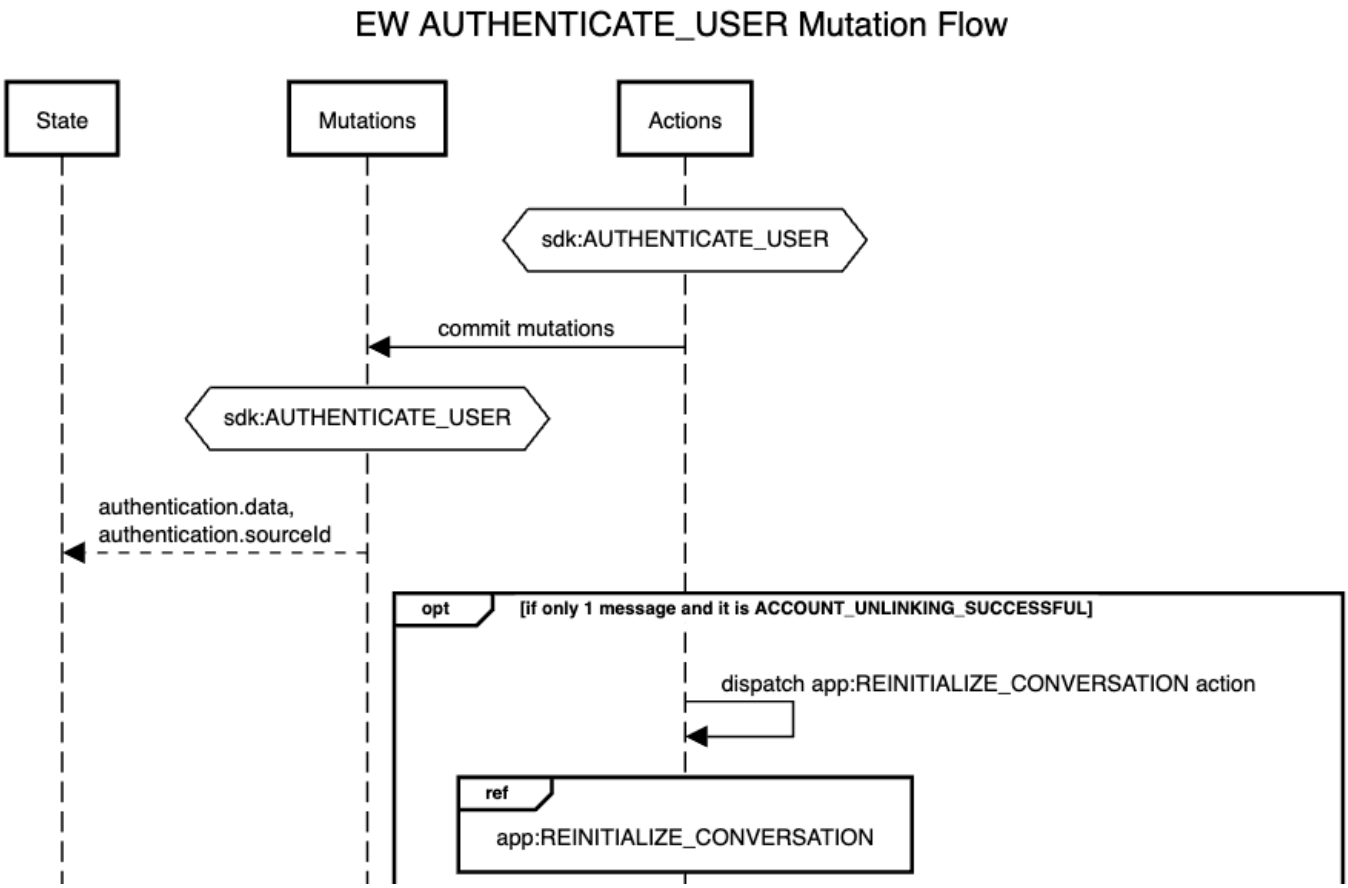
The SDK state houses authentication data once host sites call SDK authentication related methods

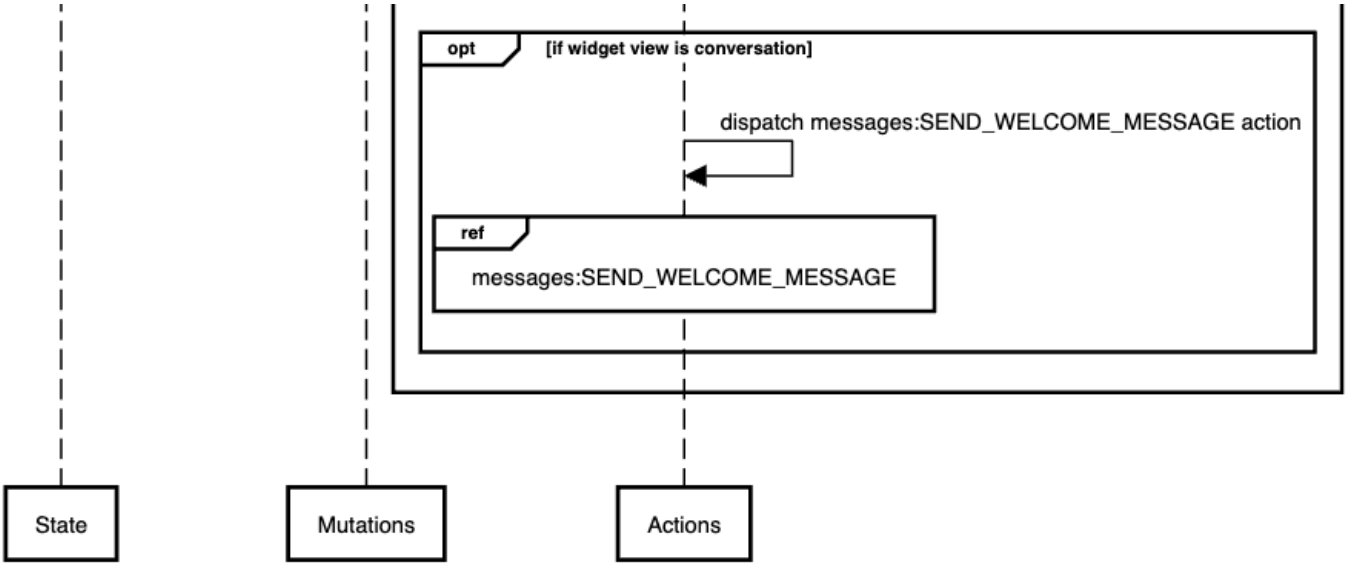
Initial State

```
{
  authentication: {
    data: null,
    sourceId: null,
  },
}
```

Actions

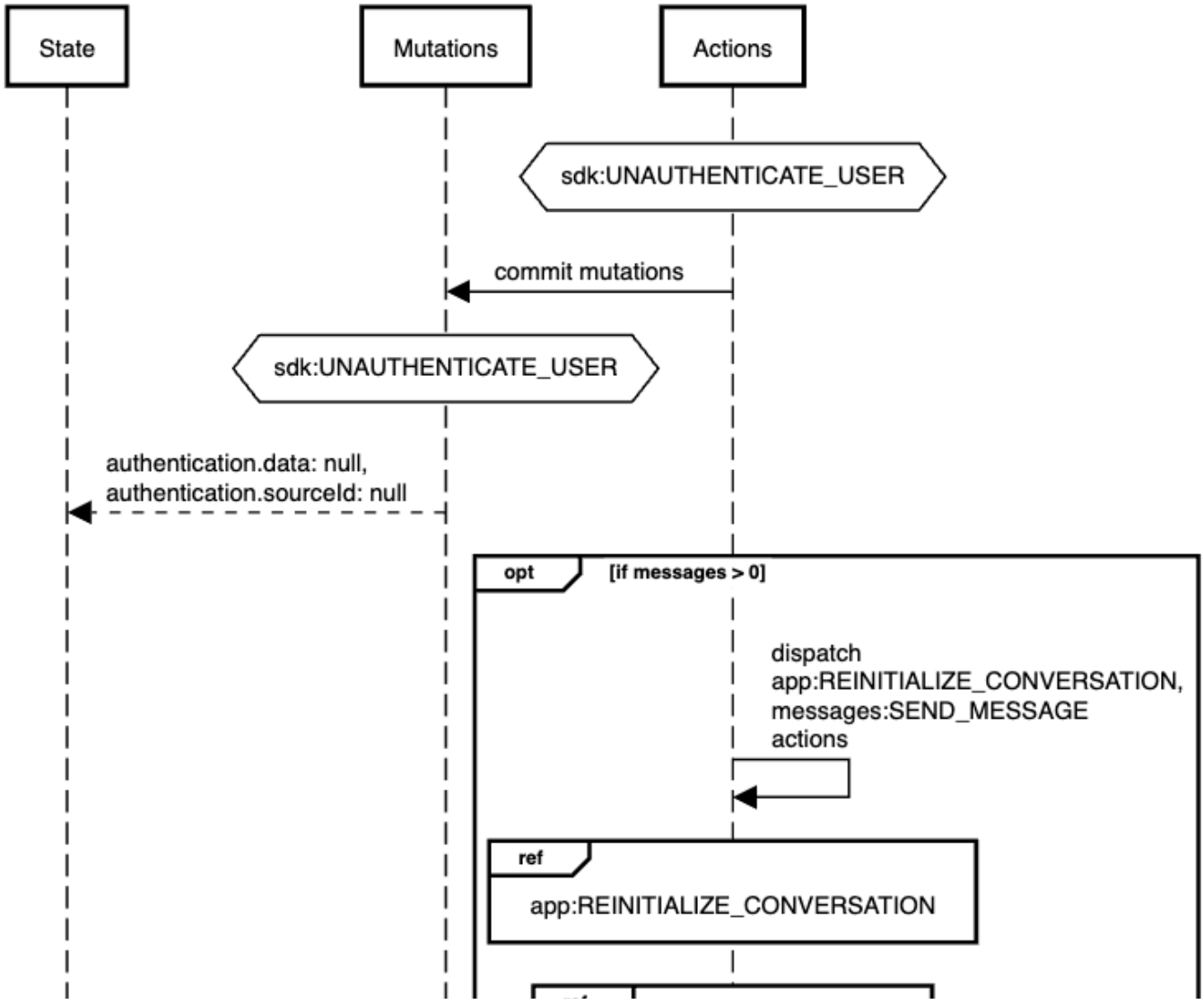
AUTHENTICATE_USER

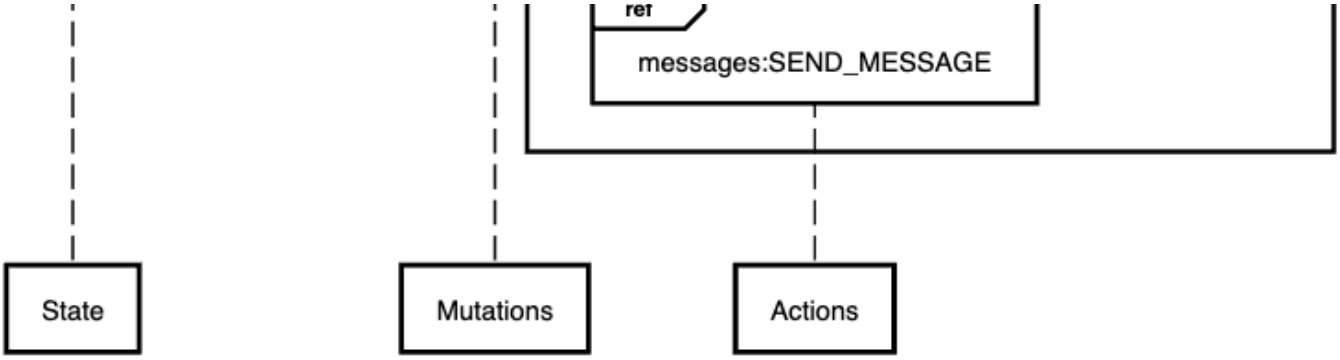




UNAUTHENTICATE_USER

EW UNAUTHENTICATE_USER Mutation Flow





Survey State

Overview

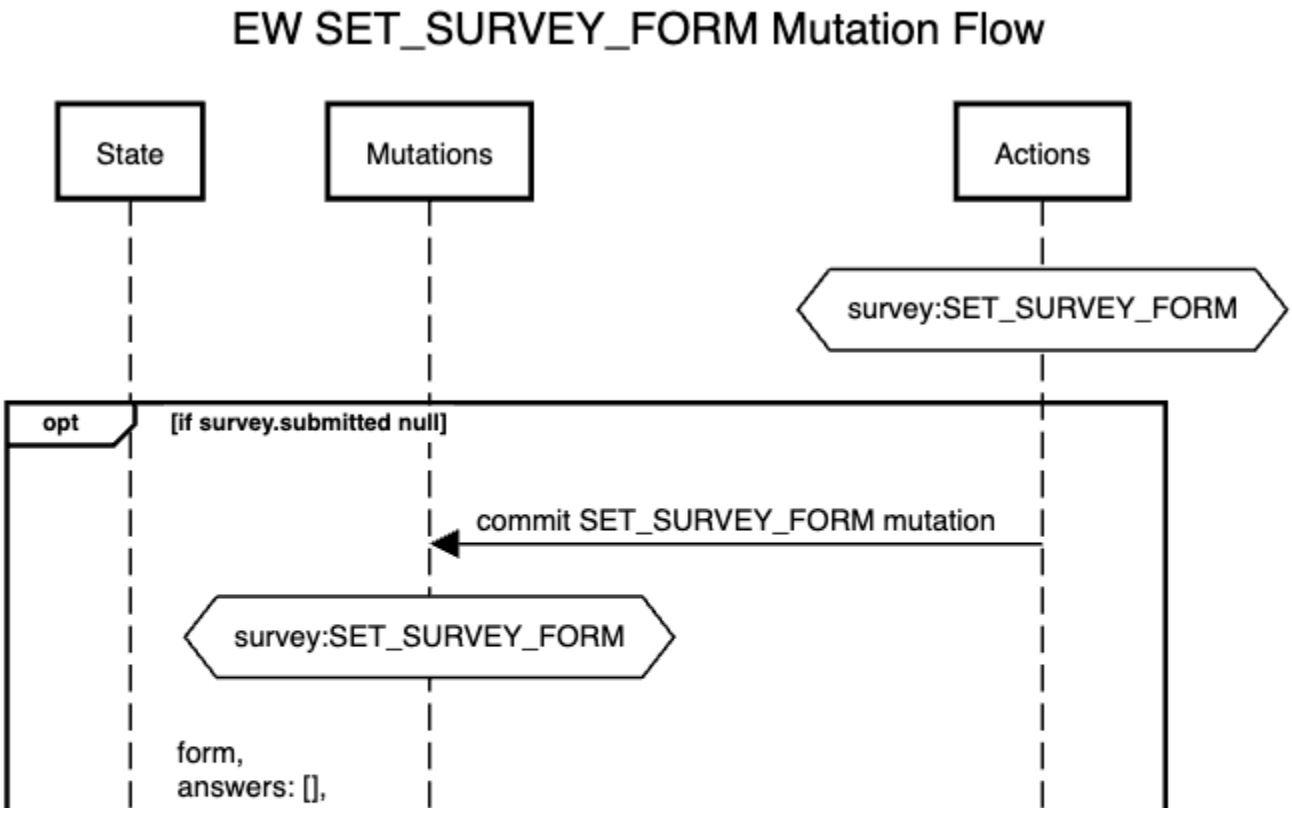
The Survey state houses information for surveys after a human handoff session ends.

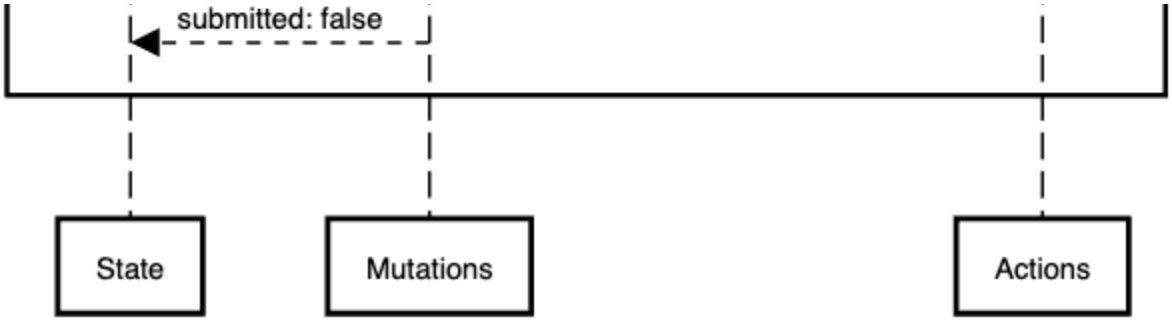
Initial State

```
{
  form: null,
  answers: null,
  submitted: false,
}
```

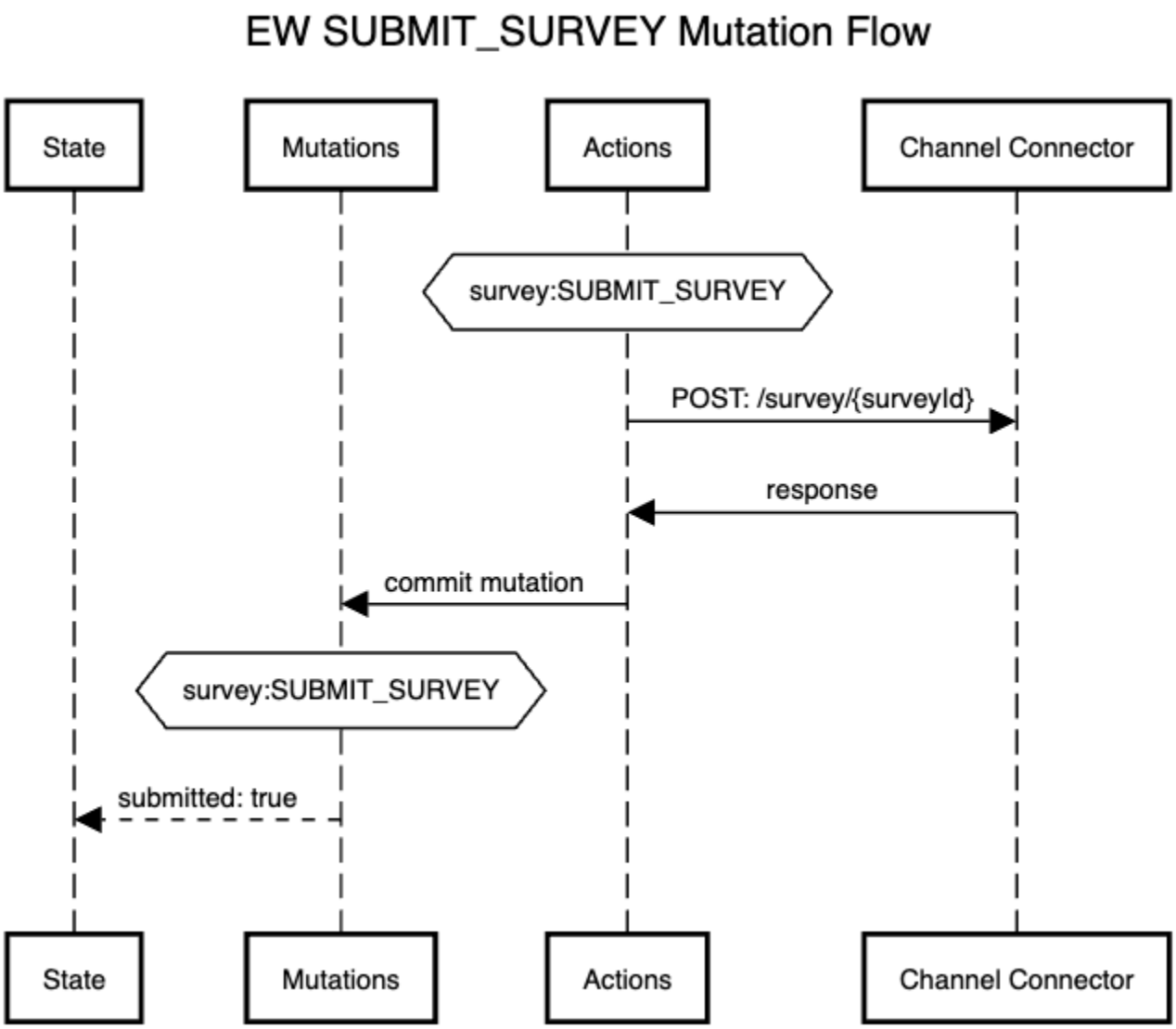
Actions

SET_SURVEY_FORM





SUBMIT_SURVEY



Embedded Mobile

Embedded Mobile is an SDK for iOS and Android that provides fully-functional UI to enable end-users to interact with a.i. assistants and human assistant's from within a client's native iOS or Android application.

- Assistant Features
  - A.I. Assistant communication
  - Quick Replies: 1-tap preloaded responses
  - Human Handoff: speak with a human agent
  - Customization via CUI settings
    - Current Options

Assistant Features

A.I. Assistant communication

The SDK provides a Button to be placed within the client app. When the button is on-screen, it will trigger necessary configuration checks before becoming visible, and clicking this button will launch the fullscreen assistant.

The assistant contains all necessary UI elements to facilitate the chat conversation with a.i. or a human. The assistant can be closed and re-opened with the conversation state persisting.

Quick Replies: 1-tap preloaded responses

Some agent messages contain preset user responses that a user can tap to send that response rather than typing out their own response. These quick replies can be configured via Conversate UI (CUI).

Human Handoff: speak with a human agent

When an agent has this feature enabled, end-users can choose to speak with a human instead of the a.i. agent. The backend services handle this transition, and then the UI is updated to indicate to the user that they are now speaking with another human. We currently support Human Handoff with Salesforce and our own Contact Center.

Customization via CUI settings

CUI provides some customization for the widget, and we may add more in the future.

Current Options

- Assistant name
- Tagline
- Brand color
- Logo URL

Potential future options

- Launch button screen position

Co-Browse: screen sharing with human agent

Contact Center only

Utilizes Twilio Video SDK to facilitate screen sharing of the user's device with the contact center human agent. Screen sharing is expected to stop when the user leaves the app.

Once screen sharing is active, the agent can then draw over the screen feed and generate drawing events. These events are sent over socket and received by the SDK which then recreates these drawings on screen.

Video/audio call with human agent

Contact Center only

Utilizes Twilio Video SDK to facilitate video/audio calls with the human agent.

SDK Features

As an SDK, we provide more functionality to developers than the UI. However, we keep it limited in order lessen or eliminate the need for code changes with our client’s apps after the initial implementation.

User Authentication

Beyond the UI, the SDK's also provide hooks for user authentication credentials to be provided. These credentials can also be cleared out whenever it may be necessary.

Events

We currently support customer apps launching the Assistant UI themselves, and we provide a few signals via Events to help them do so.

- Unread Messages: Provides to count of unread messages.
- Cannot Launch: Indicates that there was an error when the app tried to launch the Assistant.
- Assistant Ready: Indicates that runtime checks and configuration have been completed successfully.

iOS SDK

Tech Stack

- SwiftUI using Combine

Source Code

1. UI SDK - https://github.com/abeai/abe-ai-sdk-ios-ui
2. API SDK - https://github.com/abeai/abe-ai-sdk-ios-core
3. Tutorial - https://github.com/abeai/abe-ai-sdk-ios-tutorial
4. Certificate Manager - https://github.com/abeai/ios-match-certificates

Architecture

Our architecture follows the design patterns and philosophies of MVVM. We follow the guidelines set forth by Stanford Professor Paul Hegarty who formerly worked with Steve Jobs at NeXT. You can follow his university course online on YouTube; https://www.youtube.com/watch?v=jbtqIBpUG7g .

Technical Overview

Views

The UI consists of the following views:

1. [internal] AbeAISwiftUIConfigSwitcherViewController
2. [internal] AbeAIKeyboardManager
3. [internal] AbeAISiriWaveView
4. [internal] AbeAISwiftUIDockView
5. [internal] AbeAISwiftUIKeyboardView
6. [internal]AbeAISwiftUIMessageView
7. [internal] AbeAISwiftUIMicView
8. [internal] AbeAISwiftUITableView
9. [external] AbeAISwiftUIContentView
10. [external] AbeAISwiftUILaunchButton

AbeAISwiftUIConfigSwitcherViewController

Type: View Controller

This is a special View Controller to handle tweaking UI config settings without relaunching the app. This is solely used for internal demos.

AbeAIKeyboardManager

Type: Observable

This class is a SwiftUI observable wrapper with the sole purpose of watching and reacting to keyboard events. In certain views, if you have a need to react to keyboard changes, you can leverage this observable to handle these state changes.

AbeAISiriWaveView

Type: SwiftUI View

This SwiftUI View is used within the embedded microphone experience and handles all animations. This view is deprecated.

AbeAISwiftUIDockView

Type: SwiftUI View

AbeAISwiftUIMicView

Type: SwiftUI View
This SwiftUI View is used within the embedded microphone experience and handles all microphone interactions. This view is deprecated.

AbeAISwiftUITableView

Type: SwiftUI View

This internal SwiftUI View contains all chat tables behavior.

AbeAISwiftUIContentView

Type: SwiftUI View

This is a container for the main SwiftUI View. This view is central to all navigation inside the chat experience.

AbeAISwiftUILaunchButton

Type: SwiftUI View
This is an extremely import Abe.ai SwiftUI View. For most consumers, this will be a self contained entry point to the application. This button has the ability to retain state and

End User Documentation

The iOS SDK enables developers to add in the component to any iOS app running iOS 10+. Upon dropping the component in and initiating the class with the configuration data, the UX can be launched to engage with the virtual assistant.

Prerequisites

- If you haven't created an agent yet, please follow the guides on creating your first agent. After creating your first agent, you have multiple options to install the AbeAI component in your project.
- You will need the following information from your Account Manager.

Variable	Description
Channel Connector URL	This is the URL for your mobile channel. Do not include a trailing "/".
Verify Token (HMAC Key)	This key is required for securing communication.

Step 1: Install AbeAI

CocoaPods	Swift Package Manager	Carthage	Zip file
Request access to the private spec repo.	Request access to the private repo.	Request access to the private repo.	Request access and we will share the file securely.

Step 2: Initialize the "User Object"

Create an `AbeAIUser object`.

- The User ID will be referred to as `${userId}`. This is the unique external identifier for a user from your institution. Refer to https://docs.platform.abe.ai/api#CreateMessage .

```
AbeAIUser(sourceType: String,
          userId: String,
          authenticationData: String,
          authenticated: Bool
          firstName: String,
          lastName: String)
```

Example:

```
let abeAIUser: AbeAIUser = AbeAIUser(sourceType: "SOURCE",
                                     userId: "ID",
                                     authenticationData: "{\\"token\\":
\\"USER_TOKEN\\"}",
                                     authenticated: true,
                                     firstName: "Abe",
                                     lastName: "Lincoln")
```

Step 3: Initialize the "Service Connector"

Create an `AbeAIServiceConnector object`. This object's purpose is to configure the server side portion of Abe's Embedded Mobile.

```
AbeAIServiceConnector(channelConnectorUrl: String,
                      verifyToken: String)
```

Example:

```
let abeAIServiceConnector = AbeAIServiceConnector(channelConnectorUrl:
"<https://www.abe.ai/ID",>
                                                  verifyToken: "Very
Secrect Token")
```

Step 4: Add the chat button to your view controller.

Now it is time to add the chat button to your view controller.

```
 var abeAIButton: AbeAIButton?
```

```
func addAbeAIChatButton() {
    let abeAIUser: AbeAIUser = AbeAIUser(sourceType: "SOURCE",
                                         userId: "ID",
                                         authenticationData: "{\\"
token\\":\\"USER_TOKEN\\"}",
                                         authenticated: true,
                                         firstName: "Abe",
                                         lastName: "Lincoln")

    let abeAIServiceConnector = AbeAIServiceConnector
(channelConnectorUrl: "<https://www.abe.ai/ID",>
                                                    verifyToken:
"Very Secrect Token")

    abeAIButton = AbeAIButton(frame: CGRect(x: 0, y: 600, width: 100,
height: 100),
                              viewController: self,
                              serviceConnector: abeAIServiceConnector,
                              user: abeAIUser,
                              buttonImage: UIImage(named:
"message_48pt"),
                              tintColor: ConfigManager.
AGENT_NAME_FONT_COLOR)

    self.view.addSubview(messageButton!)
}CI/CD
```

UNDER CONSTRUCTION

Android SDK

Tech Stack

- Kotlin with XML layouts
- ViewModel and LiveData
- OkHttp and Retrofit for network operations & image loading
- Gson for JSON de/serialization
- Handler for posting to main thread
- EncryptedSharedPreferences for local storage
- Twilio Video SDK for audio/video calls and Co-Browse (screen sharing)

Repositories

- Android SDK: https://github.com/abeai/abe-ai-sdk-android/
- Android Tutorial: https://github.com/abeai/abe-ai-sdk-android-tutorial

Architecture

In general, the code follows Android's architecture guidance utilizing a UI and Data layer while omitting the optional Domain layer. The UI layer contains Views, Activities, Fragments, and ViewModels. The Data layer contains Repositories, REST API's, Web Sockets, and EncryptedSharedPreferences. There are a few classes named *Manager that act as the central computation for specific features. Technically these would be a part of the Data layer, and thus should be Repositories not Managers, but this distinguishes them for the logic they perform and state they hold.

Thus far, app dependencies have been kept minimal in order to limit the number of transitive dependencies of the SDK due to our deployment process (*more in CI/CD*). Because of this, certain common Android libraries have been excluded in favor of less favorable or custom-built options. Specifically, Coroutines are not used, and the SDK uses its own dependency injection code modeled after Koin.

UI

The UI consists of 2 Views, 1 Activity with 2 Fragments, and a 2nd Activity with no Fragments.

- ***[public]*** AbeAILaunchButton
- ***[public]*** AbeAIAssistantActivity
  - AbeAIMessengerFragment
  - AbeAiIncomingCallFragment
- AbeAICallActivity
- ***[public]*** AbeAICobrowseOverlay
  - AbeAIAgentDrawingView
  - AbeAICobrowseButton: FloatingActionButton

AbeAILaunchButton

Loads the logo url from CUI config into a FloatingActionButton and displays a red bubble with the count of unread messages when the count is greater than 0. The View itself is sized to contain just these 2 elements and the shadow, but this could be changed to match-parent for both width and height so that the button can be repositioned via CUI config.

This View waits for CUI config to be loaded before becoming visible.

When the FloatingActionButton is clicked, it launches AbeAIAssistantActivity.

AbeAIAssistantActivity

This Activity contains the UI for messaging an agent and processing an incoming video/audio call. These 2 primary functions are each contained in a Fragment. This Activity is public so that consumers of the SDK may launch it as well without the AbeAILaunchButton.

AbeAIMessengerFragment

This Fragment contains the UI for messaging an agent. This is split into 3 areas; title, messages, and edit. The Title area contains the brand logo, assistant name, a tagline, and a minimize button. The Messages area is just that, messages of each party in the conversation and a typing indicator. The Edit area is where users can type and send their message. If there is a speech recognition service available on the system, then there will be a mic button in the Edit area in-place of the send button until text is entered.

AbeAiIncomingCallFragment

This Fragment manages a request from a human agent to initiate a video/audio call. It handles permission gathering (mic, camera, bluetooth) and presents errors when permissions are not granted. Mic & Camera permissions are required while Bluetooth is optional because it is only used for audio device switching. There is also a preview of the user's camera feed and buttons to toggle on/off the mic & camera. When all permissions are granted, the user can either decline or accept the call. Declining the call will return the user to the messages Fragment. Accepting the call will launch AbeAICallActivity. Either response is reported back to Channel Connector via the web socket connection.

AbeAICallActivity

This Activity manages an active call. This includes user and agent video feeds, mic & camera toggle buttons, a button to choose listening audio device, a minimize button, a hangup button, and can enter PictureInPicture mode displaying just the agent's video. This Activity will be removed with the coming redesigns because the agent video will split the screen with the messenger, thus making only one Activity necessary.

AbeAICobrowseOverlay

This is a View that contains the UI for displaying the agent's drawings during Co-Browse as well as controls for exiting Co-Browse, and it must be placed within the consuming app's layout. The View sizes itself to take up all possible area on the screen. System bars and ActionBar's placed by the app theme are excluded because they cannot be drawn over.

AbeAIAgentDrawingView

This View fills the entire overlay and is completely transparent except for any agent drawings that are rendered over everything.

AbeAICobrowseButton: FloatingActionButton

This is currently a small FloatingActionButton that anchors to the side of the screen. It can be dragged around and will snap to the nearest side when released. When tapped, it will end the Co-Browse session. There is room to expand the functionality by making this more of a toolbar than a single button.

Initialization flow

The SDK needs to be initialized by the consuming app via the `AbeAI` object.

```
fun initialize(
  application: Application,
  channelURL: String,
  channelID: String,
  verifyToken: String
)
```

application	Registers ActivityLifecycleCallbacks for fine-tuning screen-sharing, camera usage, and Picture-in-picture
channelURL	Base URL for channel API and web socket communication
channelID	ID of the agent
verifyToken	Token to verify that API access is permitted

When the SDK is initialized, it builds the dependency graph, registers ActivityLifecycleCallbacks with Application, and initiates the `AbeAIConfigRepository` which will then fetch the CUI config. Once the config has returned successfully, an `AssistantReadyEvent` is published to indicate to the app that it is now safe to launch an `AbeAIAssistantActivity`. If an `AbeAILaunchButton` is on the screen, it will now load the logo and become visible.

Conversation flow

Currently, a conversation is not initialized until the assistant is launched, however changing this would be trivial.

The conversation is started by POSTing to **/{channelID}/custom/conversations/** and then POSTing to **/{channelID}/custom/conversations/{conversationId}/messages** with the intent `CONVERSATION_INITIALIZE` to receive the assistant welcome message.

Building

Debug

For debugging and testing, there is a demo app within the project with both Kotlin and Java Activities. Simply run the project from Android Studio with the desired Activity selected to launch.

Release

From command line run `./gradlew clean assembleRelease`. Which generates `./abe-ai-sdk-android/build/outputs/aar/abe-ai-sdk-android-release.aar`

Demo app

A demo app is required to test the SDK, and there are several options that can be configured here as well. All settings are stored in a SharedPreferences so that they persist between app launches. However, the "demo" environment, the environment set via `AbeAI.initialize` is always set regardless of the latest environment used.

Public Views

The demo app uses both the `AbeAILaunchButton` and the `AbeAICobrowseOverlay` to demonstrated and test their usage. When the environment changes, the `AbeAILaunchButton` is replaced with a new one.

Custom Launch

There is a custom launch button and TextView to show unread message count to demonstrate and test scenarios that do not use `AbeAILaunchButton`. Since we offer 2 methods of receiving Events, the demo app lets you switch between the 2 to verify they both work.

Environment Switcher

The Environment Switcher receives environment configuration via gradle, and then loads its account and agents for selection via Spinners. Only environments with an EMBEDDED_MOBILE channel are enabled.

Auth View

The AuthView lets you enter auth credentials for the agent to use as well as un-authenticate. In order to test a few scenarios, there are also options to change the user's name or un-authenticate after a specified time.

Testing

Unit tests are written with JUnit 5 & MockK. We have configured JaCoCo Java Code Coverage Library which can be run with `./gradlew getCoverage`.

CI/CD

UNDER CONSTRUCTION

TODO

Thus far has been described the current state of the SDK, but changes are coming soon. There have been designers working on a completely new look for the UI which will also come with some architecture improvement opportunities.

- As the look is being modernized, fewer Views will be influenced by the brand color as it is now. It is still unclear if any of the colors will be driven by CUI config or if they will all be static.
- Currently, the `AbeAILaunchButton` is just the button and thus must be placed on screen manually by developers. Though it is not currently supported in CUI, we may change this View to be fullscreen and allow CUI config to drive the position on screen. This will allow clients that utilize contract developers to reposition the button whenever they please without getting developers involved.
- The original UI for mobile and Axos' UI can be removed as part of the redesign.
- We will be implementing a CI/CD pipeline which will include hosting the SDK artifact in a Maven repository. This will eliminate the concern of adding too many transitive dependencies since they will be managed by Gradle.
- Without dependency constraints, we really should be using Coroutines and Flows.
- Consider using Koin or Hilt for dependency injection
- Consider migrating to DataStore from SharedPreferences. (Security)
- UI Layer -> "Do not send events from the ViewModel to the UI."
  - Explore removing permission check logic from ViewModels
- Modularization to better separate the layers. This could also set us up to release separate artifacts for the core SDK functionality and the UI so that customers can opt to build their own UI and exclude our UI classes.
- Use Jetpack Compose
- Explore how a Domain layer may be implemented, especially where "Views" make use of multiple ViewModels. Think of the Domain Layer as a ***state accessor*** not a ***state holder***.

SDK Usage Docs

Installation

Prerequisites

- If you haven't created an agent yet, please do so first.
- You will need the following information from your Account Manager.

Variable	Description
Channel Connector URL	This is the URL for your mobile channel.
Channel ID	This is the ID of your mobile channel.
Verify Token (HMAC Key)	This key is required for securing communication.

Step 1: Install AbeAI

Option 1: Gradle

UNDER CONSTRUCTION

build.gradle

```
'TODO: Include steps to reach private maven repo.'
```

app/build.gradle

```
dependencies {
    implementation 'ai.abe.sdk.android:abe-ai-sdk-android:$abe_version'
}
```

Option 2: Zip file

1. Request access and we will share the file securely.
2. Download AbeAI component for Android and unzip the file.
3. Copy `abe-ai-sdk.aar` into your project.
4. Add `abe-ai-sdk.aar` as a dependency in your app gradle file.

```
dependencies {
    implementation files('[PATH]/abe-ai-sdk.aar')
}
```

Note: Using the archive file directly will require manual imports of transitive dependencies.

Library	Owner	Version	Usage
Material Components	Google	1.5.0	A FloatingActionButton as the base of the AbeAILaunchButton
Retrofit Gson Converter	Squareup	2.9.0	Uses Gson to parse API responses and includes OkHttp, Retrofit, and Gson

Gradle Imports

```
dependencies {
    implementation 'com.google.android.material:material:1.5.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
}
```

Step 2: Initialize AbeAI

Initialize AbeAI:

```
import ai.abe.sdk.android.abe.AbeAI

class MyApplication : Application() {

    override fun onCreate() {
        super.onCreate()

        AbeAI.initialize(
            this,
            channelURL,
            channelID,
            verifyToken,
        )
    }
}
```

Step 3: Add Widget to layout

Option 1 (Recommended)

Add the self-contained button to your view.

```
<ai.abe.sdk.android.view.AbeAILaunchButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

Option 2

You manage when the Abe AI Assistant is displayed.

```
startActivity(Intent(this, AbeAIAssistantActivity::class.java))
```

Reference

The Mobile SDK exposes multiple methods to interact with the component programmatically.

Authenticate User - authenticate

```
fun AbeAI.authenticate(userId: String, firstName: String?, authToken:
String): Unit
```

Notify the SDK that the user is authenticated by providing the authentication `userID`, `firstName` (optional), and `authToken`.

`authToken` the token required for calls to your backend to succeed.

If any data changes, calling `authenticate` is required. The object is not watched for modifications.

Unauthenticate User - unauthenticate

```
fun AbeAI.unauthenticate(): Unit
```

Notify the SDK that the user that is currently authenticated is no longer authenticated. This can be called even if `authenticate` has never been called to be utilized as a sanity check if desired.

Events

Events can be implemented in 2 ways, either a platform-specific observable or an event bus. Each event is either persistent or not, meaning new subscribers will or will not be notified of the latest event. The supported events are described below:

Event	Content	Persistent	Description
UnreadMessagesEvent	**count**: Integer	Yes	This event is typically fired when the messenger is closed to indicated that new messages have been received. It will also fire with a count of ***0*** when the messenger is opened.
CannotLaunchEvent	**error**: AbeAIException	No	This event can be fired at 2 points, either when the api call to retrieve agent config fails, or when the application attempts to launch the messenger after the config api call fails. ***AbeAIException*** is an ***Exception*** on Android and a ***???*** on iOS, so details of the error can be gathered like usual.
AssistantReadyEvent	*n/a*	Yes	This event is fired once, after initialization has completed. This indicates that all initialization steps have completed, and thus the app can proceed to launch the Abe AI Assistant.

There are 2 ways to observe these events, either by using the provided event observables or by registering an `AbeAIEventSubscriber`.

Observables

We have provided platform-specific observables via the `AbeAI` singleton:

```
// val unreadMessageCount: LiveData<Int>
AbeAI.Events.unreadMessageCount.observe(lifecycleOwner) { count -> }

// val error: LiveData<AbeAIException?>
AbeAI.Events.error.observe(lifecycleOwner) { error -> }
```

Register an `AbeAIEventSubscriber`

Implement the `AbeAIEventSubscriber` interface to process events.

```
interface AbeAIEventSubscriber {
    fun onEvent(event: AbeAIEvent)
}
```

Declare an `AbeAIEventSubscriber` to process events.

```
import ai.abe.sdk.android.events.AbeAIEventSubscriber

private val eventSubscriber = AbeAIEventSubscriber { event ->
    when (event) {
        is AbeAIEvent.UnreadMessagesEvent -> {
            val count = event.count
        }
        is AbeAIEvent.CannotLaunchEvent -> {
            val error = event.error
            val message = error.message
        }
    }
}
```

Register the subscriber with the SDK:

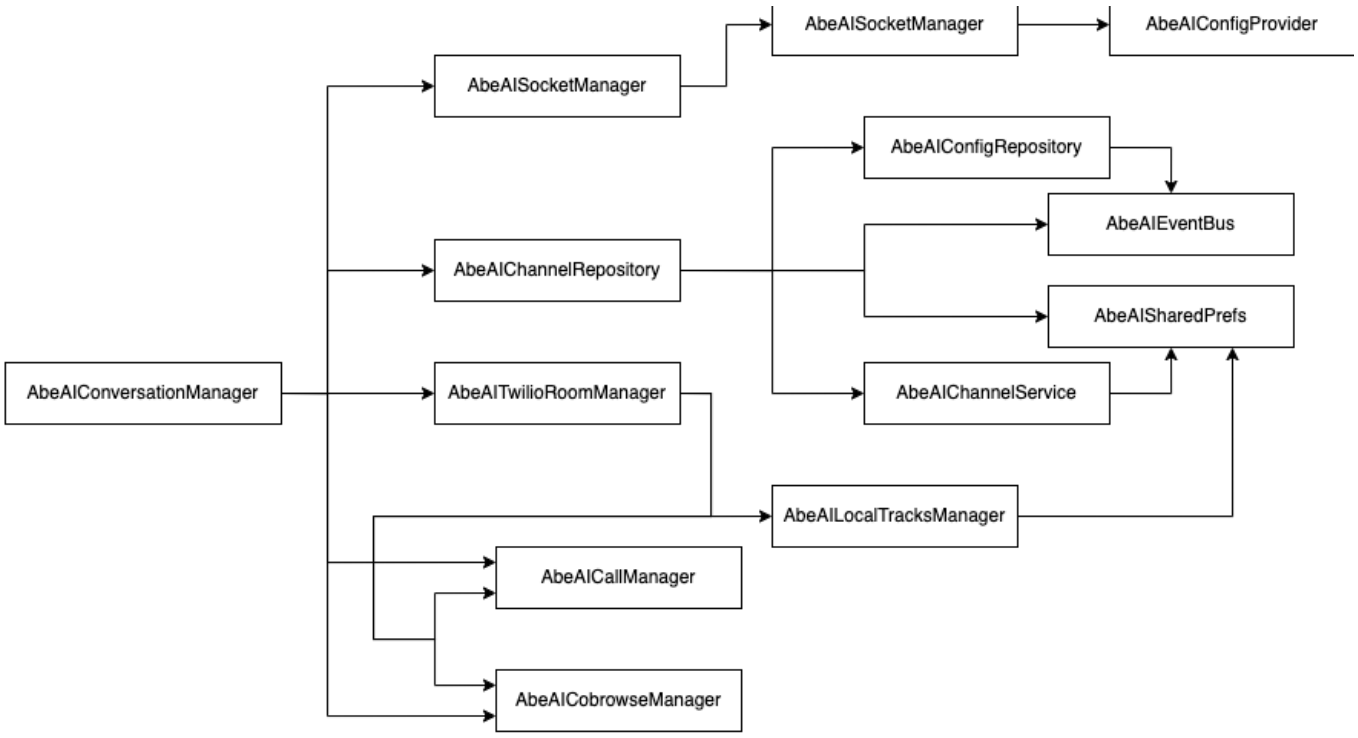
```
if (!AbeAI.hasSubscriber(eventSubscriber)) {
    AbeAI.registerSubscriber(eventSubscriber)
}
```

Be sure to unregister the subscriber when it is no longer needed:

```
AbeAI.unregisterSubscriber(eventSubscriber)
```

Current Class Summary

This page exists to list at a very high level the function of certain Class in the Android codebase and how they interact with other classes.



VideoViewModel

- collects state from:
  - ConversationManager
    - TwilioRoomManager
      - LocalTracksManager
  - CallManager
  - CobrowseManager
  - SharedPrefs
- track rendered video sinks
- check camera & mic status
- request permissions
  - toggle answer button enabled
  - set permissions denied banner
- audio switch
  - start manager
  - get devices
  - show selection dialog (UIEvent)
  - select audio device
- RoomManager
  - collect state
  - toggle devices
  - check hardware availability (LocalTracksManager)
- check CallManager::isAudioCall
- show/hide errors

ConversationManager

- collects state from:
  - TwilioRoomManager
  - CallManager
  - CobrowseManager
  - ChannelRepository
  - and maintains agentDrawingEvents
- listens to and controls sockets
- ChannelRepository
  - signal new events

    - signal agent typing
    - startNewConversation
    - sendConversationIntent: Intent pass-through
    - sendConversationMessage: Message pass-through
    - onError: Trigger hard-coded error message
    - open (dump unread queue):
    - minimize:
        - Remove typing indicator
        - Post unread count
    - cleanup
        - cancel network calls
        - clear unread queue
        - clear message cache
- ConfigProvider
    - useDefaultTheme: will be removed
- SharedPrefs
    - conversationId CRUD
    - signal session restore done
- TwilioRoomManager
    - requestVideoCall (Socket/SessionRestore)
    - endCoBrowse (Socket/User)
    - requestCoBrowse (Socket)
    - clearRoom (end conversation)
    - onResume (resume conversation)
    - joinVideoCall (User event)
    - endVideoCall (Socket/User)
    - joinCoBrowse (User event)
    - onEventHandled (UIEvent pass-through)

CallManager

- stores call state in SharedPrefs
- PermissionUtil.hasRequiredPermissions
    - (isAudioCall || hasCameraPermission) && hasMicPermission
    - also used in ConversationManager

CobrowseManager

- stores cobrowse state in SharedPrefs

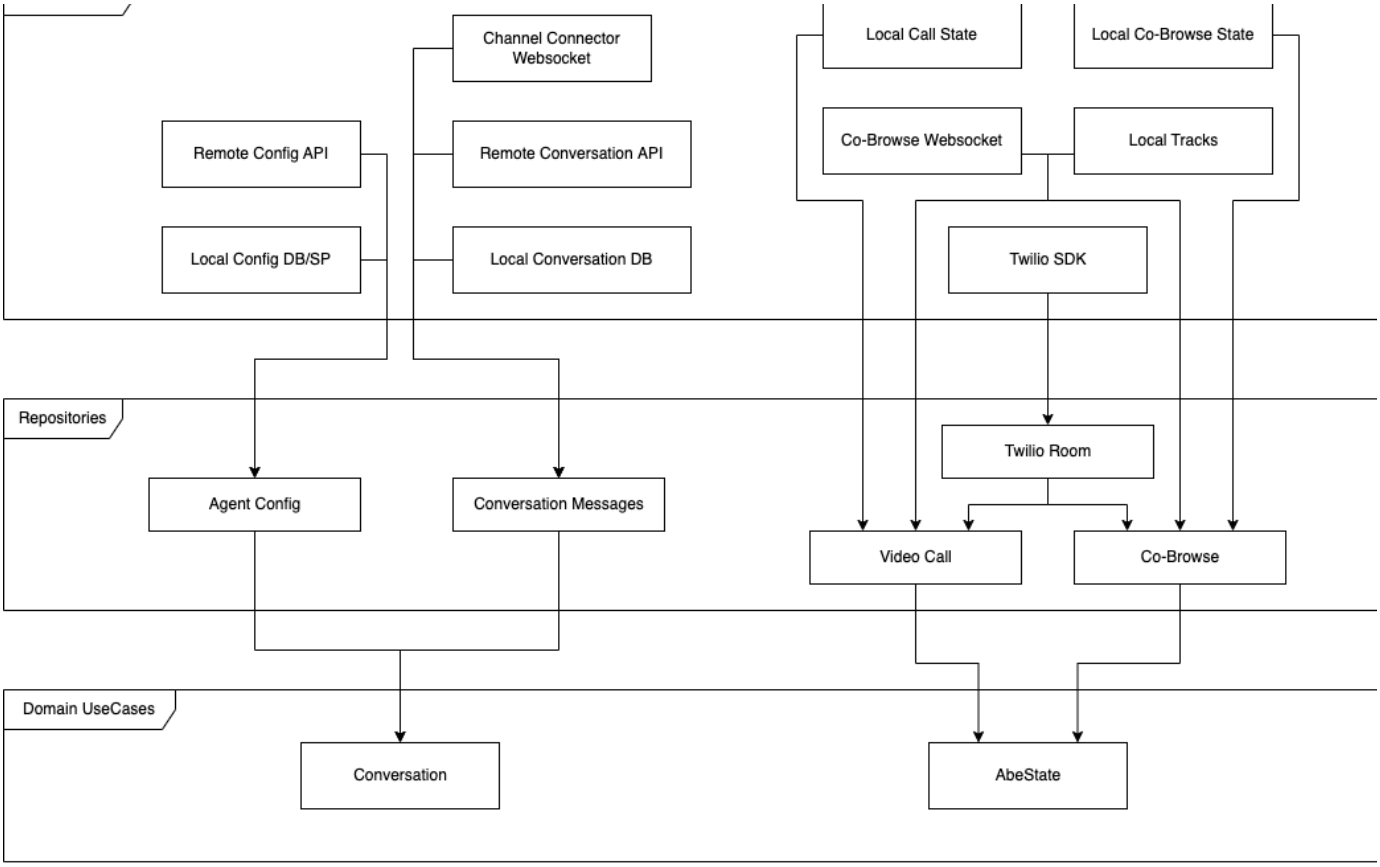
ChannelRepository

- holds conversation messages state
    - message list
    - current agent
    - assistant name from CUI config
- SharedPrefs
    - check if session is active
    - readCount CRUD
- ChannelService
    - POST start conversation, intent, and utterance
    - GET conversation events
- ConfigRepository
    - get assistant name
- EventBus
    - post unread message count

Future Architecture

Alongside the UI redesign, we have a great opportunity to refine the architecture of the SDK to better align with Android's architecture guide and to modularize the code. These steps will improve code readability, testability, and position us to provide 2 separate SDK's; a `ui` version with all functionality and UI for customers wishing to utilize our UI and a `core` version with no UI for customers wishing to build their own UI.

Following the guide, there will be a UI, Domain, and Data Layer where the Domain layer contains various UseCases and the Data layer is split into Repositories and Data sources. Each class is to expose immutable data via a Kotlin Flow, receive events indicating user or application actions, and if it maintains state, it will handle events immediately and update its state. Here is a high level graph of the app state layers (Domain & Data):



Features

Chat

The Chat feature encapsulates the text based messages sent back and forth between a user and an agent. This includes configuration from CUI, a web socket to signal new events, sending/receiving messages via a REST api, and the UI to facilitate all this.

Remote Config API

Calls Channel Connector's REST api to get configuration for the given agent.

Local Config Store

**** Optional **** This could be used to store config locally to limit the need for the network call, but likely is not necessary.

Channel Connector Web Socket

We connect to the web socket with the channel ID and the conversation ID. This socket sends us events to indicate status of an agent typing and when there are new events on the conversation. Then we fetch the conversation events ourselves. Here are some sample socket events:

Events:

- ***NEW_EVENTS***: There are new events in the conversation needing to be fetched
- ***AGENT_TYPING_UPDATE***: There has been a change of status of an AI or human agent typing.

Remote Conversation API

This is the REST api for conversation functions:

- begin a conversation
- send intents & utterances
- fetch conversation events, utilizing a cursor string to fetch only new events

Local Conversation Store

This local data store will contain the list of conversation messages. It will be the single source of truth and simplify the conversation recovery process. Using a Room database may be a good option, but keeping this in memory is OK too.

Conversation Repository

This repository will be responsible for maintaining the state of the conversation. Once initialized, it will maintain the socket connection and fetch events has needed. It will be scoped to the application context so that it may operate without any views, observers, or Flow collectors. It will maintain the state with private methods that will be called under certain conditions based upon external actions i.e. message-read or new events. As new events arrive from the network, the repository will manage local storage via memory or the Local Conversation Store.

```
State (
  val agent: Agent(
    val type: AgentType,        // Enum[AI || HUMAN]
    val name: String?,          // Eiher provided by api or null
  ),
  val messages: List<Message>,
)
```

Conversation Use Case

This Use Case will be responsible for combining conversation data with CUI config data, specifically the assistant's name when it is missing from the conversation data. Using a UseCase will be a cleaner pattern than having both the Conversation Repository and Remote Conversation API aware of the Config Repository.

Contact Center

The Contact Center as a feature encapsulates both video/audio calls with a human agent and Co-Browse. Both of these can operate independently of each other.

Local Call State

This will maintain the state of a video call on disk so that it may be recovered in the case of an app crash and quick return. Since permissions are involved, this state will be kept simple thus leaving other aspects, such as permissions denied or hardware failures, to the UI layer.

Local Co-Browse State

This will maintain the state of a Co-Browse session on disk so that it may be recovered in the case of an app crash and quick return. Since permissions are involved, this state will be kept simple thus leaving other aspects, such as permissions denied, to the UI layer.

Co-Browse Socket

This web socket will be used to facilitate video calls and co-browse sessions. It will only be opened when a human handoff session using our Contact Center is initiated and closed upon ending of this session.

Events:

- ***ringing***: The agent is requesting a video call
- ***endVideoCall***: The agent has ended the video call
- ***ready***: The agent is requesting a Co-Browse session
- ***agentDrawing***: Events indicating how the agent has drawn over the user's screen
- ***clearAgentDrawing***: The agent has cleared their drawing frame
- ***end***: The agent has ended the Co-Browse session

Local Tracks

This class will create and hold the local tracks need for the Twilio SDK to facilitate video calls and co-browsing. These tracks are one LocalAudioTrack using the devices microphone and 2 LocalVideoTracks using either the devices camera or screen capture.

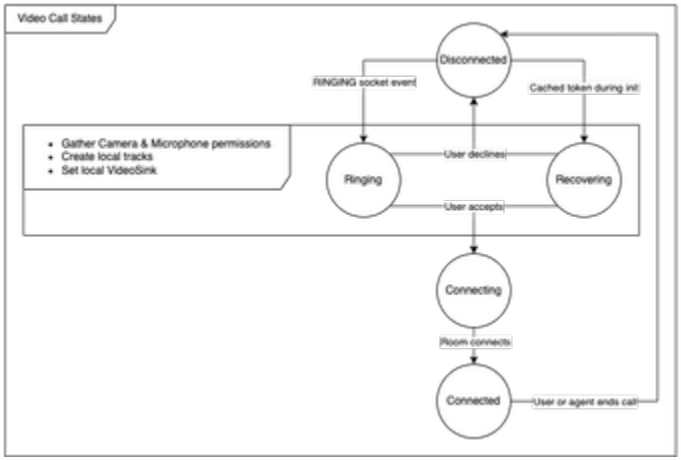
Twilio SDK

The Twilio SDK is used to facilitate video calls and co-browsing. It works by joining a `Room` using the connection token provided by Channel Connector, and then collecting the Contact Center agent's video as well as publishing the user's local tracks.

Twilio Room Repository

This repository will be responsible for communication with the Twilio SDK and maintaining the Room state. When actions need to be taken on the local tracks, like publishing and releasing, they will occur here. Since this repository will connect to the Twilio Room, it is within this repository's state that the agent's video track will be exposed.

Video Call Repository



This repository will coordinate video calls and maintain their state. It will listen to the web socket in order to react to changes on the Call Center side. It will request and hold the local mic and camera tracks, enabling and disabling as directed by the UI layer. It will pass the call token to the Twilio Room Repository to initiate the Twilio Room and watch the room's state to add the tracks when able.

Co-Browse Repository

This repository will coordinate co-browse sessions and maintain their state. It will listen to the web socket in order to react to changes on the Call Center side. It will request and hold the screen capture track, releasing and creating new tracks as directed by the UI layer. It will pass the call token to the Twilio Room Repository to initiate the Twilio Room and watch the room's state to add the track when able. It will also maintain the list of AgentDrawing events.

AbeState

This use case will be used to aggregate all portions of the state to be exposed as one entity to the UI layer.

Modularization for publishing 2 Archives

With proper modularization of the SDK, we will be able to publish 2 different Android Archive files (.aar) for different use cases. One will be the core functionality allowing customers to implement their own UI, and the other will contain both the core functionality and our UI.

Public State

```
State (
  val agent: Agent(
    val type: AgentType,                 // Enum[AI, HUMAN]
    val name: String,
  ),
  val messages: List<Message>,
  val unreadCount: Int,
  val callState: CallState,              // Enum[DISCONNECTED,
RINGING, RECOVERING, CONNECTING, CONNECTED]
  val selectedAudioDevice: AudioDevice,
  val coBrowseState: CoBrowseState,      // Enum[DISCONNECTED, READY,
RECOVERING, CONNECTING, CONNECTED]
  val agentDrawings: List<AgentDrawing>(
    ...
    AgentDrawing(
```

```
        val prevX: Float,
        val prevY: Float,
        val currX: Float,
        val currY: Float,
        val color: Int,
        val lineWidth: Int,
      )
    ),
    val errors: List<Exception>,
  )
```

Public Actions

- AbeAI
  - `initialize(application: Application, channelURL: String, channelID: String, verifyToken: String)`
  - `startNewConversation()`
  - `authenticate(id: String, authToken: String, firstName: String?)`
  - `unauthenticate()`
  - `onErrorHandled(error: Exception)`
  - `state: Flow<State>`
  - MessageHelper
    - `sendUtterance(utterance: String)`
    - `sendIntent(intent: String)`
  - VideoHelper
    - `declineCall()`
    - `enableMic(isEnabled: Boolean)`
    - `enableCamera(isEnabled: Boolean)`
    - `acceptCall()`
    - `scanAudioDevices()`
    - `selectAudioDevice(device: AudioDevice)`
    - `endCall()`
  - CoBrowseHelper
    - `declineCoBrowse()`
    - `acceptCoBrowse(screenCaptureIntent: Intent)`
    - `endCoBrowse()`

Services

The service breakdowns below should be used to supplement your reading of the various guides related to https://envestnet.atlassian.net/wiki/spaces/CAPD by providing deeper insights into each service's internal functionality and greater purpose in the architecture.

Table of Contents

/98595799919. Intent method executions, journeys, feature flags, and location configurations are defined here. Currently, the design supports multi-tenancy, multiple banking clients can and are supported by one instance of an integratio

User Manager

User Manager is a simple microservice that stores end user account data as well as end user client platform data. These records are required to identify the end user when a request is made from any client/device where that user has completed the account linking flow (@see Auth Web Client). Two main database records are managed by User Manager; Channel Identity records and User records .

NLU

Auth Web Client

Auth Web Client(AWC) acts as the main authentication gateway for end users when linking their Financial Institution account to their physical device and the Conversational AI ecosystem. It does this by providing front end views for authentication as well as redirecting the end user to third parties for authorization when necessary. This microservice also provides a RESTful HTTP API to administrate Agent authentication settings for example changing authentication/authorization schemas, or managing access keys for existing configurations.

Auth Web Client authentication schemas include username/password (local auth), OAuth2, as well as code or question driven MFA flows. It acts as a bridge from the Conversational AI ecosystem to the FI ecosystem by redirecting end users to FI or third party authentication providers, or by directly ingesting user credentials and negotiating with those providers on the user's behalf. This process is known as "Account Linking" and is a critical component for AI Agents.

End users interact with AI Agents via third party controlled channels (Amazon Alexa, Google Home, etc), and AI Agents must be able to gather banking/PII data to fulfill end user requests via these platforms devices/clients. The account linking flow allows the AI Agent to gather the credentials (or equivalent) and to pass them on to the FI for authorization, or to redirect the end user to an FI controlled authentication platform and to subsequently receive authorization credentials from that flow.

Once the authentication process is completed by the authentication provider will supply the Conversational AI ecosystem with authorization credentials (Bearer token, cookie, etc) which are then securely stored and used for gathering banking/PII data during conversation flows. Auth Web Client also supports "pre authentication" which is a truncated flow that is utilized by clients (website/mobile app) where the user is already authenticated.

Given that many end users will interact with the Conversational AI ecosystem via an embedded chat widget on an FI's website, or via an FI's mobile app, it is expected that end users are already authenticated by the site or app. It then becomes unnecessary to have the end user perform account linking simply on behalf of the AI Agent. During this pre authentication flow the FI will pass authorization credentials directly to the AI Agent which are then validated and securely stored.

Account Linking Schemas

Account Linking Schemas are the ways users can authenticate with a real world Financial Institution or FI through the Abe platform. An Account Linking Schema will leverage a backend built specifically for an FI which will interface with their API.
There are two types of Authentication Schemas for Account Linking.

Server Side Login Form

> A Server Side Login Form is essentially a Username and Password login similar to how one would log into Online Banking. If the FI requires MFA, that is supported and integrated as well. This Schema provides a front end to the user with options available for styling in CUI.

OAuth 2.0

> https://en.wikipedia.org/wiki/OAuth is an established protocol for Authentication. Many Financial Institutions utilize OAuth 2, and the Abe platform does too. The OAuth schema will require the various endpoints to request authentication grants and refresh authenticated tokens.

MFA Schemas

Multi Factor Authentication or MFA Schemas are the different ways a Financial Institution or FI might require extra steps for Authentication. This could be, sending you a text message with a code, or asking you security questions. Each of these Schemas will leverage a backend built for integrating with an FI through their API.
The two mentioned MFA Schemas are

Code

> The Code schema handles requests for an MFA token to be sent to the user via the FI, and validation for that token once provided.

Questions

> The Questions schema is able to gather questions from the FI for the user, and validate the users answers once provided.

Channel Connector

Overview

Channel Connector is the primary interface for clients (browsers, HTTP clients, third party channel managers) to engage with the Conversational AI infrastructure. It's acts as an HTTP API gateway, and web socket server, for end users's devices to communicate with an AI Agent.

Channels are created and manages through the Channel Settings page in CUI.

Service Dependencies

- Connects to the Conversate API
- Leverages the Events system
- Utilizes twilio as an SMS provider

Data Interface

Channel Connector will take data from an incoming channel and normalize it for Conversate. Here is an example of a normalized payload from Channel Connector to Conversate's `/message` endpoint.

```
{
  "agentId": "133fe2e8-ecbd-4b31-824f-ba6e33e97cb8",
  "message": {
    "type": "TEXT",
    "payload": {
      "utterance": "Hello"
    }
}
```

Here is an example response:

```
{
  "SMS": {
    "followUp": {
      "state": "general",
      "intent": "LIST_CAPABILITIES_AND_COMMANDS",
      "displayName": "FAQ: List Capabilities",
      "intentPackage": "System",
      "declinedScenario": "default"
    },
    "messages": [
      {
        "type": "TEXT",
        "payload": {
          "text": "Hey Amanda, I'm ready to talk money \n\nWould you
like a list of things I can help you with?"
        }
      }
    ],
    "metadata": {}
  },
  "TEXT": {
    "followUp": {
      "state": "general",
      "intent": "LIST_CAPABILITIES_AND_COMMANDS",
```

```
        "displayName": "FAQ: List Capabilities",
        "intentPackage": "System",
        "declinedScenario": "default"
      },
      "messages": [
        {
          "type": "TEXT",
          "payload": {
            "text": "Hey Amanda, I'm ready to talk money \n\nWould you
like a list of things I can help you with?"
          }
        }
      ],
      "metadata": {}
    }
  },
```

As you can see, there are various responses in this payload under `TEXT` and `SMS`. Channel Connector will take the response that correlates with the channel is interfacing with.

Table Of Contents

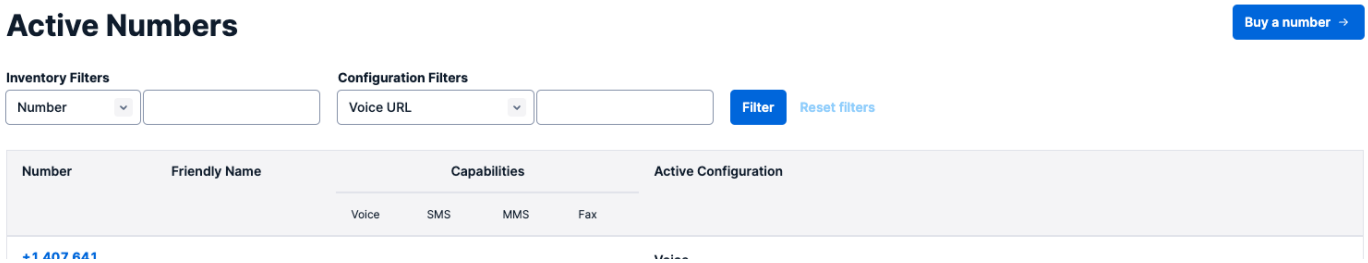
SMS Channel

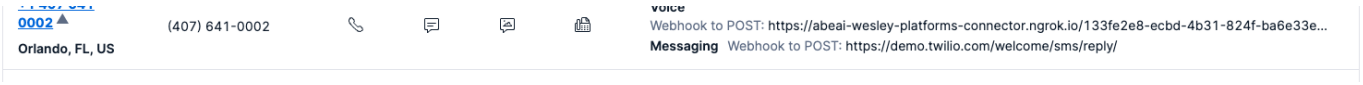
Overview

The SMS channel is a REST HTTP architecture.

Dependancies and Providers

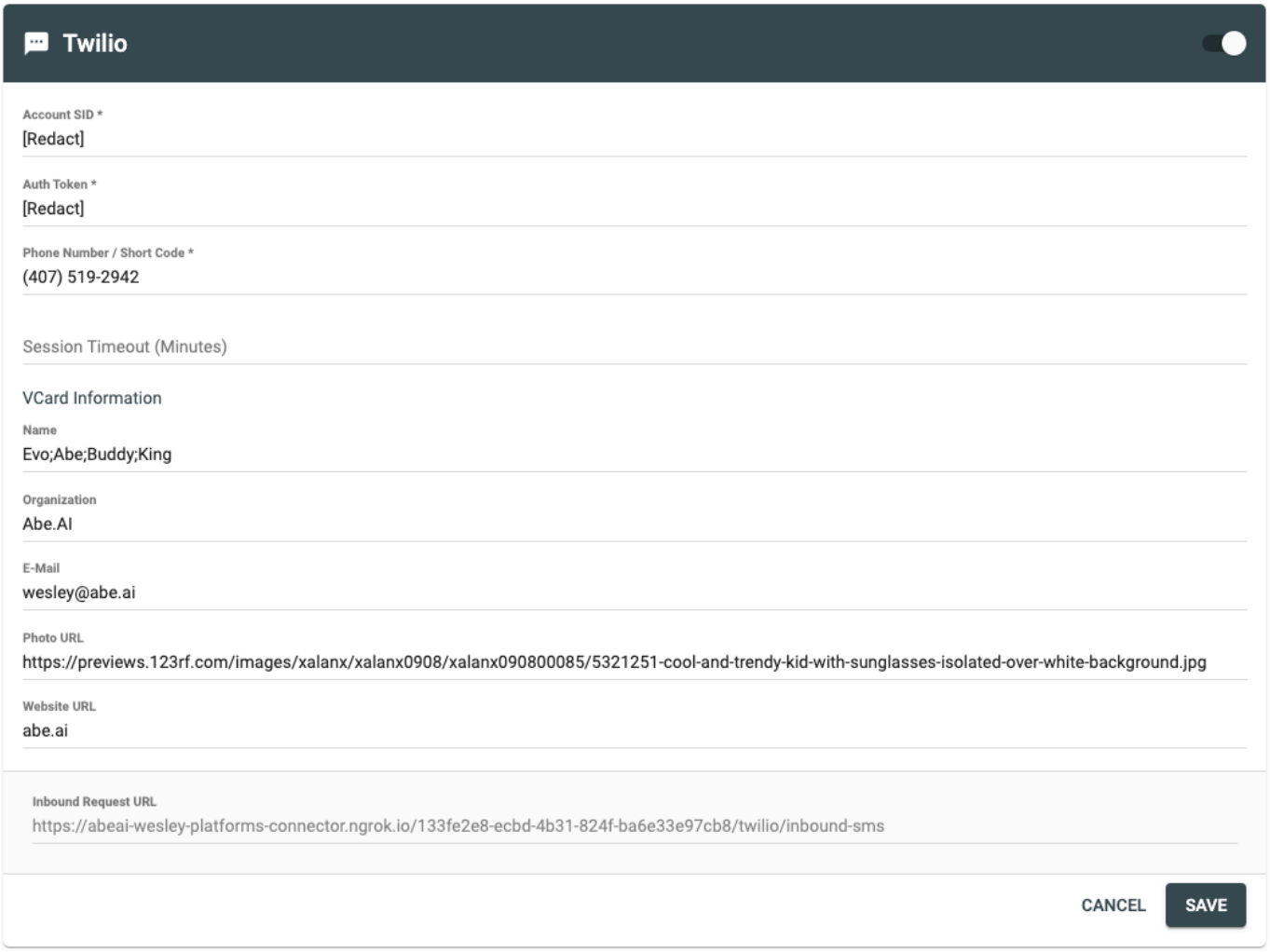
leverages Twilio as an SMS provider. A Phone Number purchased through Twilio will yield options to define webhooks. A webhook will be defined for your channel to POST for incoming SMS messages. Twilio will also provide an endpoint to POST to for outbound SMS messages.





Creating an SMS Channel

All channels are created in the Agent Channel Settings page in CUI.
The following screenshot shows an example of a completed SMS channel. Note that an `Inbbound Request URL` is generated.



Required Fields

- **Twilio Account SID and Auth Token.** These fields are necessary to authenticate with Twilio.
- **Phone Number.** Provided through the Twilio console.

Additional Optional Fields

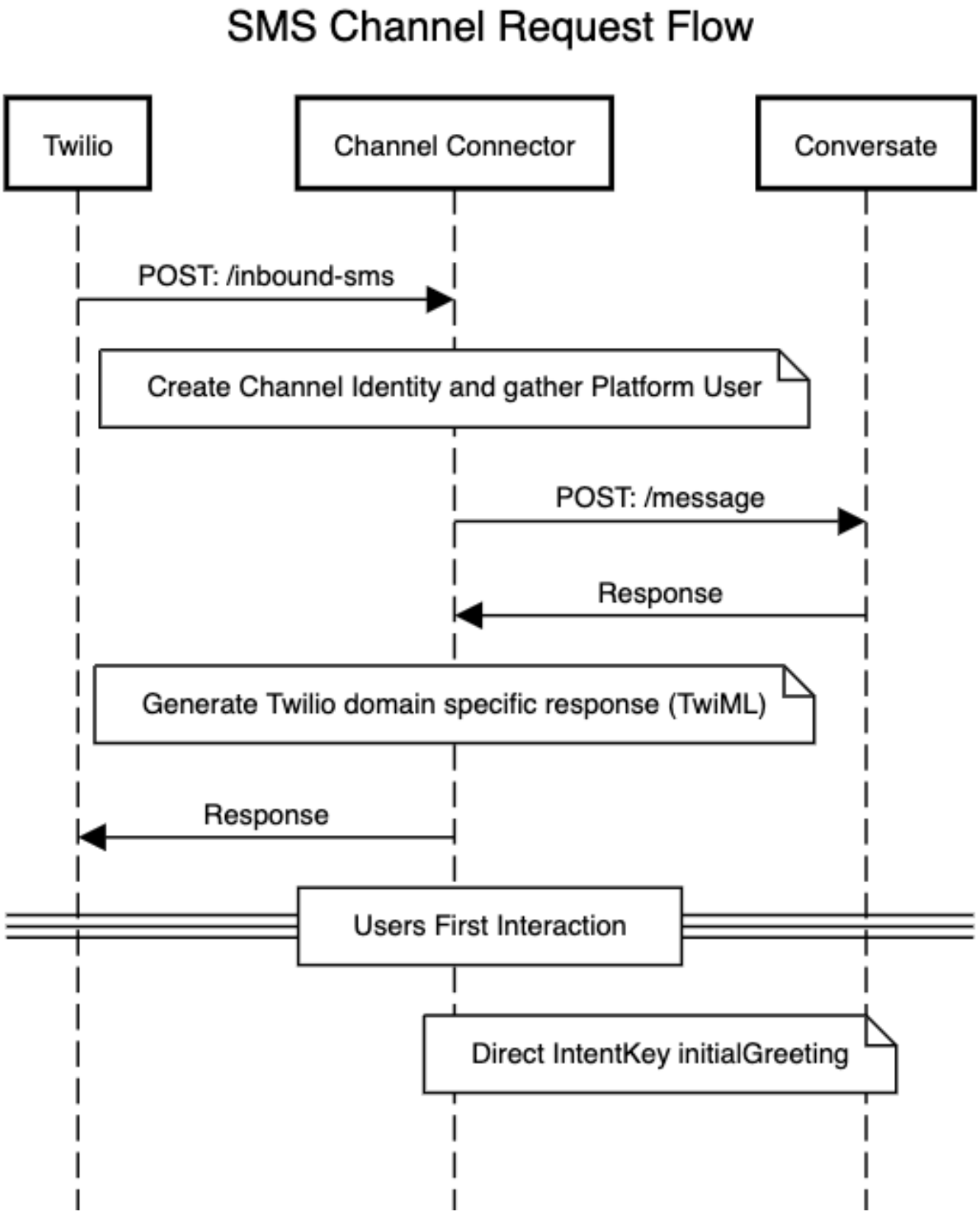
- **Session Timeout.** Specification for how long between messages the user will maintain their conversation session in minutes.
- **VCard Information.** To be compiled into a VCard to be sent to the user for additional information about the Agent.
  - Name
  - Organization
  - E-Mail
  - Photo URL
  - Website URL

Request Flow

The main functionality of the SMS channel interfaces with Twilio using their domain specific XML format, Twilio Markup Language or TwiML.

Requests come inbound from Twilio, a Channel Identity is created, as well as looking up any potential User Model for authenticated users.

Channel Connector will leverage the Conversate messages API to query responses. In the case that it is the users first interaction, Channel Connector will leverage a direct intentKey initialGreeting for a welcome message.



```
title SMS Channel Request Flow

participant Twilio
participant Channel Connector
participant Conversate

Twilio->Channel Connector: POST: /inbound-sms
note over Channel Connector: Create Channel Identity and gather
Platform User
Channel Connector->Conversate:POST: /message
Conversate->Channel Connector:Response
note over Channel Connector: Generate Twilio domain specific response
(TwiML)
Channel Connector->Twilio: Response
== Users First Interaction ==
note over Channel Connector, Conversate: Direct IntentKey
initialGreeting
```

Payloads

An example of an incoming Twilio Request to the generated `/inbound-sms` endpoint.


```
{
  "ToCountry": "US",
  "ToState": "FL",
  "SmsMessageSid": "SM1e3f2344ed8e92a85877ed8dc004bec0",
  "NumMedia": "0",
  "ToCity": "GENEVA",
  "FromZip": "",
  "SmsSid": "SM1e3f2344ed8e92a85877ed8dc004bec0",
  "FromState": "WA",
  "SmsStatus": "received",
  "FromCity": "",
  "Body": "Hello",
  "FromCountry": "US",
  "To": "+14075192942",
  "MessagingServiceSid": "MGa72bb6a80037bd7343717e56734be5d6",
  "ToZip": "32732",
  "NumSegments": "1",
  "MessageSid": "SM1e3f2344ed8e92a85877ed8dc004bec0",
  "AccountSid": "[Redact]",
  "From": "+12064765384",
  "ApiVersion": "2010-04-01"
}
```


An example response payload Channel Connector will send back to Twilio in their domain specific TwiML

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
  <Response>
    <Message>
      Hey Amanda, I'm ready to talk money \n\nWould you like a list of
things I can help you with?
    </Message>
  </Response>
```

Five9 Channel

Overview

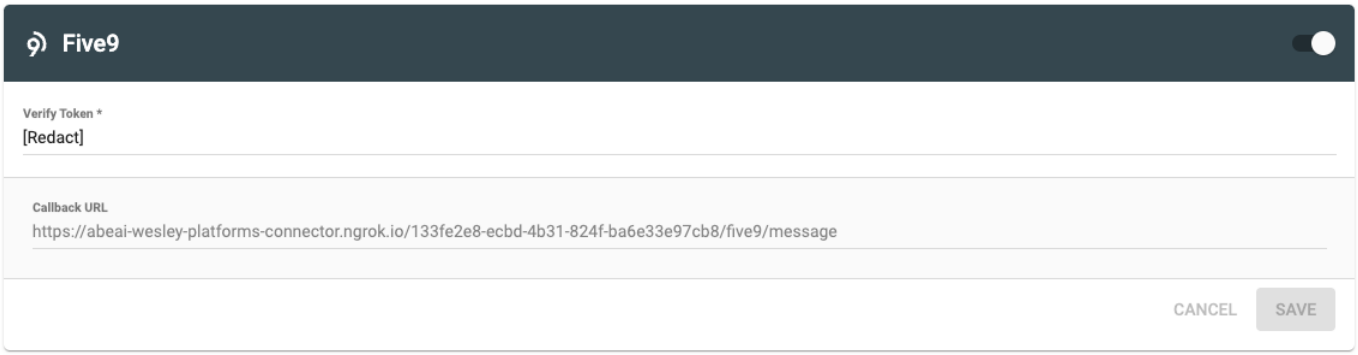
These channel types are built as integrations with other embedded chat providers. They exist as different channels, because they each need to interface with another proprietary API and thus a backend exists for each of them. Fundamentally they all function the same way, creating a `/message` endpoint to be provided to the Chat Widget provider.

Providers

Five9 is a digital communication provider we have integrations with.

Creating a Five9 Channel

All channels are created in the Agent Channel Settings page in CUI.
The only field required to make a Five9 channel is a Verify Token.



Request Flow

The request flow for Five9 channels follows the same flow as the Custom Channel Incoming Message Flow.

Custom Channel
Incoming messages follow a very simple flow. Inbound messages hit the Channel Connector through a `/messages` endpoint, and channel connector will normalize the payload and pass it along to Conversate. This is referred to in Conversate as an Utterance Query.

Glia Channel

Overview

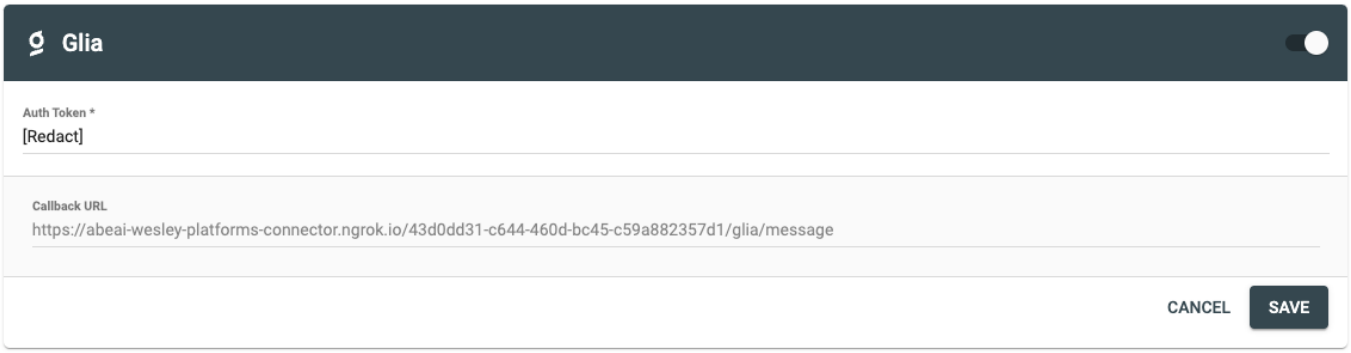
These channel types are built as integrations with other embedded chat providers. They exist as different channels, because they each need to interface with another proprietary API and thus a backend exists for each of them. Fundamentally they all function the same way, creating a `/message` endpoint to be provided to the Chat Widget provider.

Providers

Glia is a digital communication providers we have integrations with.

Creating a Glia Channel

All channels are created in the Agent Channel Settings page in CUI.
The only field required to make a Glia Channel is an Auth Token.



Request Flow

The request flow for Glia channels follows the same flow as the Custom Channel Incoming Message Flow.

Custom Channel
Incoming messages follow a very simple flow. Inbound messages hit the Channel Connector through a `/messages` endpoint, and channel connector will normalize the payload and pass it along to Conversate. This is referred to in Conversate as an Utterance Query.

Directives

The embedded Glia widget supports directives, these will be included in Conversate's response payload as a sibling of `messages`
Here is an example of a Glia directive. Note that templating variables are supported.

```
"directives": {
    "channel": {
        "type": "gliaCobrowse",
        "template": true,
        "payload": [
            {
                "type": "custom_command",
                "properties": {
                    "virtual_assistant_cobrowsing": {
                        "type": "navigate",
                        "target": "{{agent.urlOlbRoot}}{{agent.
urlPathAccountDetails}}{{account.id}}"
                    }
                }
            }
        ]
    }
},
```

PopIO Channel

Overview

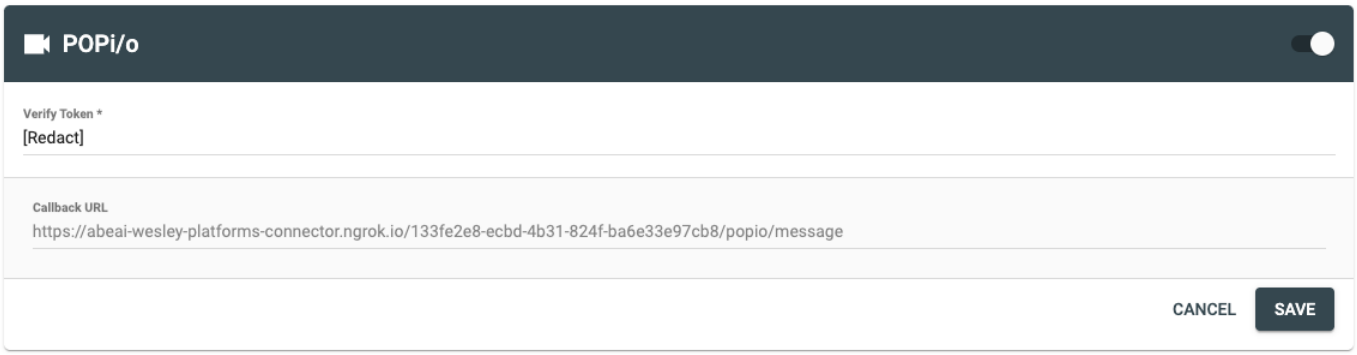
These channel types are built as integrations with other embedded chat providers. They exist as different channels, because they each need to interface with another proprietary API and thus a backend exists for each of them. Fundamentally they all function the same way, creating a `/message` endpoint to be provided to the Chat Widget provider.

Providers

PopIO is a digital communication provider we have integrations with.

Creating a PopIO Channel

All channels are created in the Agent Channel Settings page in CUI.
The only field required to make a PopIO channel is a Verify Token.



Request Flow

The request flow for PopIO channels follows the same flow as the Custom Channel Incoming Message Flow.

Custom Channel
Incoming messages follow a very simple flow. Inbound messages hit the Channel Connector through a `/messages` endpoint, and channel connector will normalize the payload and pass it along to Conversate. This is referred to in Conversate as an Utterance Query.

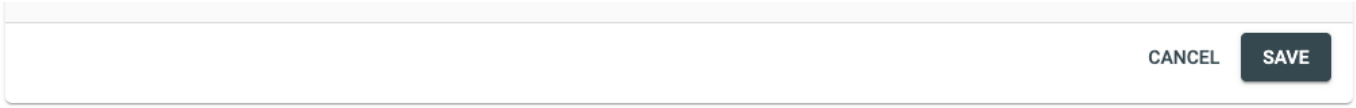
Custom Channel

Overview

The Custom Channel is an open ended channel type, where endpoints are created for submitting messages `/message` and refreshing auth tokens `/oauth/token`. This provides a dynamic way to interface with the platform that is agnostic of providers.

Creating a Custom Channel

All channels are created in the Agent Channel Settings page in CUI.
The following screenshot shows an example of a completed Custom channel. Note that a `Callback URL` is generated.



Required Fields

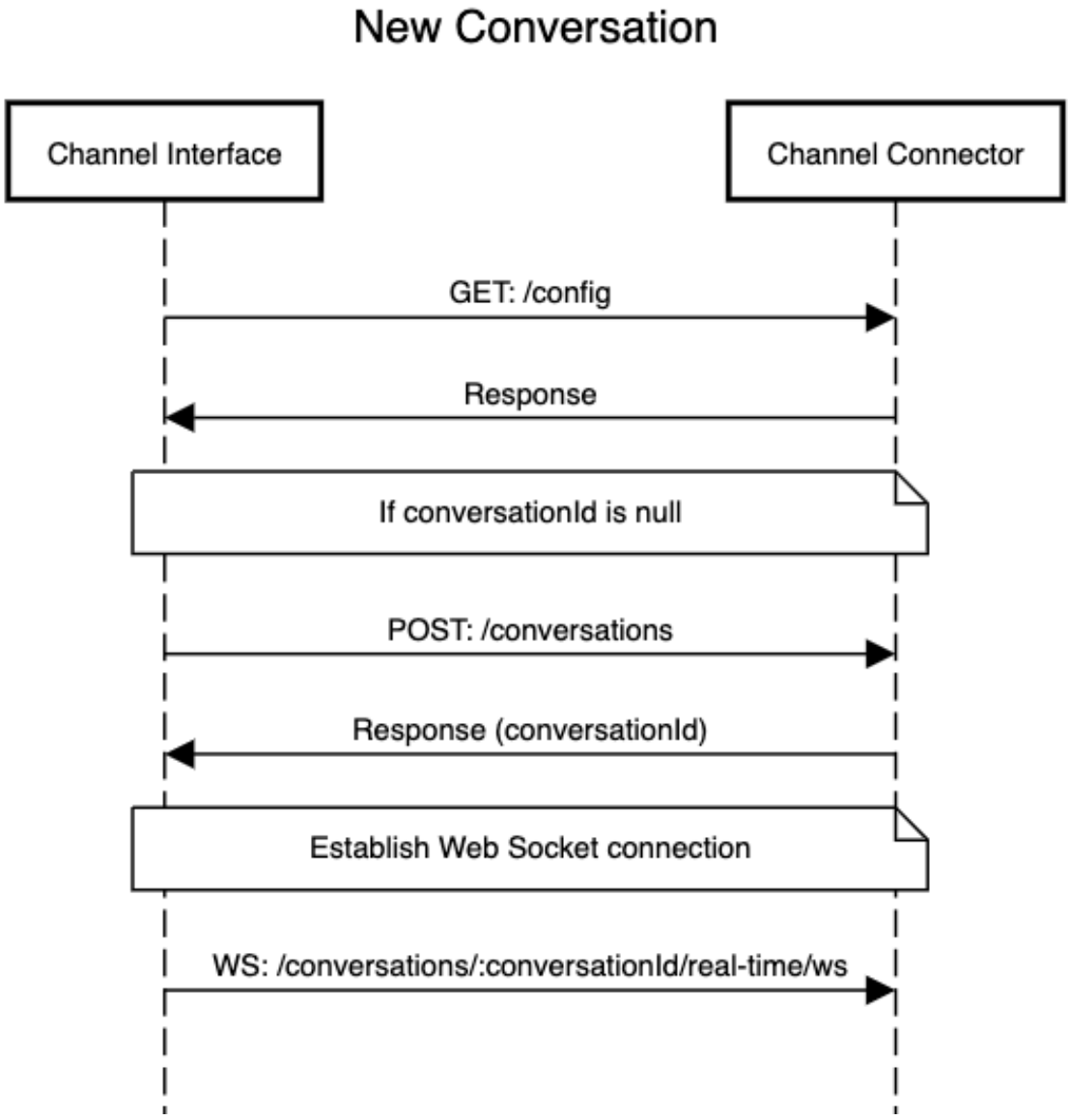
- **Verify Token**. For Verifying requests.

Additional Optional Fields

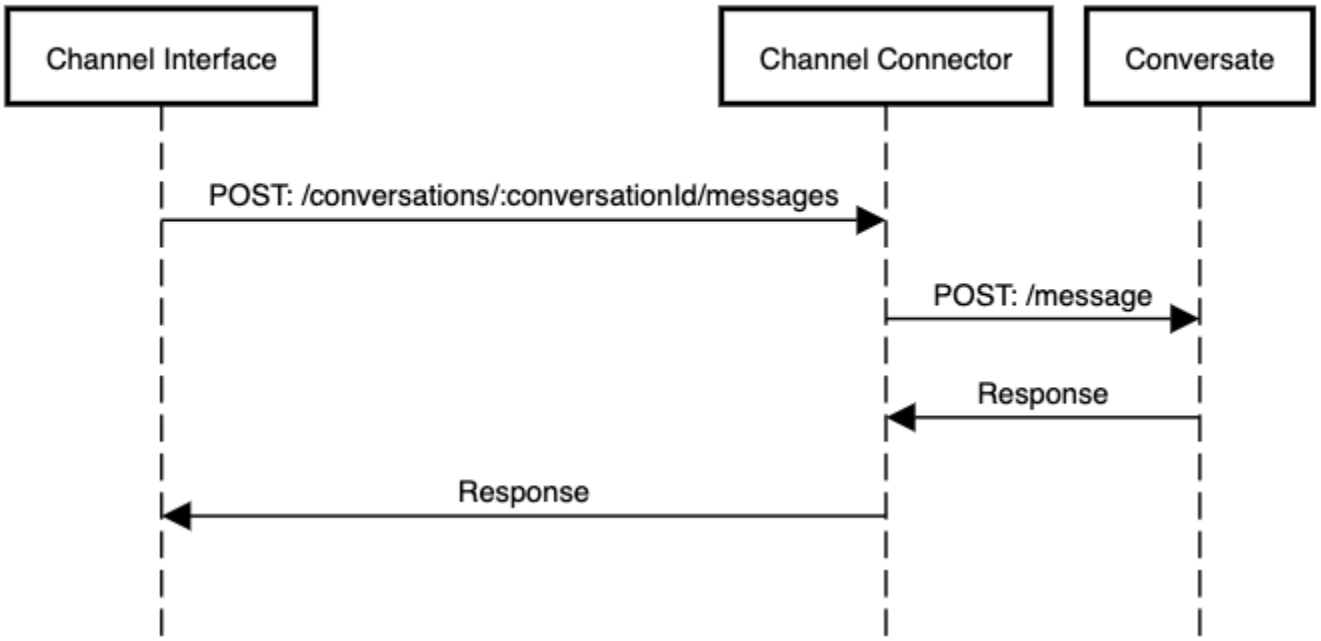
- **Client ID.** Used for OAuth.
- **Client Secret.** Used for OAuth.
- **Session Timeout.** Length of time a user session remains without interaction in minutes.

Request Flow

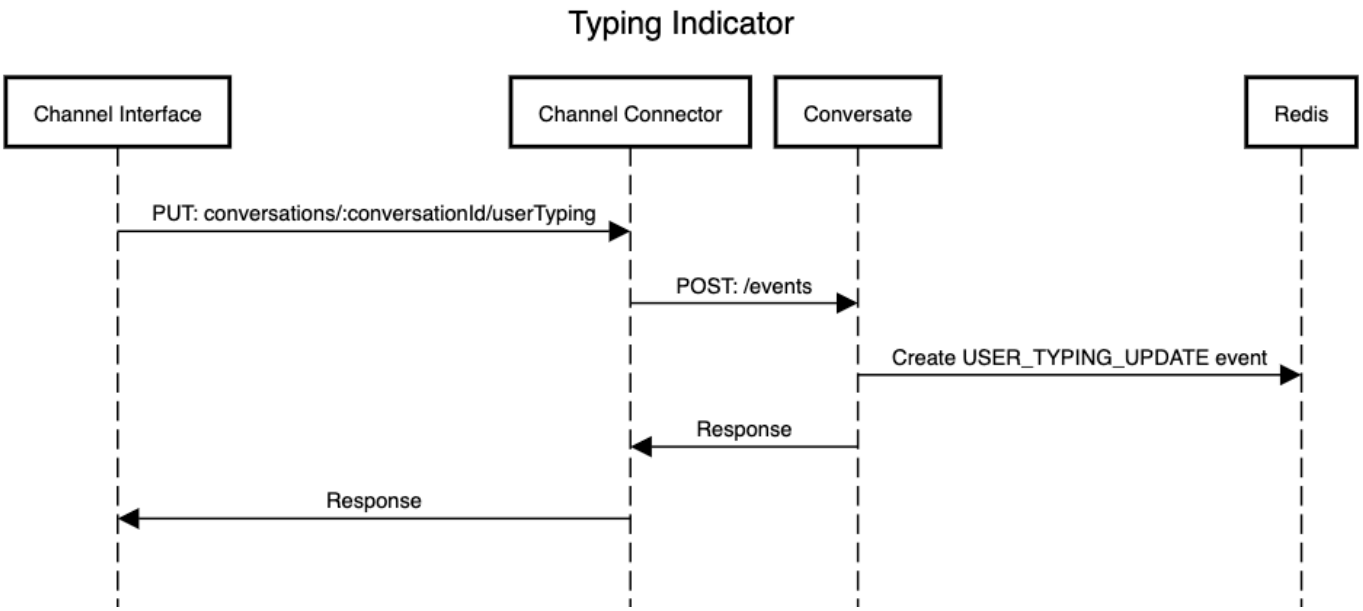
The first interaction with a Custom Channel initializes the conversation. The Channel Interface can make a call to `GET /config` for channel and agent data, and `POST /conversations` to initialize a conversation and fetch the conversation ID. Additionally the Channel Interface can establish a web socket connection with `/conversations/:conversationId/real-time/ws` if the channel supports polling for web socket events.



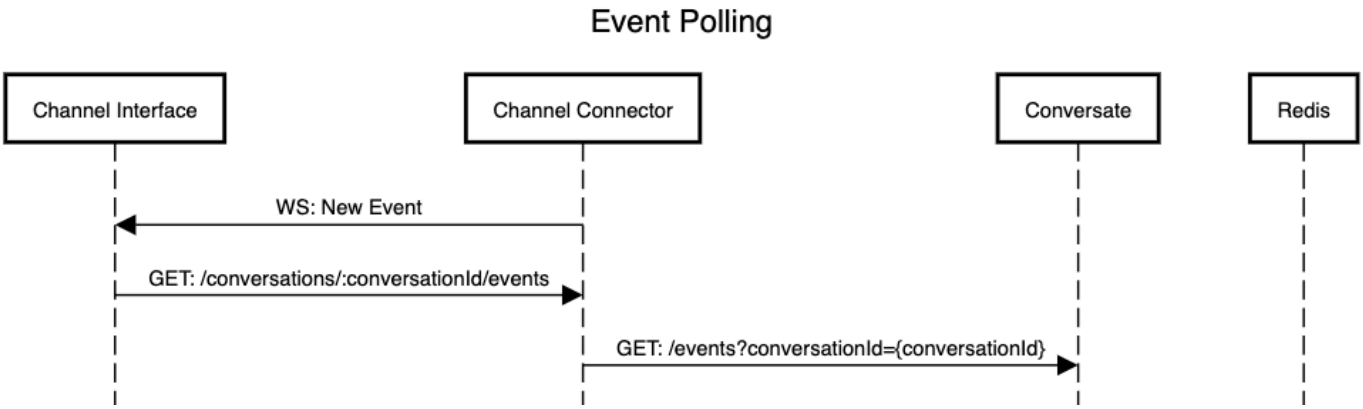
Incoming messages follow a very simple flow. Inbound messages hit the Channel Connector through a `/messages` endpoint, and channel connector will normalize the payload and pass it along to Conversate. This is referred to in Conversate as an Utterance Query.

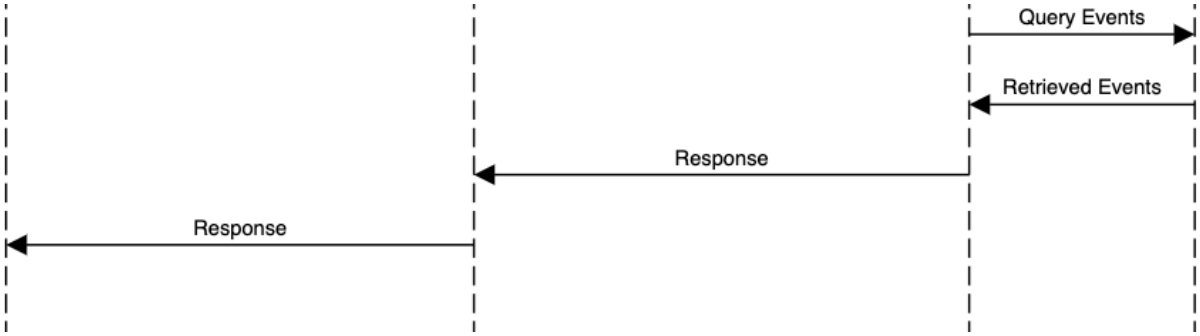


Some custom channels support a Typing Indicator. The Custom Channel supports this with an endpoint `PUT /conversations/:conversationId/userTyping`. Channel Connector manages this with the Conversate Mesasges API with `POST /events`.



Some custom channels interface with Event Polling. For this, on establishing a new conversation a web socket connection is made, and Channel Connector will ping that connection with a New Event message. This tells the Channel Interface they can call `GET /conversations/:conversationId/events` for the events related to their conversation.





Embedded Web Channel

Overview

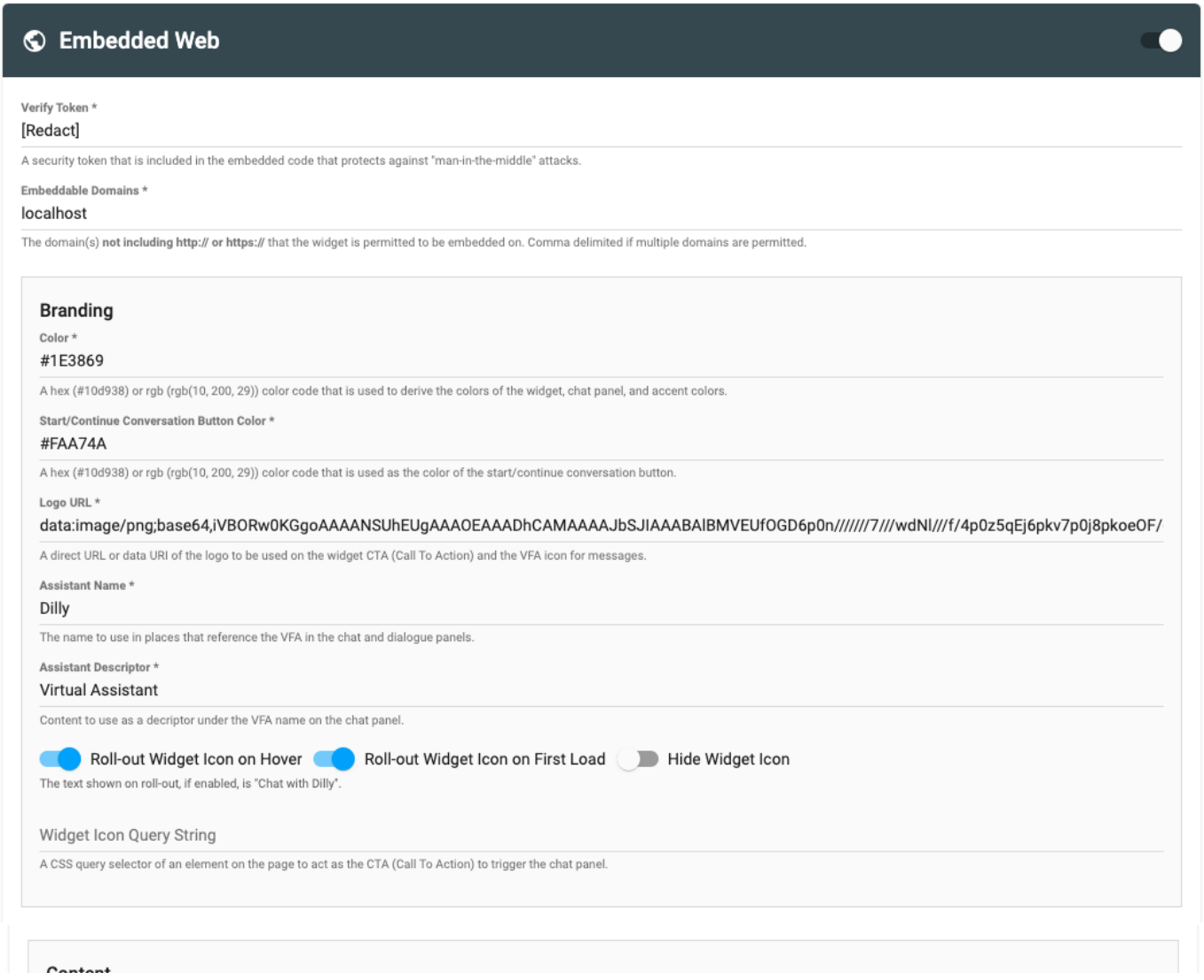
The Embedded Web channel allows for use of the Embedded Web Widget . Under the hood, the Embedded Web Channel uses the Custom Channel pipeline.

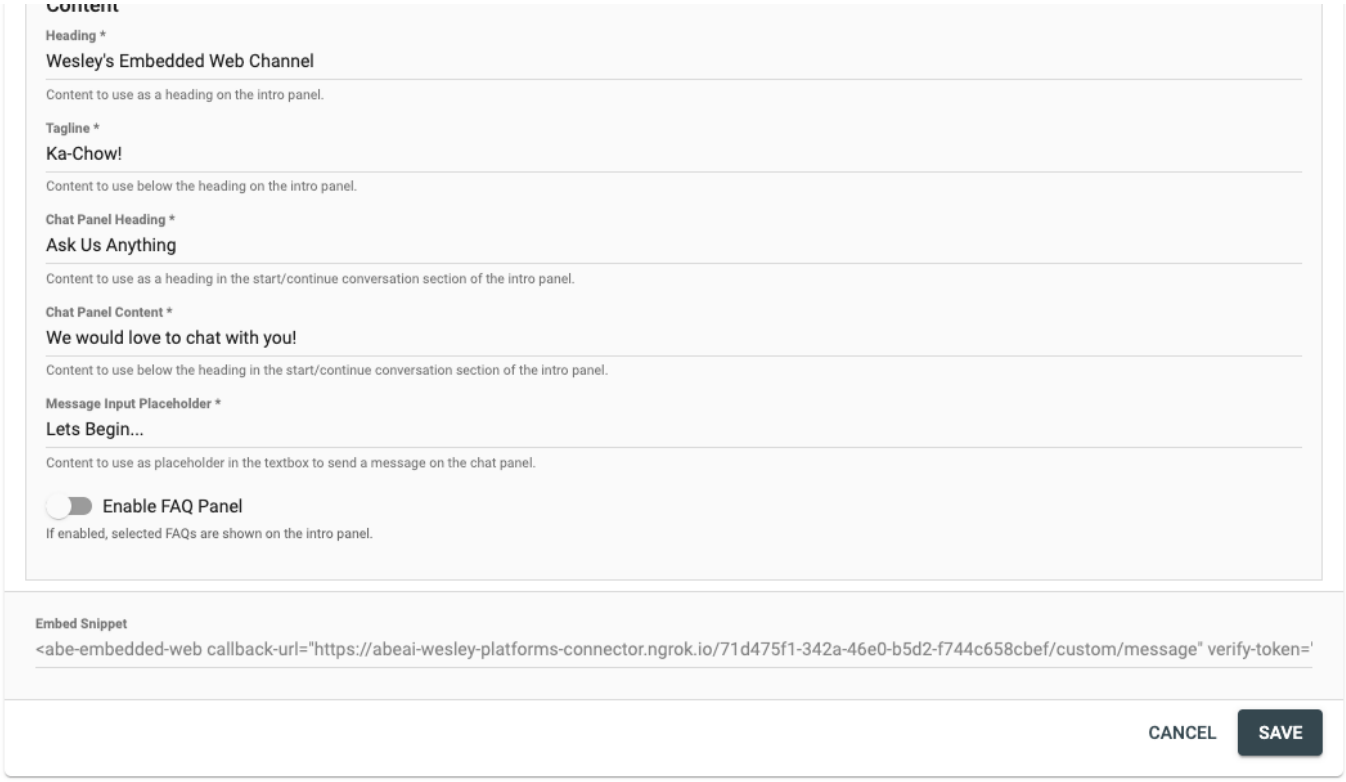
Dependancies

The Embedded Web Channel uses Conversate's Events API as well as the Conversate Messages API.

Creating an Embedded Web Channel

All channels are created in the Agent Channel Settings page in CUI.
Note the option to list Embeddable Domains, and the provided Embed Snippet.





Required Fields

- **Verify Token.** Used to validate incoming requests from the Embedded Widget.
- **Embeddable Domains.** A comma separated list of domains the Widget is allowed to exist in.
- **Branding.** Information used for styling the Widget in your web page.
  - **Color.** A hex (#10d938) or rgb (rgb(10, 200, 29)) color code that is used to derive the colors of the widget, chat panel, and accent colors.
  - **Start/Continue Conversation Button Color.** A hex (#10d938) or rgb (rgb(10, 200, 29)) color code that is used as the color of the start /continue conversation button.
  - **Logo URL.** A direct URL or data URI of the logo to be used on the widget CTA (Call To Action) and the VFA icon for messages.
  - **Assistent Name.**

    The name to use in places that reference the VFA in the chat and dialogue panels.
  - **Assistent Descriptor.** Content to use as a decriptor under the VFA name on the chat panel.
  - **Toggles.** Used to specify how the Widget is interact-able on the web page
    - **Roll-out Widget Icon on Hover.**
    - **Roll-out Widget Icon on First Load.**
    - **Hide Widget Icon.**
- **Content.** Specifies the text used to populate the Widget.
  - **Heading.** Content to use as a heading on the intro panel.
  - **Tagline.** Content to use below the heading on the intro panel.
  - **Chat Panel Heading.** Content to use as a heading in the start/continue conversation section of the intro panel.
  - **Chat Panel Content.** Content to use below the heading in the start/continue conversation section of the intro panel.
  - **Message Input Placeholder.** Content to use as placeholder in the textbox to send a message on the chat panel.
  - **Toggles.**
    - **Enable FAQ Panel.** If enabled, selected FAQs are shown on the intro panel.

Additional Optional Fields

- **Branding.**
  - **Widget Icon Query String.** A CSS query selector of an element on the page to act as the CTA (Call To Action) to trigger the chat panel.

Request Flow

The Embedded Web Channel utilizes an Event based architecture with a Web Socket connection as well as a HTTP REST messaging endpoint.

Requests come inbound from the Embedded Widget, utilizing Channel Connector as a middle man between the Widget and the Conversate Events API for polling as well as the Conversate Messages API.

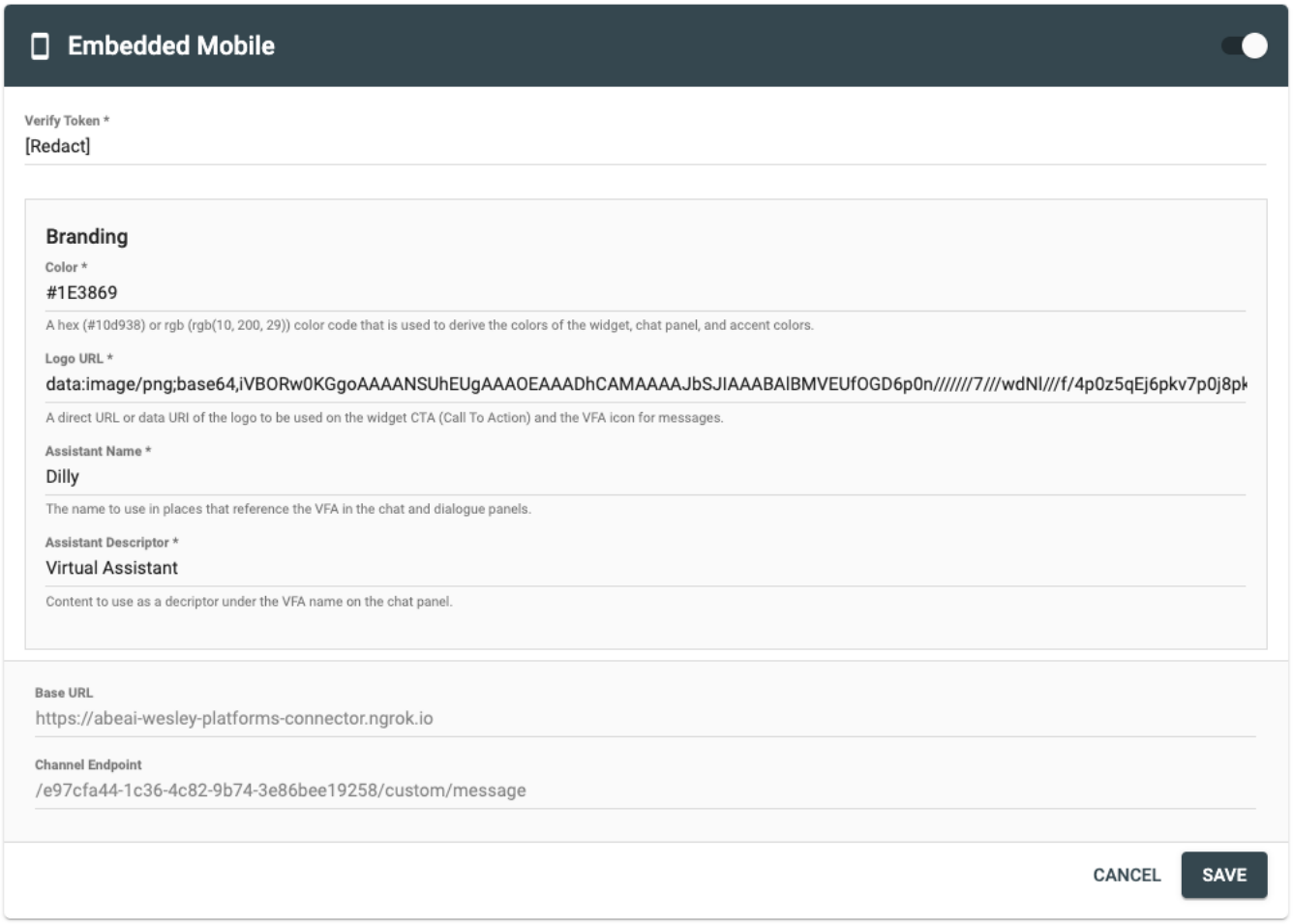
Embedded Mobile Channel

Overview

The Embedded Mobile channel allows the agent to interface with the mobile apps for iOS and Android. Similar to the Embedded Web channel, under the hood, the Embedded Mobile Channel uses the Custom Channel pipeline.

Creating an Embedded Mobile Channel

All channels are created in the Agent Channel Settings page in CUI.
The following screenshot shows a completed Embedded Mobile Channel.



Required Fields

- **Verify Token.** Used for verifying requests.
- **Branding.** Some styling for the app.
  - **Color.** A hex (#10d938) or rgb (rgb(10, 200, 29)) color code that is used to derive the colors of the widget, chat panel, and accent colors.
  - **Logo URL.** A direct URL or data URI of the logo to be used on the widget CTA (Call To Action) and the VFA icon for messages.
  - **Assistant Name.** The name to use in places that reference the VFA in the chat and dialogue panels.
  - **Assistant Descriptor.** Content to use as a decriptor under the VFA name on the chat panel.

Conversate

Conversate is the driver for the Dialogue Engine and acts as the hub for the Conversational AI Platform.

Within the context of the Conversational AI ecosystem a conversation can be viewed as a series of intents, trained on sample utterances, being invoked by end user requests.

The primary feature of the Dialogue Engine is the '**brain**', which is used to process events and messages, then track and control the state of all conversations.

Conversate can also be accessed by other microservices within the platform as well as external parties through the Abe Platform API.

The Web Socket provides support for the CUI application which allows end users to configure their Organization's Assistants and its various Applications, as well as manage users and their permissions within the system.

Service Dependencies

- Connects to
  - Integration Services
  - Channel Connector
  - CUI
  - Auth Web Client
  - NLU Python Services

Table of Contents

Tech Stack

Express

The HTTP server utilized is Express.

Postgres

Conversate utilizes a Postgres database and uses migrations to build out schema and seed initial data.

Redis

Conversation data is stored in Postgres but current state of the conversation memory is stored in Redis database. Conversations expire after 10 minutes.

Socket-IO

CUI communicates with the Conversate API via web-socket, and we use the socket-io library to establish the connection. The web-socket events are used more like REST HTTP requests, rather than typical BE FE broadcast events, there is a clear request response pattern used.

Conversations

Various events trigger actions across the dialogue engine. To process a message event, the brain is initialized first by loading an agent. Then, the Brain & Memory is loaded from Redis. The Redis short term memory expires after 10 minutes of conversation inactivity. The conversation is then loaded from the database and determines which conversation mode is appropriate.

There are two primary flows or modes of conversation; Conversation handled by the Conversate or Human Handoff . Human Handoff is when the conversation requires human interaction. At this point Conversate does not process user messages and it switches control to one of the configured HH Providers**.** When the provider has ended its session, control is returned. Otherwise, the configuration for the conversation is check against intents and entities. Custom entities are fetched from the Integration Service Virtual Banking . The message is then processed by the NLU engine which responds with the predicted intent and entities with a confidence level. The response is interpolated into the correct intent. Memory is updated and the response is returned.

- Conversation Flow
- States

Conversation Flow

The importance of the brain is significant because it controls the conversation flow from receiving utterance details from the Channel Connector getting possible intents from the NLU , and response requests additional information from theIntegration Service Virtual Banking for additional entities. The formatted response is returned the Channel Connector.

First the message is received from Channel Connector in a format that Conversate understands, including information about what channels format to retrieve intent information (ALL, SMS, etc). A common use case would be SMS so we will that for the examples.

The brain will immediately go into “Process Message” mode.

Example incoming message.

```
{
  channelIdentity: {
    id: "6f9d0cdc-d19b-4279-9fca-46e54c88f5d7__a72d68b1-f68d-4f07-b0f2-
d3bf97216ec4",
    agentId: "a72d68b1-f68d-4f07-b0f2-d3bf97216ec4",
  },
  user: {
    id: "B2F4B621-738C-48A1-BD06-AB6C78D5D26B",
    authenticated: false,
    firstName: "Abraham",
    lastName: "Lincoln",
    locale: "en-US",
    sourceType: "",
    sourceId: "",
    yodleeLogin: "",
    yodleePassword: "",
    timezone: "America/New_York",
    agentId: "a72d68b1-f68d-4f07-b0f2-d3bf97216ec4",
  },
  channel: "ALL",
  testing: false,
  message: {
    payload: {
      intent: "",
      utterance: "hello",
    },
  },
  agentId: "a72d68b1-f68d-4f07-b0f2-d3bf97216ec4",
  userSpecificEntities: undefined,
  parameters: undefined,
}
```

One can see that the message payload contains user information, the Channel `ALL`, the `Agent Id` where the utterance originated as well as the utterance `Hello`.

The brain is initialized with message details and initializes the CUI Config with preferred training status groups. The training status can be `READY`, `PARTIALLY READY`, or `TRAINING`. The grouping are determines by whether the conversation is initiated by the Dialogue Simulator in CUI or from the End User like in Embedded Web. Asynchronously, the Integration Service(s) (IS) are initialized as well, with both the config and IS attached to the brain. There is also a flag to determine if LINC will be used but this is to be deprecated. The brain then loads the memory, first initialized. The brain’s short term memory keeps track of the utterances, responses, intents, entities, store, store stack and the mode.

The conversation is then checked either from the message or the memory for the conversation id. If neither of those exist a new conversation ID is established. At this point the conversation is then checked to see if it’s a human handoff conversation which will trigger a different flow outside of conversate changing the brain’s state of to `humanHandoff*`. An utterance that would trigger human handoff would be something like `Speak to Agent`. This in turn would follow the same above flow, but then split off to the humanhand provider and trigger a state change that trigger the` Move Money` Journeys .

States

This `Hello` case the memory's State would be in `general`. There are multiple states that memory can be in. In particular, `general, utility, humanHandoff*`. Human Handoff specifically can move through multiple. states such as

- `humanHandoffSessionWaitingFor`
- `humanHandoffInquiryTypeSelection`

Initial `Hello` Utterance Brain state.

```
{
  initialState: "general",
  state: "general",
  mode: "default",
  utterances: [   "hello",  ],
  responses: [  ],
  intentKeys: [  ],
  intents: [  ],
  entities: [  ],
  storeStack: [  ],
  store: {  },
  consecutiveFallbacksCount: 0,
}
```

Dialogue Engine

The Dialogue Engine orchestrates the requests from the Channel Connector. It does this by identifying the Intent of the incoming message via the NLU Engine and then generating a proper response based on the agent configuration. If the incoming message requires external data, it will send a request to the Integration Service to gather Financial Institution data and return the formatted response back to the Conversate.

Events & Messages

The are several events that happen with in the system. The message events contain the event source type information regarding message origination.

- Event Source Types
- Events
  - Human Handoff Events
- Diagrams

Event Source Types

- AI AGENT
- USER

Events

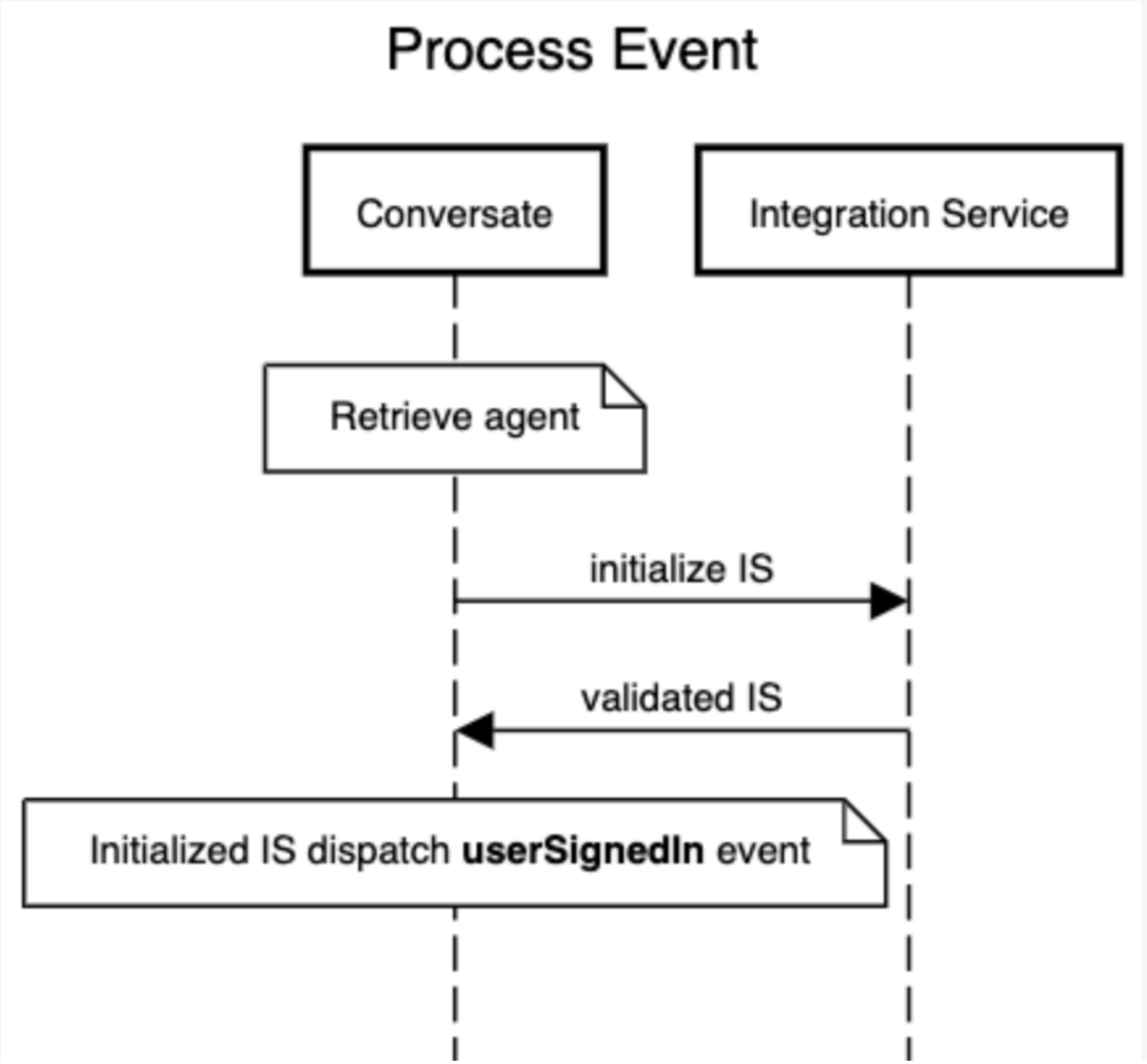
- MESSAGE_RECEIVED
  - Message received, contains brain context and conversations as well as response and intent information.
- MESSAGE_SENT
  - Message sent, contains brain message type payload, parameters and meta data.
- AGENT_TYPING_UPDATE
  - Publish event for AI or Human typing event.
- SET_USER_FEEDBACK
  - Set positive or negative feedback for a message meta data. Feedback count is updated on the conversation.

Human Handoff Events

- HUMAN_HANDOFF_SESSION_INITIALIZATION_SUCCEEDED
  - Sets the active human handoff provider session and wait time queue details and updates the active and last human handoff session Conversation record.
- HUMAN_HANDOFF_SESSION_INITIALIZATION_FAILED

- Processes the message with failed details.
- HUMAN_HANDOFF_SESSION_WAIT_TIME_UPDATE
  - Updates the active and last human handoff session Conversation record. Builds an intent message to update the agent.
- HUMAN_HANDOFF_SESSION_ESTABLISHED
  - Removes the wait time update value on the human handoff session and sets the human agent and session information. Updates the active and last human handoff session Conversation record. Builds an intent message to update the agent. Processes the message with session established details.
- HUMAN_HANDOFF_SESSION_ENDED
  - If task queue id exists then Survey and Question workflow to gather survey metadata. Builds an intent message to update the agent session ended. Processes the message with session ended details.
- HUMAN_HANDOFF_SESSION_USER_IDLE_WARNING
  - Builds an intent message to update the agent session idled. Processes the message with session idle details.
- HUMAN_HANDOFF_SESSION_USER_IN_QUEUE
  - Refreshes Short Term Memory.
- USER_TYPING_UPDATE
  - For an active human handoff session, the provider's chat client trigger typing action. Event published with type and payload.

Diagrams



Brain & Memory

The brain acts as a state machine for the dialogue engine but is currently treated as a God Object that is passed around the dialogue engine that keeps track of the conversation state.

The brain consists of Short Term and Day Term Memory. The memory is referenced by the current User ID or the Channel Identity ID as the base key and a environment variable `REDIS_KEY_PREFIX`.

The memory keeps track of conversations the initial state, utterances, responses, intentKeys, intents, entities, and the store. Conversations are referenced by their conversation id. Conversations 'expire' from memory after 10 minutes.

Abe Platform API

The Abe Platform API supports System Internal, Platform, Messages, or Flex API authentication.

- Access Token Types

Access Token Types

The are three types of access tokens that authorize access to specific endpoints.

1. System Internal - Driven by environment variables.
2. Platform - Platform level API tokens access to entire account and all agents.
3. Messages - Messages API tokens only access to message level endpoints.
4. Flex - Flex API tokens have access to endpoints to support flex contact center.

Actions	Internal	Platform	Flex	Messages
accounts	getAccounts			
agents	createAgent deleteAgent getAgent getAgents getMessages createIntegrationServices deleteIntegrationServices importDialogConfig exportDialogConfig	createAgent deleteAgent getAgent getAgents getMessages createIntegrationServices deleteIntegrationServices importDialogConfig exportDialogConfig	getMessages	getMessages
flex			Flex	

Flex

Available API actions

- Chat
  - Add Member
  - Remove Member
  - Update Member
- Engagement
  - Complete
  - Join
  - Wrapping
- Messages
  - Get By Conversation Id
- Queue
  - Add Queue
  - Delete Queue
  - Get Queue
  - Update Queue
- Survey
  - Question
    - CRUD Questions
  - Create Survey
  - Delete Survey
  - Get Survey

    - Update Survey
- Sync
    - Get Token
- Team
    - Add Queue
    - Add Team
    - Add Worker
    - Delete Team
    - Get Team
    - Remove Queue
    - Remove Worker
    - Update Team
- Video
    - Get Token
- Worker
    - Get Worker
    - Update Worker

Users, Groups & Roles

Supports User Permissions in
CUI

For CUI, there was a need for limiting users to specific actions within an application.

Groups are *owned* by either an Account (Organization) or an Agent and contain a set of permissions that grant users access to various actions for its owner. These actions are are defined as permissions.

> An Account or Agent can have many of groups.
>
> A User can have membership of many groups.

Special case. There is an 'Root Group" that supports access to any Account/Organization. All actions across the system are permitted for these users.

*Group design will be deprecated to support Users and Roles.

Groups are no longer assigned per an application but instead at global level. Groups are now called Roles.

Roles are defined per an Organization and can be assigned to specific agent or assistants.

Users can have different roles across Agents. Users may belong to 1 or more Roles.

- CUI Users
- Permissions
- Application Types
- Roles

Application Types

1. Conversation Designer (CUI)
2. Contact Center
3. Admin
4. Engagement Portal (Not yet Implemented)

Admin Application

Management is handled within the Admin Application.

Users

Users exist are uniquely identified in the platform by email address. A user may have access to multiple organizations and accounts. The groups give them access to their accounts. Roles are assigned to users, where they can have multiple roles across the system.

CUI Users

Users exist are uniquely identified in the platform by email address. A user may have access to multiple organizations and accounts. The groups give them access to their accounts. Roles are assigned to users, where they can have multiple roles across the system.

- User Requirements
- SuperUser

User Requirements

1. Must be unique email address.
2. First Name
3. Last Name
4. Password must be at least 15 characters. Must use 3 of 4 characters, lowercase, uppercase numbers and special characters.

SuperUser

There is a need to have users that have elevated access to the CUI that would have sudo access to all accounts and their agents and applications. Users assigned to the Super User groups have access to entire platform. They cannot be added to the groups through any API calls to Conversate, they must be added by an Operations user. This type of users would typically be an Operations or Abe User for a managed account customer. Platform customers would manage their Super Users similarly.

These super users exist in the Root Group and are assigned to the SuperUser role. The group will always have an UUID of `00000000-0000-0000-0000-000000000000`.

Permissions

Permissions

Permissions are stored in a JSON file with the following structure. New permissions need to be defined in this file.

The following tables are just an example and not an exhaustive list of available permissions.

Permissions.json

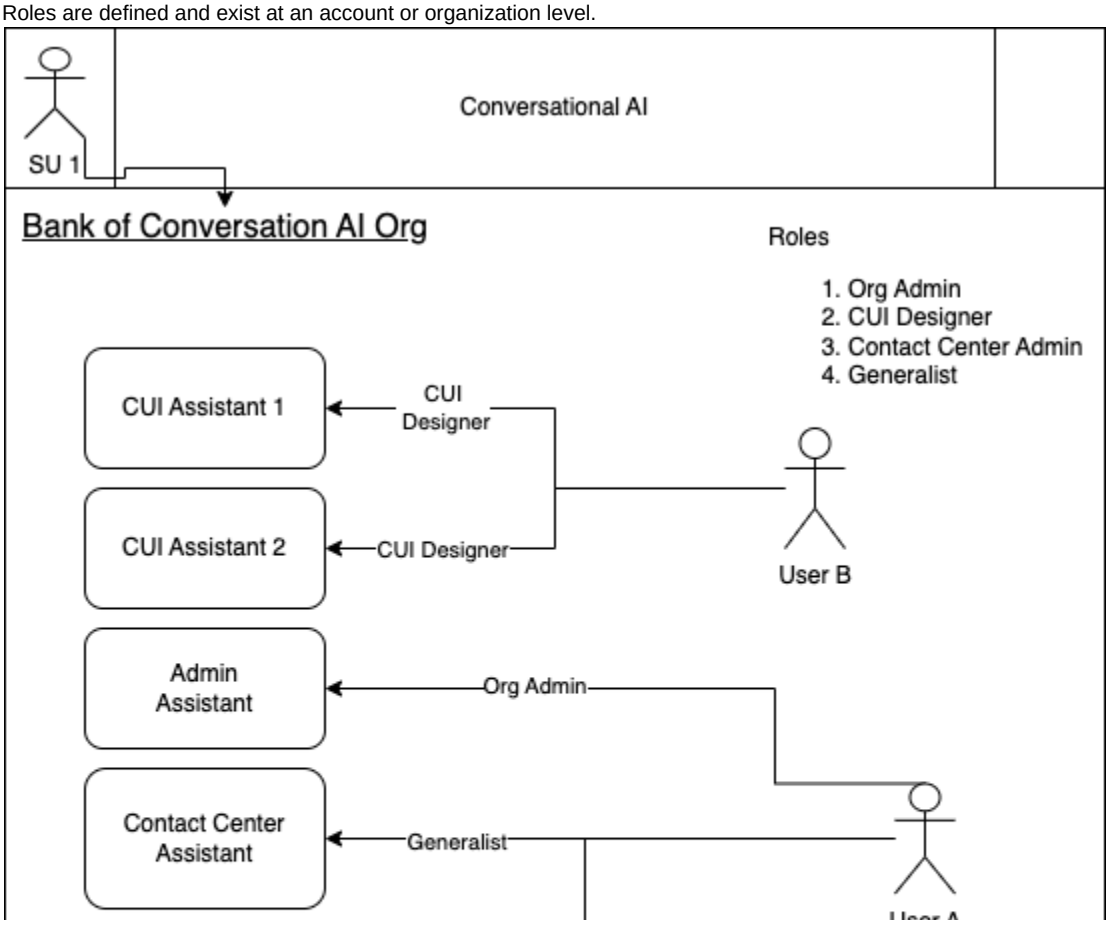
Application Types

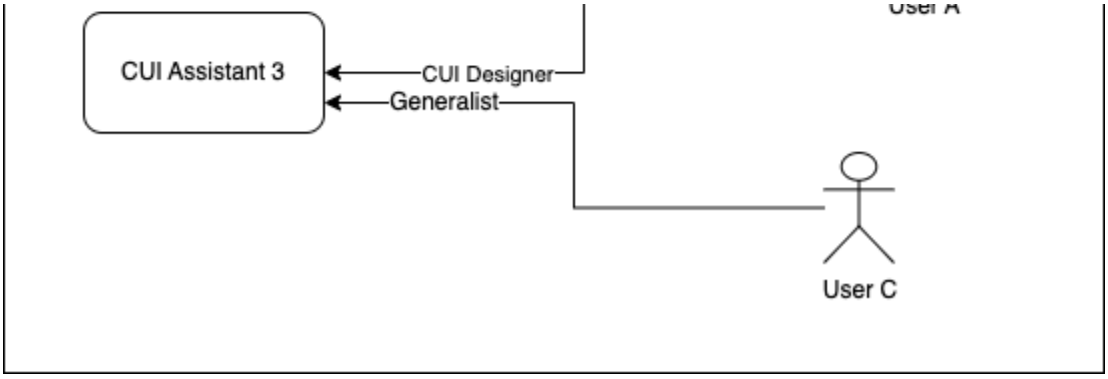
There are 4 Applications available to an Assistant.

1. Conversation Designer (CUI)
2. Contact Center
3. Admin
4. Engagement Portal (Not yet Implemented)

Roles

Roles are defined and exist at an account or organization level.





Event Bus

The Event Bus within Conversate creates the Publisher Subscriber Pattern using the Redis Client.

The Event Bus is utilized in the following scenarios.

1. WebSocket
2. Conversations

Integration Services

The integration service acts as the interface between the banking backends and Conversate, as well as defines the configuration schema for CUI. Intent method executions, journeys, feature flags, and location configurations are defined here. Currently, the design supports multi-tenancy, multiple banking clients can and are supported by one instance of an integration service.

Service Dependencies

- Connects to
  - Conversate

Table of Contents

Integration Service Virtual Banking

The integration service acts as the interface between the banking backends and Conversate, as well as defines the configuration schema for CUI. Intent method executions, journeys, feature flags, and location configurations are defined here. Currently, the design supports multi-tenancy, multiple banking clients can and are supported by one instance of an integration service.

Adapter definitions for each of the banking backends allow an authenticated user to access authorized information such as listing account data or calculating net worth. The adapters reach out the banking client's API and caches the responses. The responses are then transformed by the mappers into an Abe Platform readable format and returned to Conversate for further processing.

There are also intents that require more complex multi step processing and the definitions, Journey definitions, specifically Money Movement lives here.

Configuration Schema

There are two main sections of the Configuration Schema. Core and Features.

Core

The intents are executed for configured bank clients, not all clients will support every intent but can be defined and implemented as needed. This area contains adapter schema. Specifically, the banking client base url schema would be defined here, but it not limited to just that. This would require the user in CUI to hydrate the config with banking client urls at a minimum. There are also defaults schema configuration located here. An example would be location filtering or cacheing settings.

Current Banking Backends

1. Alkami
2. Axos
3. Lumin
4. MX
5. Q2
6. Q2-MX
7. Yodlee
8. Dummy (testing)

Banking Backend Adapters

1. AFX
2. Axos-Omni
3. Lumin
4. cufx
5. md-nexus
6. md-sso

Feature

Optional Schema Configuration.

- Campaign onboarding
  - Abe AI Campaign Onboarding settles
- Location files
  - Abe AI hosted location files.
- Locator search
  - Locator Search to gather locations for location based intents.
- Node short url
  - URL Shortener Configuration
  - Supports Firebase Short URL Provider
- Yext locations
  - Use the Yext API to gather locations for location based intents.

Journeys

Journeys provide the ability to encapsulate sets of utterances as steps in a guided conversation flow to facilitate a specific goal, such as sending money from one account to another, or paying a bill.

Feature Flag Definitions For Money Movement

- Balance Check
- Bill Pay
- Money Movement Types
- Transfer Toggler

Locations

This feature helps the users find Bank information in their proximity. These are defined in JSON files for each of the banks. Information includes but not limited to ATM locations, coordinates, address, hours, and contact information.

- Banner Bank
- Connexus
- Fairwinds
- First-Republic Bank
- Iberia
- Mercantile
- Presto
- SSP
- STCU

- Trailhead
- UCCU
- Umpqua
- USBank
- Valley CU
- White River

User Manager

Overview

User Manager is a simple microservice that stores end user account data as well as end user client platform data. These records are required to identify the end user when a request is made from any client/device where that user has completed the account linking flow (@see Auth Web Client). Two main database records are managed by User Manager; Channel Identity records and User records .

Service Dependencies

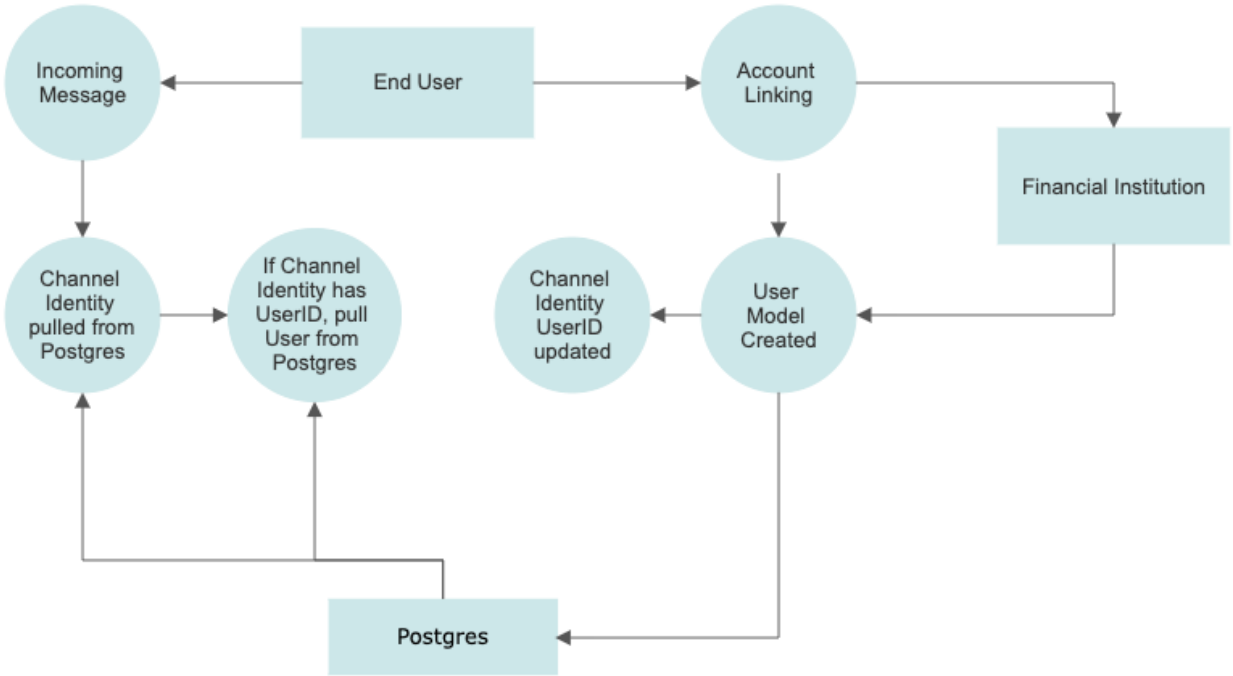
- PostgreSQL: for storing records
- Redis: for session data

Table of Content

User Model

A User Model is created through Account Linking. A User Model is a database record the User Manager tracks for an authenticated user. A User Model in addition to a Channel Identity makes up an authenticated user while a Channel Identity alone can act as a stand in for an unauthenticated user.

The relationship between User Models and Channel Identities is best understood thinking of the User Model as a record of the authenticated user in the context of a Financial Institution, while a Channel Identity is that user interfacing with in a particular Channel. If an authenticated user sends messages via SMS, that creates a Channel Identity, and if he messages via Embedded Web, another Channel Identity is created. Both Channel Identities use the same User Model when authenticated.



The User Model is made of a few core components.

First Name / Last Name (encrypted)

The First and Last name of a user to be used in conversation. The full name of a user is considered personally identifiable and is encrypted.

Source ID

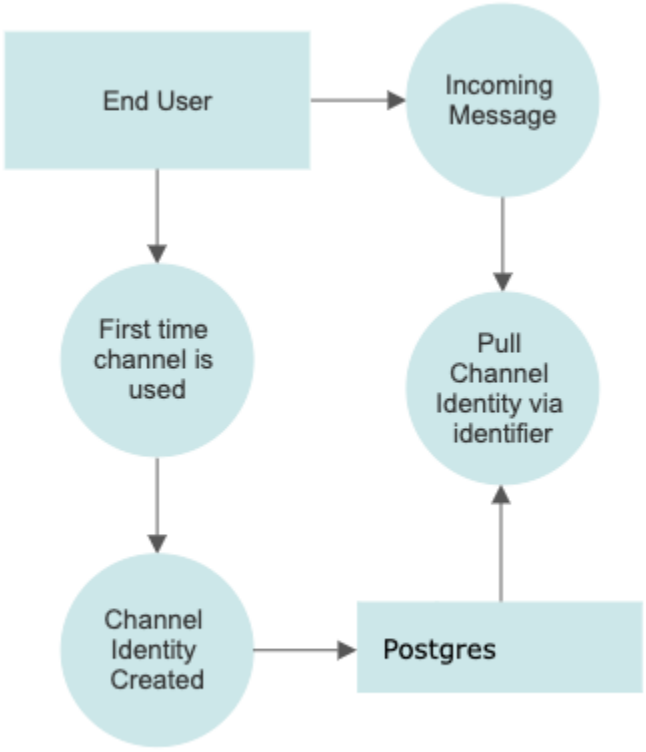
An authenticated users ID in relation to the Financial Institution. This can be thought of as the unique ID that the Financial Institution uses to track the user.

Source Data (encrypted)

The data given to us by the Financial Institution for their authentication.

Channel Identity

A Channel Identity is how the User Manager tracks a user relative to the channel they are interfacing with.
All users have a Channel Identity, while only authenticated users have a User Model.



The core of a Channel Identity comes down to a few components

User Id

The ID identifying the User Model associated to the Channel Identity, if one exists.

Channel

A string representation of the type of channel the user is interfacing with. This is important for any channel specific functionality as well as looking up channel specific responses from the Agent's dialogue configuration.

Identifier (encrypted)

A Channel Identities Identifier is a string representing a key unique to a Channel.

User Data (encrypted)

The User Data of a Channel Identity stores information about the channel such as the Agent ID of the Agent the Channel Identity is engaged with, and the Identifier and Channel key, as well as information about the User Model if the user is authenticated.

NLU

Python Services Natural Language Processsing

Features

Agents within Conversational AI have additional functionality attached to them that does not directly fall into the generic End User Request flow. For example, and end user can begin a conversation with a human customer support representative through an Agent via Human Handoff.

The included feature guides will provide an overview of each set of functionality as well as references to any necessary management and configuration information.

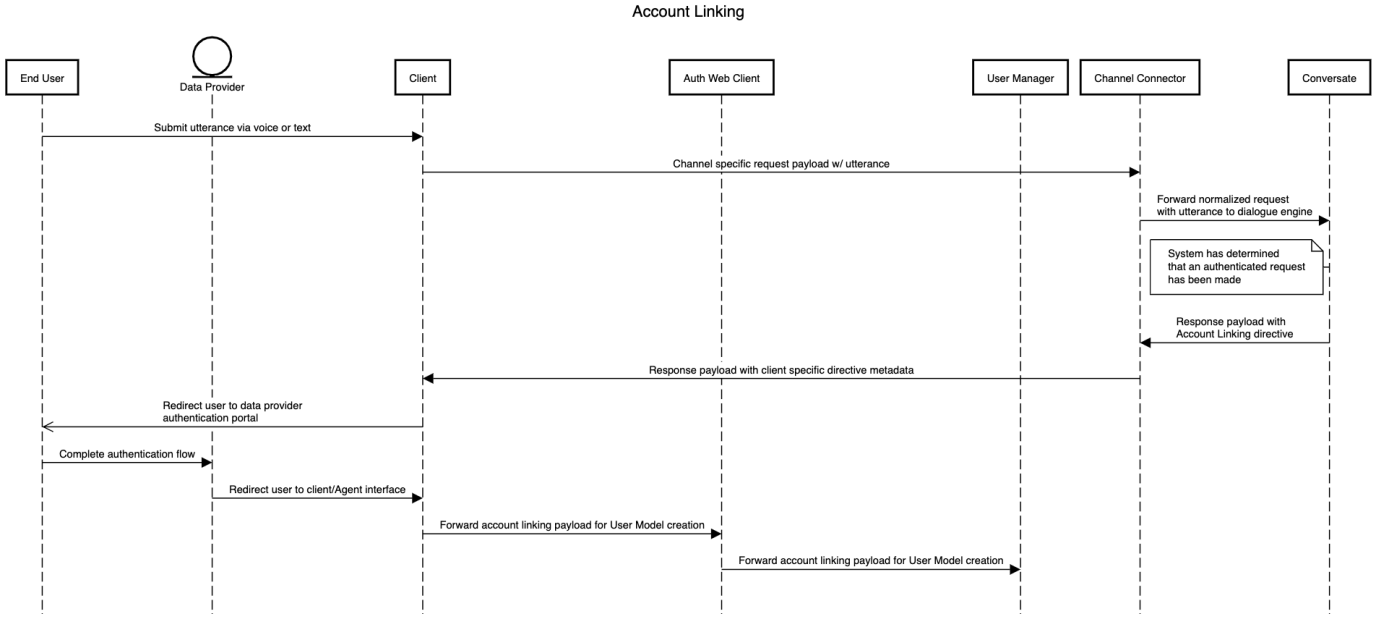
Table of Contents

Account Linking

End users will have some kind of digital representation within a data provider's system that is used to uniquely identify them. Additionally, data providers expose authenticated gateways to gather data related to these identities. The goal of account linking is to ascertain the unique identity of the end user from the data provider, and to get credentials from the data provider that allows the Agent to gather end user's data from the provider.

When an end user invokes an intent that requires access to provider data, and no credentials or identity exists for that provider's user representation (a User Model) an Account Linking directive will be passed to the end user's client. The client will then redirect the user to the data provider's authentication portal.

The user will be redirected back to the Agent's conversational flow upon completion of the provider's authentication flow, and a set of provider credentials will be given to the Agent.



Pre-Authentication

Overview

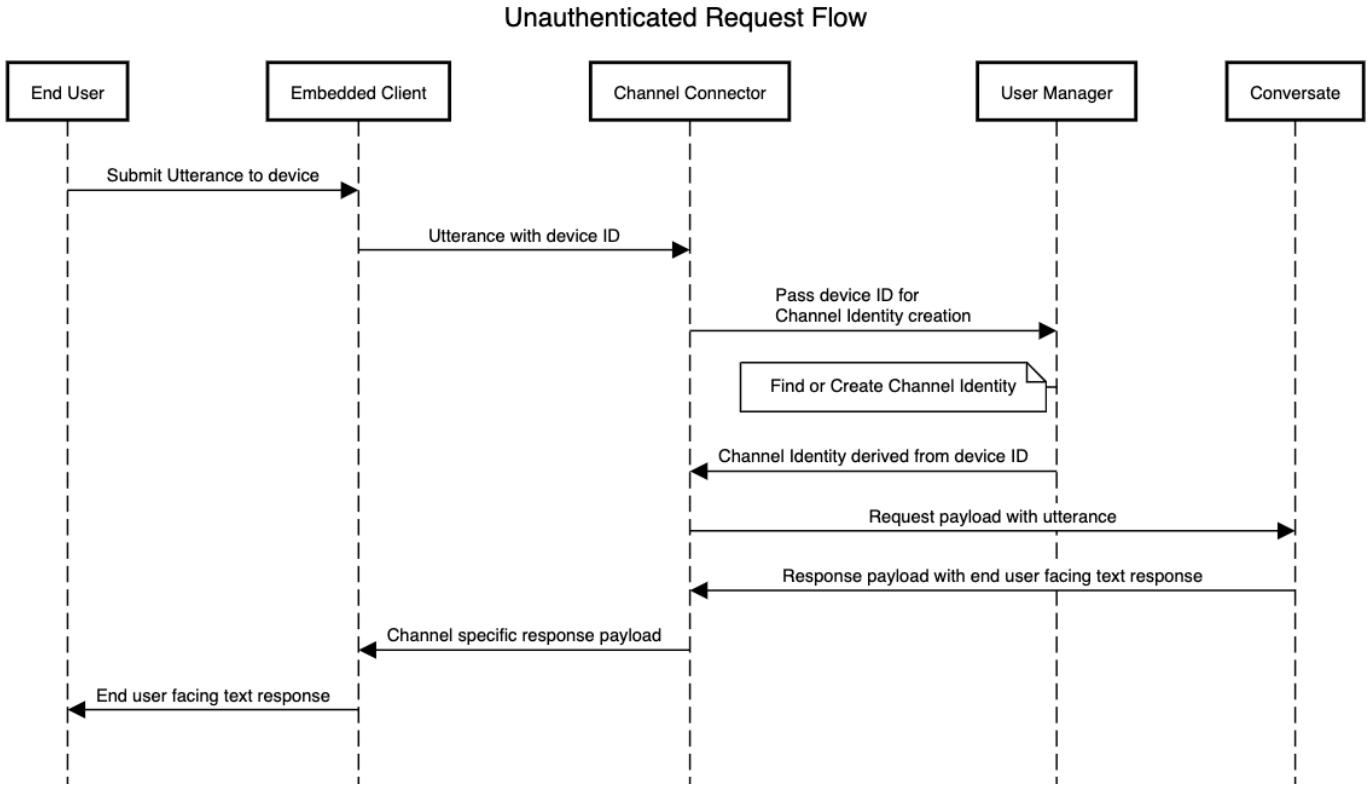
End users may communicate with an Agent using a client that directly integrates with the primary Agent data provider. For example, a Financial Institution(FI) may have their own customer support mobile app, or website, that their patrons utilize to manage their finances.
When an end user is utilizing a client that is owned by an Agent data provider they are generally already authenticated with the provider client, so it is not necessary for an end user to authenticate again on an Agent's behalf. Rather, the client can directly forward an end user's provider authorization credentials directly to an Agent with outgoing utterances. This process is known as "Pre Authentication".

Request Flow

This flow is similar to Account Linking since they both result in authorization credential gathering and User Model creation, but the Pre Authentication flow does not require the user to perform an authentication step as they do with Account Linking.

Unauthenticated Request Flow

This is an example flow with no user credentials being passed by the embedded client. In this case no User Model is created, and no authenticated requests can be performed by the end user.

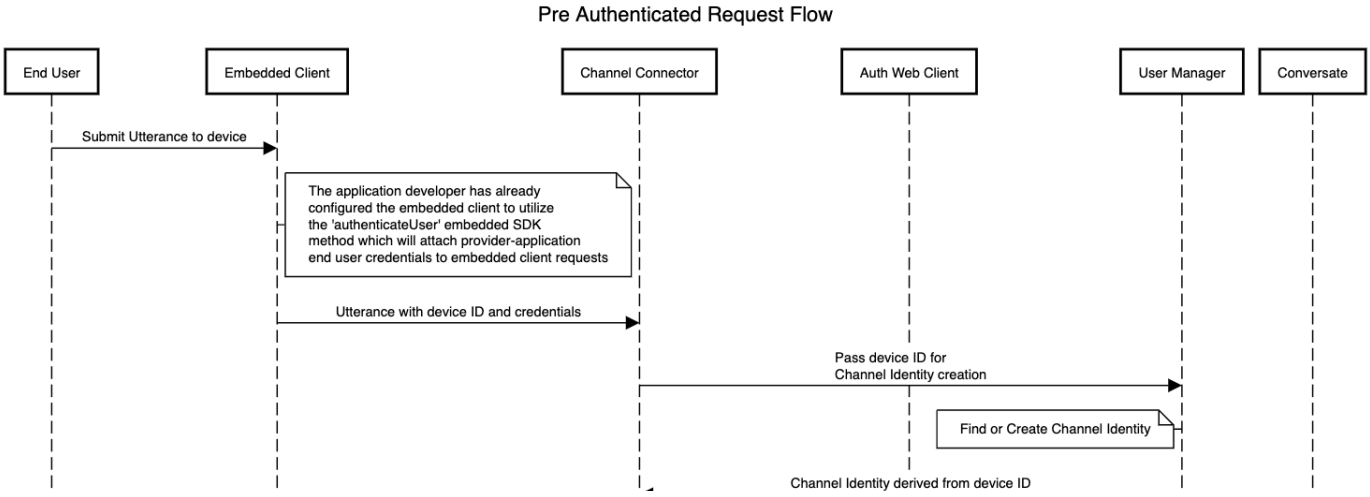


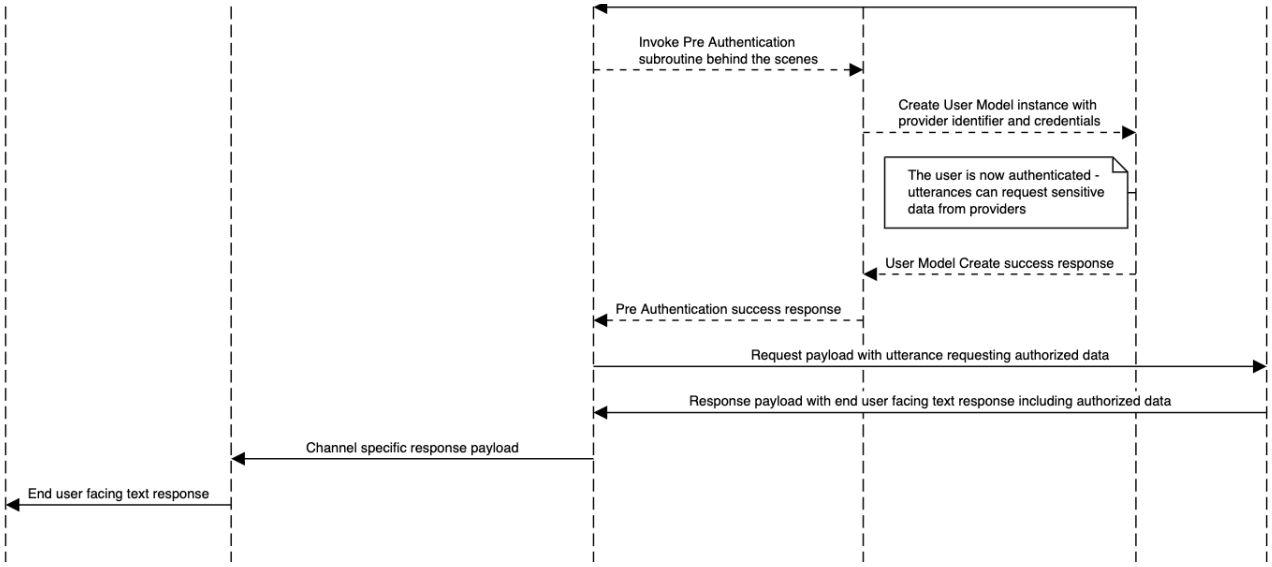
Pre Authenticated Request Flow

When embedded channels are integrated with an existing FI application the integration developer may choose to utilize the authenticateUser method of the embedded SDK. This SDK method takes authenticated user identifiers and credentials as arguments, and passes those values along with subsequent requests in the End User Request flow.

By attaching the end user's credentials to subsequent payloads via the authenticateUser method we now invoke the pre authentication subroutine on subsequent embedded client requests. When this subroutine is invoked a User Model will be created before facilitating the end user's request.

This series of events provides the Agent with necessary credentials (derived from the embedded client) and it can now gather authenticated data on behalf of the end user.





Once an end user has completed a Pre Authentication flow the Conversational AI Agent is capable of making authorized requests on behalf of the end user. Additionally, a User Model will be created and associated with the Clients derived Channel Identity.

Human Handoff

During the flow of Conversations an end user may need to discuss specific or complex issues that are better handled by a human operator rather than utilizing the standard AI driven End User Request flow for responses.
When a need for a human operator arises the conversation flow can be redirected to an asynchronous subroutine which relays end user utterances directly to a human operator, and to relay the operators responses directly back to the end user. This process is known as “human handoff”.

Requirements

An organization operating an Agent must have some form of a Human Handoff Provider. Generally this would be a Customer Relationship Management (CRM) platform such as Salesforce. The Conversational AI infrastructure currently supports Salesforce, Glia, and Twilio Flex. The configuration of these providers can be performed via CUI within the Dialogue Settings Agent Global Variables panel.

Human Handoff Intent Flow

Human Handoff is achieved by a series of intents linking to each other. When one intent step is fulfilled it will navigate to the next intent until a Human Handoff Session is established with the Human Handoff Provider, or an error occurs. A human handoff flow can be defined by a certain set of states it can be in depending on the health of the Human Handoff Provider session. The various states a human handoff session can be in are represented below.

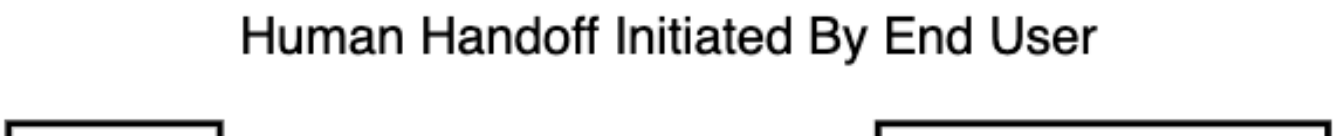
Once a Human Handoff is established a Human Handoff Session Consumer will take over the conversation flow for all end user conversations until the session is terminated by the end user, the human operator, or a session timeout.

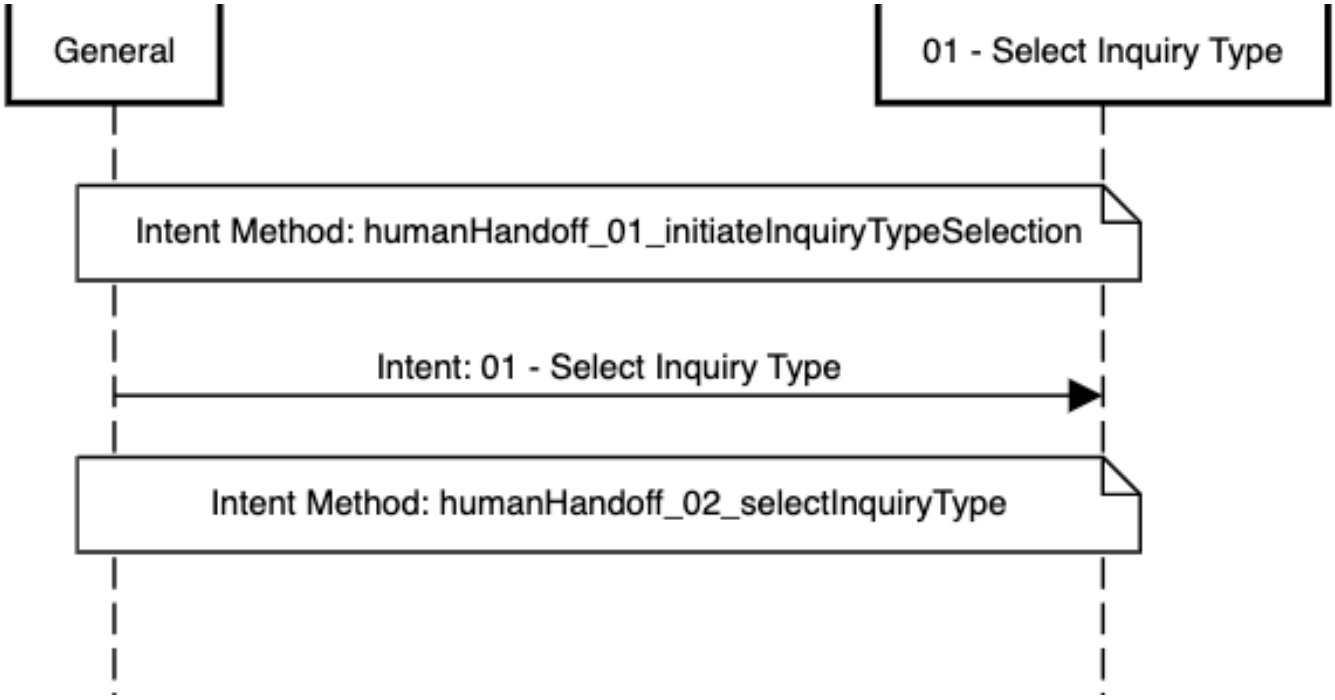
The Human Handoff Session Consumer will receive all incoming end user utterances from Channel Connector via Conversate and redirected to the human operator fielding the session. Additionally, each Human Handoff Provider will relay messages from the human operator back to Channel Connector in their own unique way. Each providers communications methods is detailed in the Human Handoff Session Consumer provider section.

All of the dialogue options you are going to see are variable and configurable in CUI through the Dialogue Explorer

In Human Handoff, the first step is when the end user asks to talk to a human triggering the `humanHandoff_01_initiateInquiryTypeSelection` Intent Method.

> *Ok, let me connect you to an Team Member. Which of the following would you like to talk about?*
>
> *1. Checking, Savings, or other deposit accounts*
>
> *2. Existing Mortgage, Auto or other loans*
>
> *You can also say "nevermind" if you don't want to talk to a team member.*





If initialization fails we move into state `02 - Session Initialization` and Intent Method `humanHandoff_03_sessionInitialization Failed` is triggered. A different response is given for if there is no team member available

> *No {{agent.fiNameShort}} team members are online at the moment. Please try again later.*

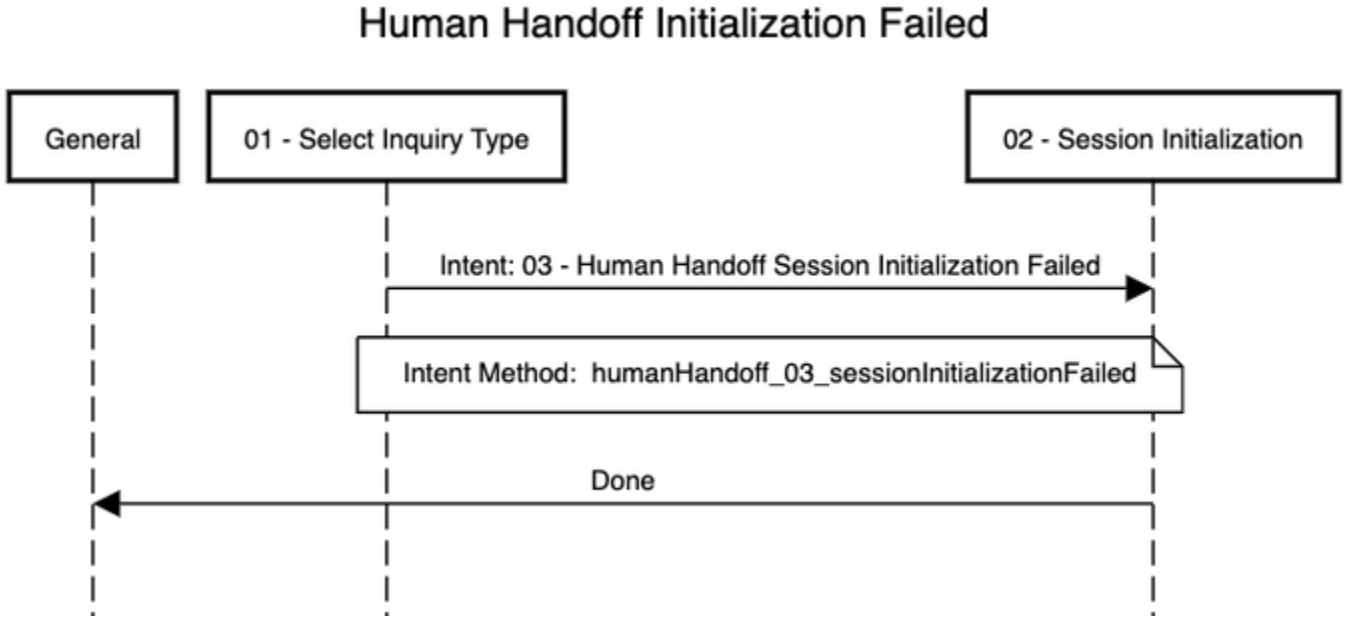
If the time is outside business hours

> *Our chat team is available {{channelSpecific agent.chatSupportHours}}, except federal bank holidays. For immediate assistance you can reach out to us at {{channelSpecific agent.phoneSupportGeneral}}.*
>
> *You can also send us a secure message within online banking by choosing "Messages" at the top of your dashboard.*

Or if an Error occurs

> *Something went wrong while attempting to connect you with a team member. Please try again.*

Since initialization failed, we return to state `General`.

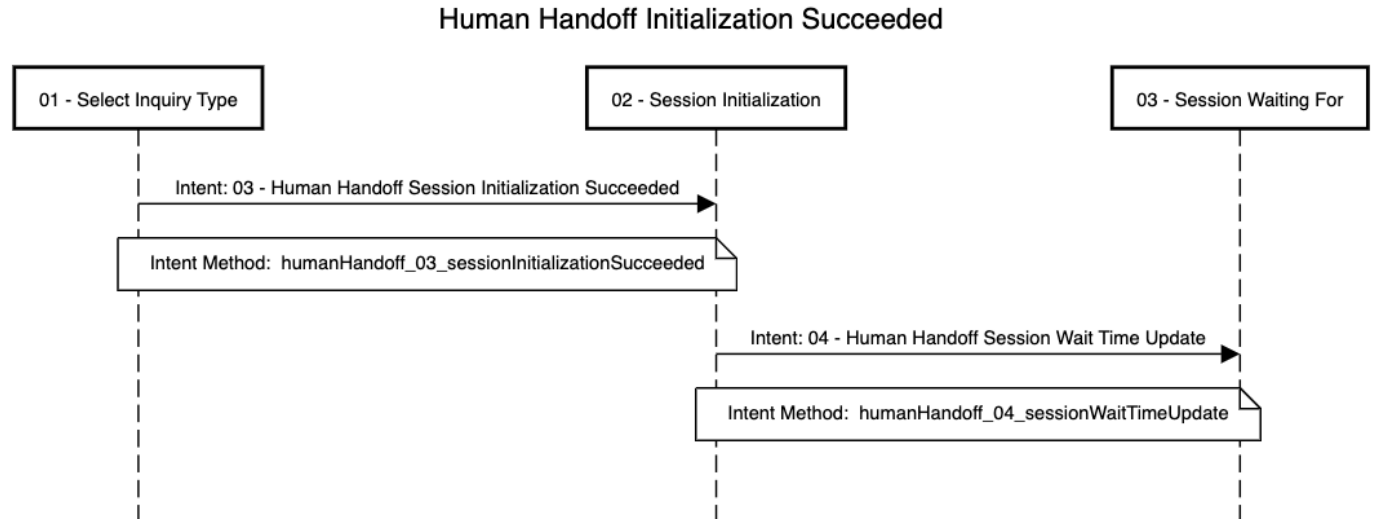


If initialization Succeeded, we move into state `02 - Session Initialization` and trigger Intent Method `humanHandoff_03_sessionInitializationSucceeded`.

>

> *You are {{ordinalize queuePosition}} in the support queue, and your wait time is approximately {{humanizeDuration estimatedWaitTime 'seconds' false}}.*
>
> *You can stop waiting at any time by texting "leave queue".*

As the text suggests, at this point we are in queue to talk to a Human Agent. We move into state `03 - Session Waiting For` and trigger Intent Method `humanHandoff_04_sessionWaitTimeUpdate`



While the End User is in Queue, they may cancel their Human Handoff request. If they do, the intent method `humanHandoff_05_cancelSessionWaiting` is triggered, and we move back to state `General`.
First, the End User is asked to confirm

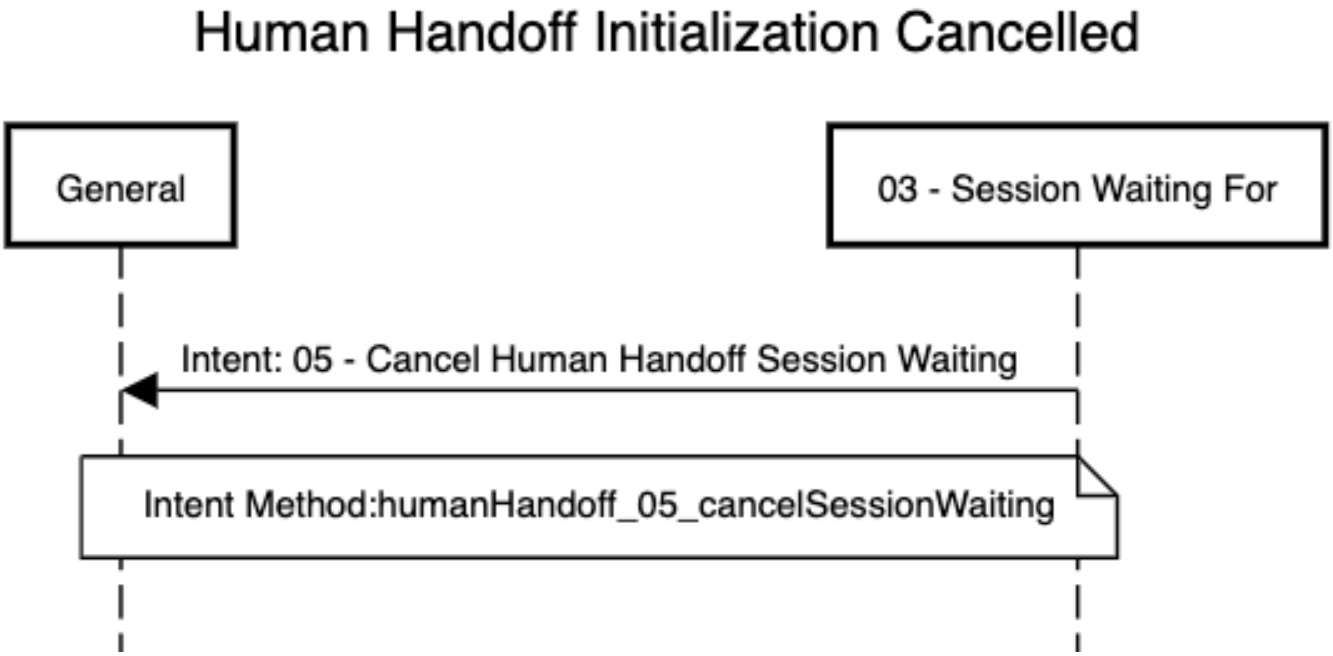
> *Ok. To confirm, you'd like to leave the support queue? Is that right?*
>
> *You are {{ordinalize queuePosition}} in the support queue, and your wait time is approximately {{humanizeDuration estimatedWaitTime 'seconds' false}}.*

And once confirmed

> *Ok, you left the support queue. Reconnecting you to {{agent.name}}.*

Or if the request to cancel is canceled,

> *Ok, you're still in line.*
>
> *You are {{ordinalize queuePosition}} in the support queue, and your wait time is approximately {{humanizeDuration estimatedWaitTime 'seconds' false}}.*



Once the Queue has completed, the End User has waited long enough and the session is ready to be Established. The Intent Method `humanHandoff_06_sessionEstablished` is triggered and we move to state `04 - Session Active`.

> *You're now chatting with {{humanAgent.name}} from {{agent.fiNameShort}}. You can end this chat at any time by texting "end chat". Otherwise the team member will end the chat once your conversation is complete.*

After the End User has had enough with the Human Agent, and either the Human Agent or the End User triggers the end of the Active Session, Intent Method `05 - End Human Handoff Session` is triggered

> *Your chat with {{humanAgent.name}} has ended. Reconnecting you to {{agent.name}}.*

If the End User is the one ending the engagement, they will be asked to confirm

> *Ok. To confirm, you'd like to end the chat with {{humanAgent.name}} from {{agent.fiNameShort}}. Is that right?*

And if they decide to continue talking to the Human Agent

> *Ok. You're still chatting with {{humanAgent.name}} from {{agent.fiNameShort}} team member.*

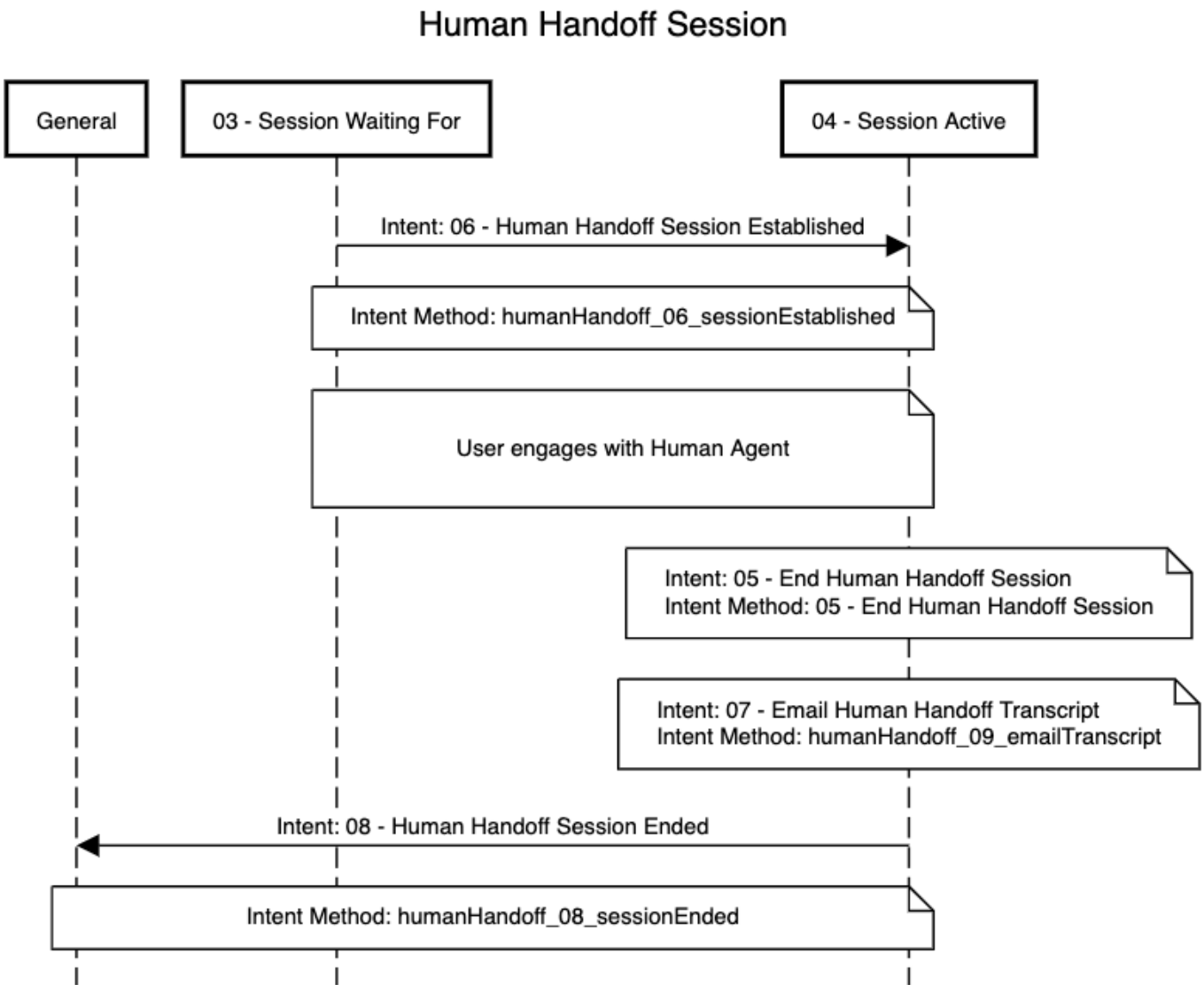
After the Engagement has been ended, the Intent Method `humanHandoff_09_emailTranscript` is triggered.

> *Hi again. This is {{agent.name}}. Would you like me to email a transcript of your chat with {{humanAgent.name}}?*

if so

> *Ok. Your transcript will be sent to you.*

Finally, we are returned to state `General` with Intent Method `humanHandoff_08_sessionEnded`



Job Broker

The Job Broker is the service that publishes Jobs to be consumed by the Human Handoff Session Consumer.
The Intent Method humanHandoff_02_selectInquiryType makes a call to the Job Broker to initiate a Job for Human Handoff.
The Job Broker supports various backends for managing Jobs

- Redis
- AWS Lambda
- Amqp
- Sqs

Human Handoff Session Consumer

The Human Handoff Session Consumer is a serverless application posed to initiate and maintain a long running human-handoff session. The Consumer's lifecycle is as follows:

- Receive @abeai/job-broker job on specified human-handoff queue.
- Select human-handoff Provider backend.
- Build/parse human-handoff session metadata.
- Initialize human-handoff session.
- initialize human-handoff session state.
- Initialize main loop:
  - Poll for new messages.
  - Warn/disconnect idle users.
  - Handle session disconnection/reconnection.
  - Handle session rollover.

The Human Handoff Session Consumer leverages Provider backends.

- Salesforce
  The Salesforce integration is a long polling driven architecture. In the Main Loop the Salesforce API is leveraged to pull new messages, and update the Salesforce session state. For instance, flagging the user as Idle or Active.
- Glia
  The Glia integration is a webhook driven architecture that requires passing an endpoint (the Channel-Connector service) when initializing a new human-handoff session with Glia.

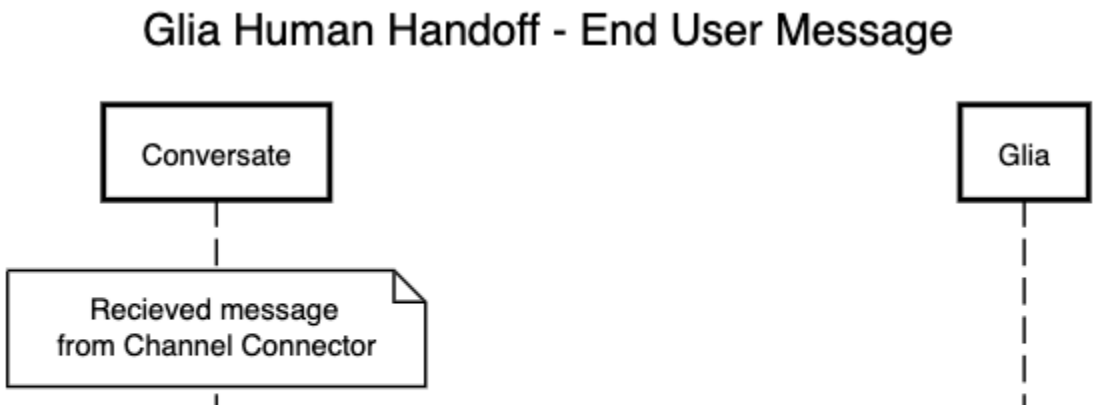
  Rather than polling for new messages on every Main Loop iteration, new message events are passed to the provider backend via subscription to the Platform Event Bus.
- Flex (Contact Center Provider)
  The Flex integration leverages the Twilio-chat suite. Webhooks are used to read message events similarly to the Glia implementation, an endpoint to Channel-Connector is given to Twilio and messages are handled through the Platform Event Bus.
  Unlike the Glia implementation, the Twilio-chat suite leverages a web socket connection with event listeners for events. Users entering and exiting the chat, and the typing indicator are handled this way.

Glia - Human Handoff

Glia is a Human Handoff provider leveraged by the Human Handoff Session Consumer

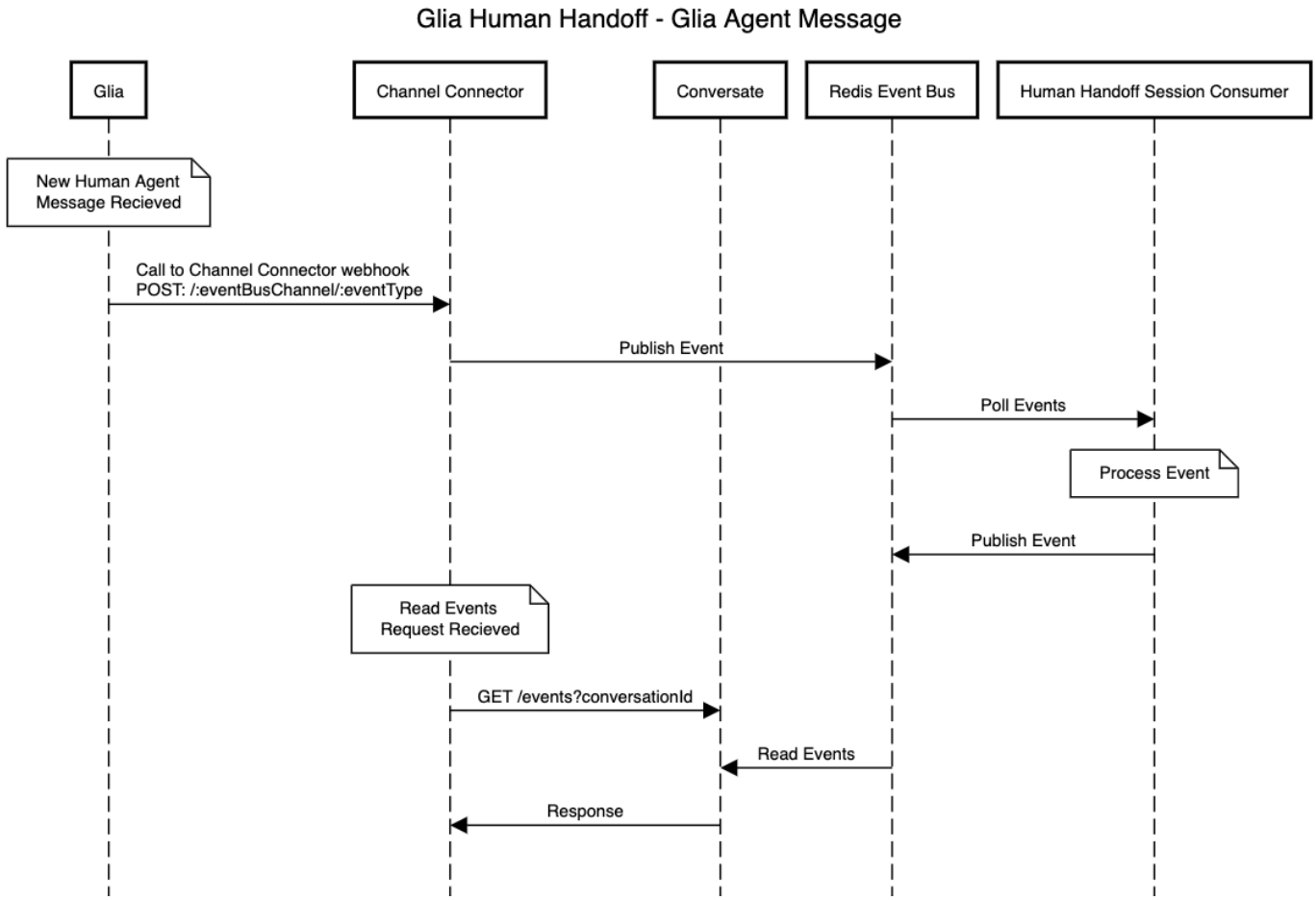
Message Flow

Incoming messages from the End User come through the Channel Connector just as any message would. When a Human Handoff Session is active in Conversate messages are passed along to the Human Handoff Provider Backend.



Incoming messages from the Human Agent are sent to an Events Webhook in Channel Connector.

Channel Connector will post these Events to an Event Bus. The Human Handoff Session Consumer continuously polls events from the Event Bus, processes them and updates the Human Handoff session as needed. Then the Human Handoff Session Consumer will publish an Event to Conversate to be read with Channel Connector's `GET /events?conversationId`.



All events in the system will follow the same pipeline. Glia will POST events to our Channel Connector webhook, and the Human Handoff Session Consumer will process them for Conversate .

These Events include:

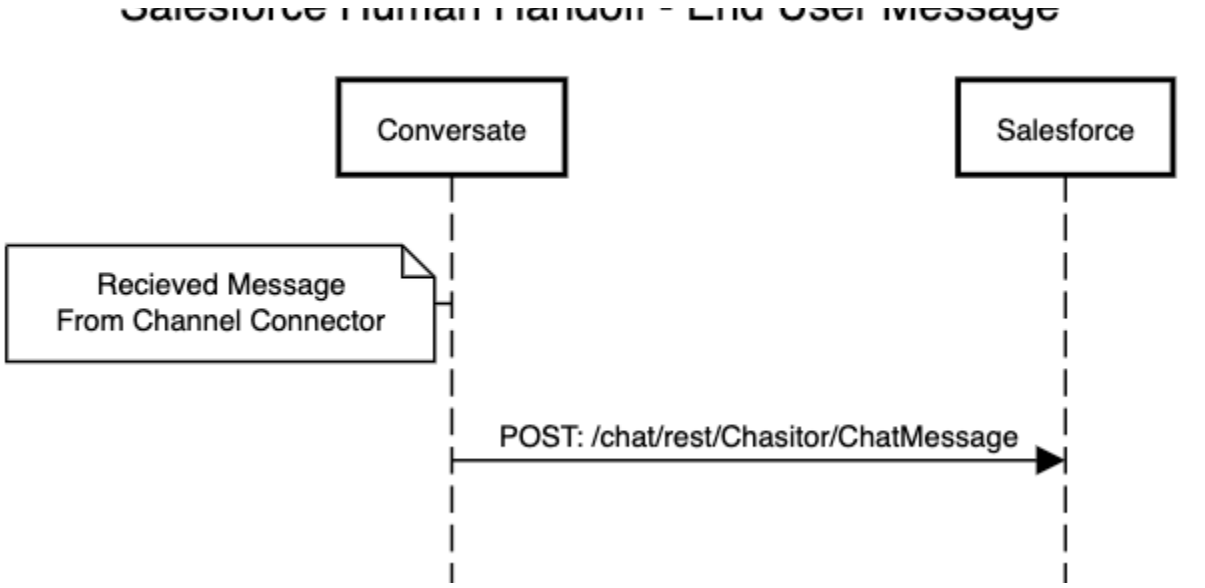
- `SYSTEM_ENGAGEMENT_CREATED` Used for handling Human Handoff states. The engagement has been initialized.
- `ENGAGEMENT_START` Used for handling Human Handoff states. The Human Agent has joined, and the Engagement has begun.
- `ENGAGEMENT_CHAT_MESSAGE` A new message from the Human Agent
- `ENGAGEMENT_CHAT_TYPING_INDICATOR_OPERATOR` For updating the typing indicator for the Human Agent
- `ENGAGEMENT_END` Used for handling Human Handoff states. Either the End User or the Human Agent have left, the Engagement is over.

Salesforce - Human Handoff

Salesforce is a Human Handoff provider leveraged by the Human Handoff Session Consumer

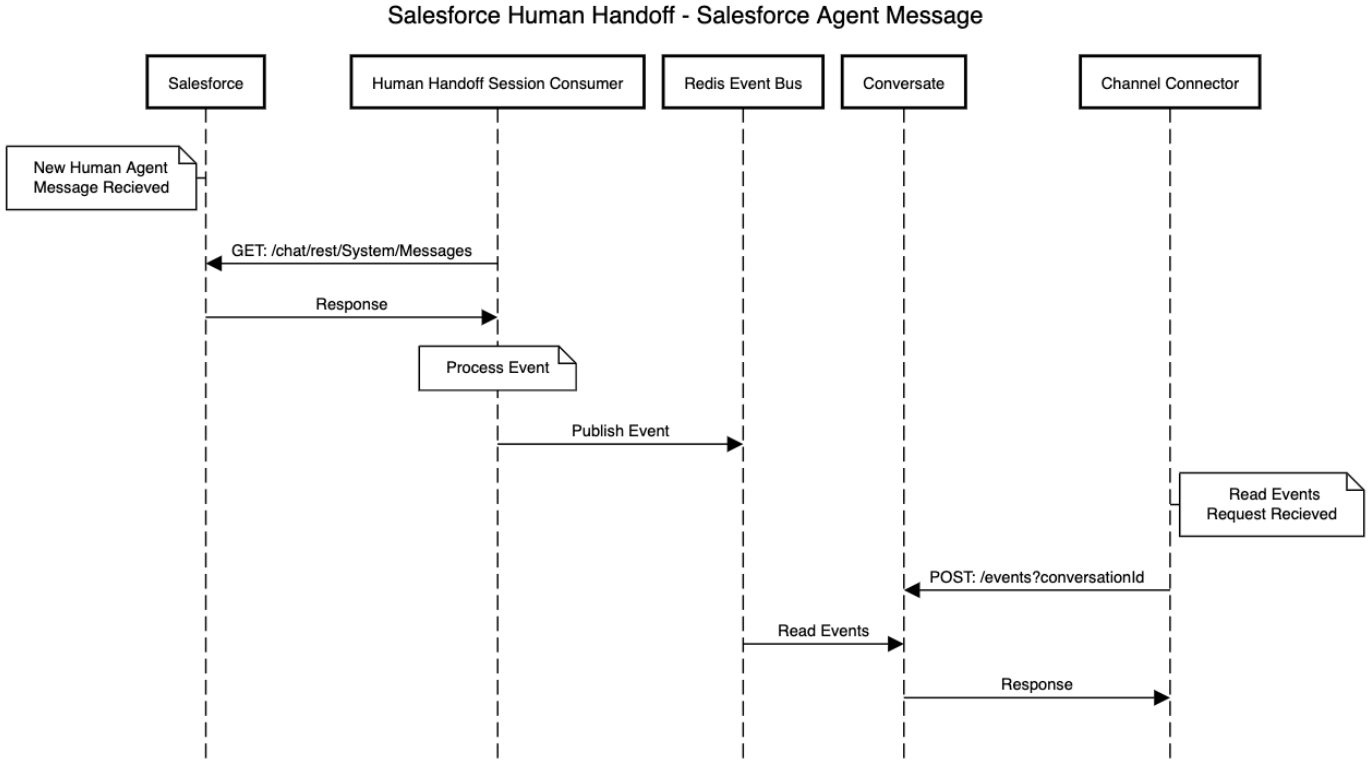
Message Flow

Incoming messages from the End User come through the Channel Connector through the main flow for an End User Request. If Conversate has a tracked Human Handoff Session for the related conversation ID, then we depart from the flow of the End User Request and leverage a Salesforce backend in Conversate



The Human Handoff Session Consumer leverages long polling to find message events. The Human Agent will post messages to be stored by Salesforce. The Human Handoff Session Consumer will fetch the messages via a Rest Endpoint `GET: /chat/rest/Chasitor/ChatMessage` . After the events are processed and the Human Handoff Session is updated, the Events are published to the Redis Event Bus.

Embedded Web reaches out to Channel Connector which reaches out to Conversate for Event Records.



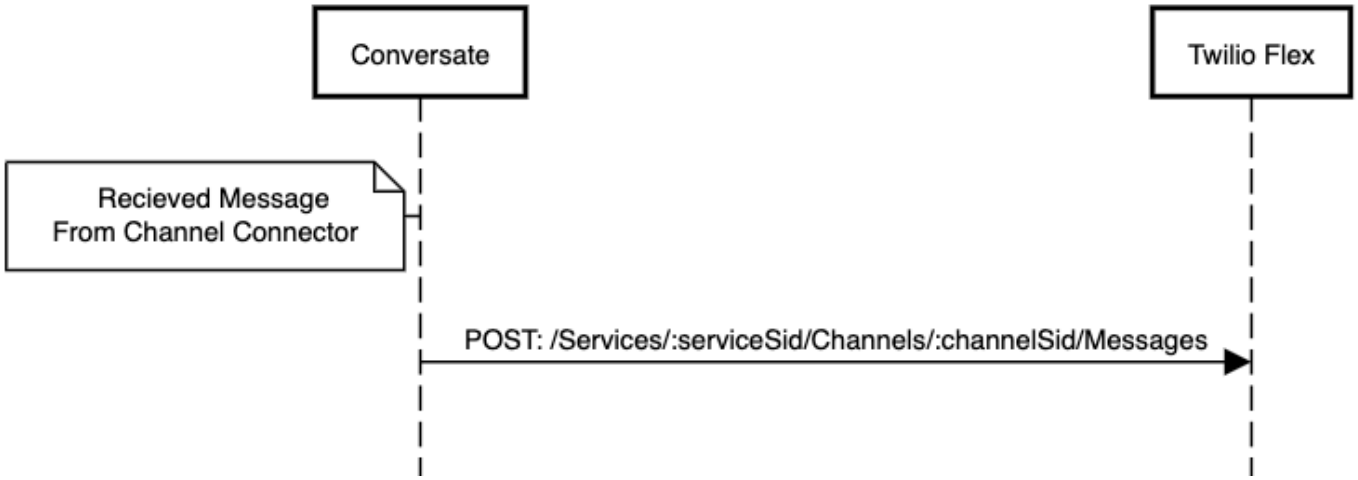
Twilio Flex - Human Handoff

Twilio Flex is a Human Handoff provider leveraged by the Human Handoff Session Consumer. Twilio Flex Human Handoff facilitates Human Handoff with the Contact Center.

Message Flow

Incoming messages from the End User come through the Channel Connector through the main flow for an End User Request. If Conversate has a tracked Human Handoff Session for the related conversation ID, then we depart from the flow of the End User Request and leverage a Twilio Flex backend in Conversate to facilitate messages reaching the Contact Center.

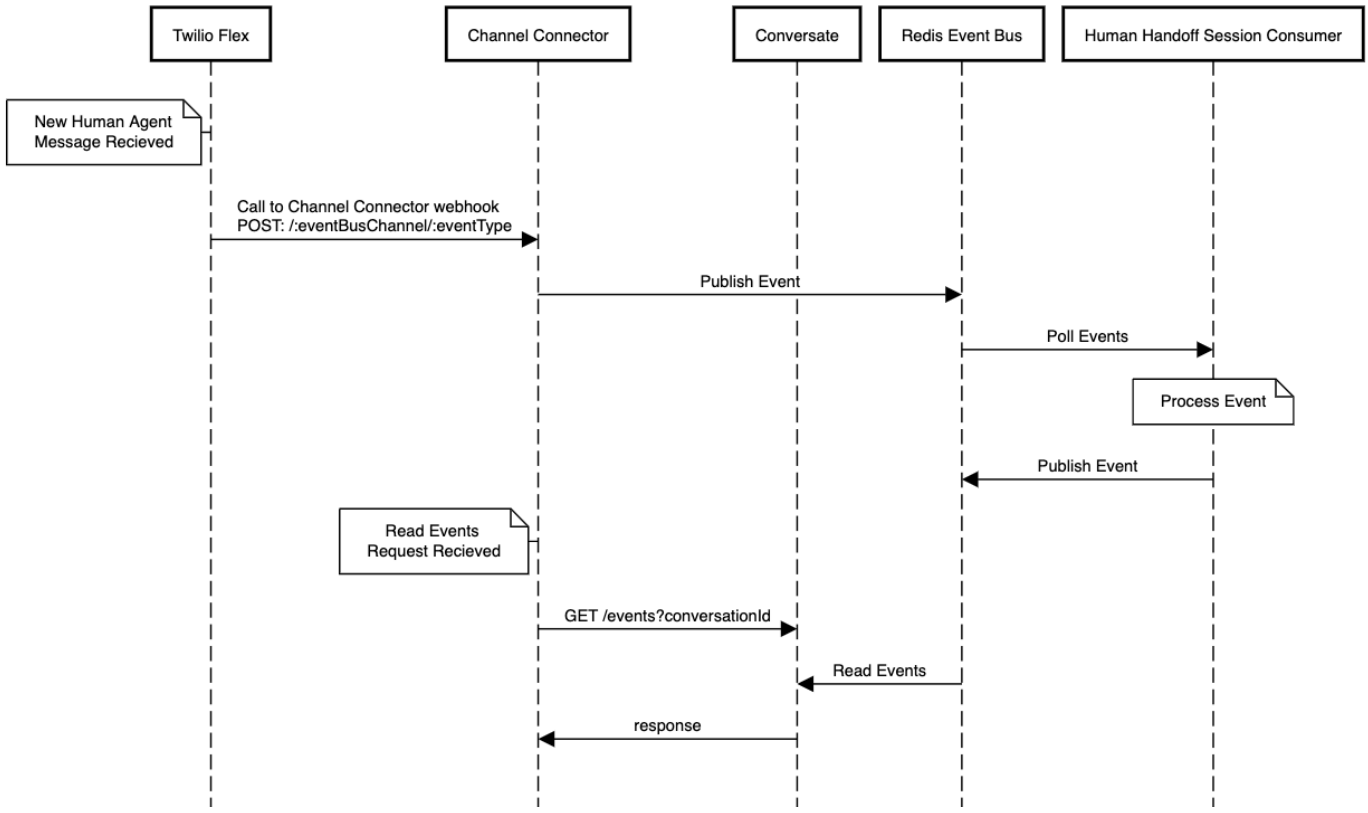
Twilio Flex Human Handoff - End User Message



Incoming messages from the Human Agent are sent to an Events Webhook in Channel Connector.

Channel Connector will post these Events to an Event Bus. The Human Handoff Session Consumer continuously polls events from the Event Bus, processes them and updates the Human Handoff session as needed. Then the Human Handoff Session Consumer will publish an Event to Conversate to be read with Channel Connector 's `GET /events?conversationId`.

Twilio Flex Human Handoff - Contact Center Agent Message



Additional Events

In addition of the incoming messages, other events come in from Twilio Flex. While the messages from the Contact Center agent come in through the Channel Connector Webhook, other events come through the Twilio-Chat SDK through web socket event listeners.

These events include:

- `channelUpdated`. When the channel is updated, we check if its status is `INACTIVE` and if it is, we close the session.
- `typingStarted`. Used for updating the Typing Indicator.
- `typingEnded`. Used for updating the Typing Indicator.

Journeys

Overview

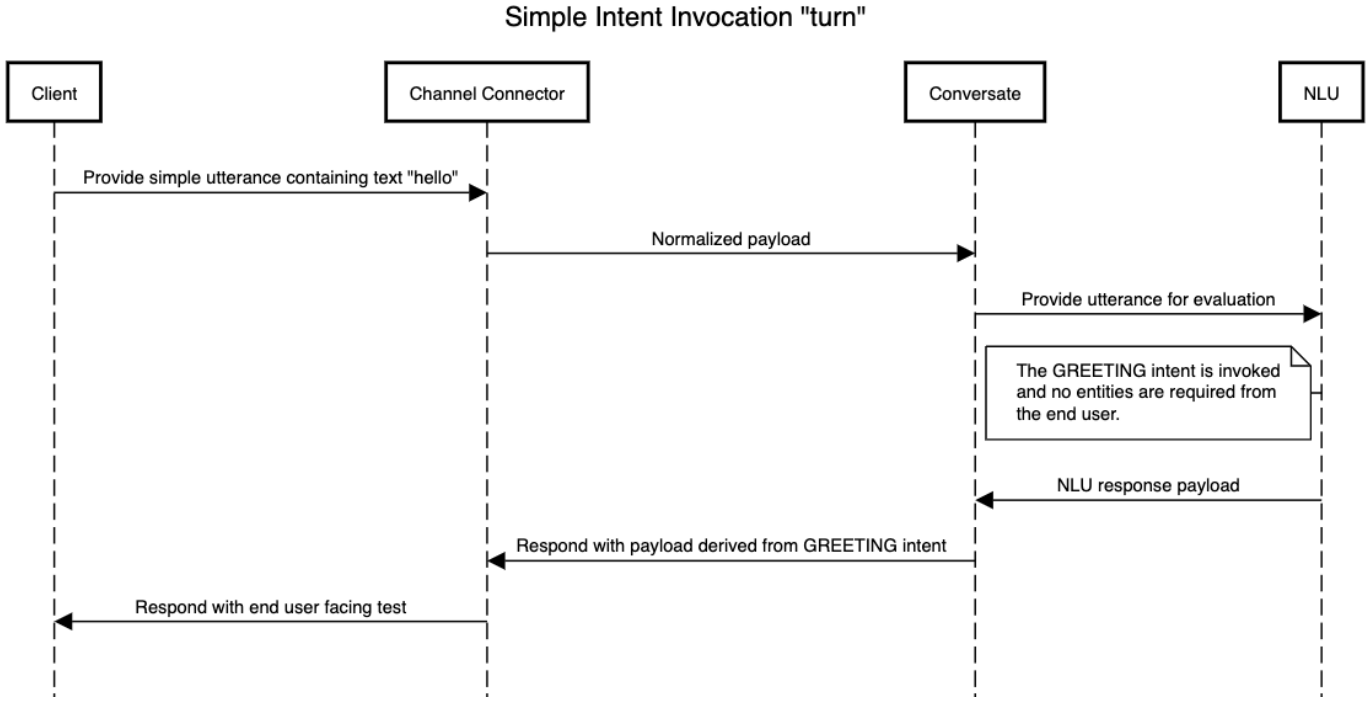
Journeys provide the ability to encapsulate sets of utterances as steps in a guided conversation flow to facilitate a specific goal, such as sending money from one account to another, or paying a bill.

The need for this functionality is clear when we consider that the End User Request flow is a linear exchange between an end user client and the Agent (powered by Conversate and the dialogue engine). A complete cycle using this request/response flow can only facilitate queries for individual responses, or entities if the Agent fielding the conversation is in gather mode.

Linear Intent Invocations

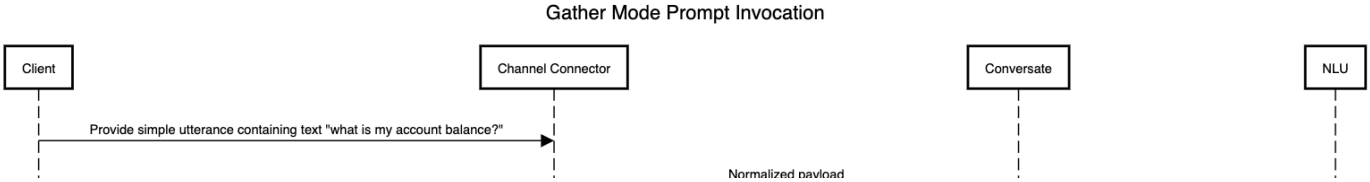
Simple Intent Invocation

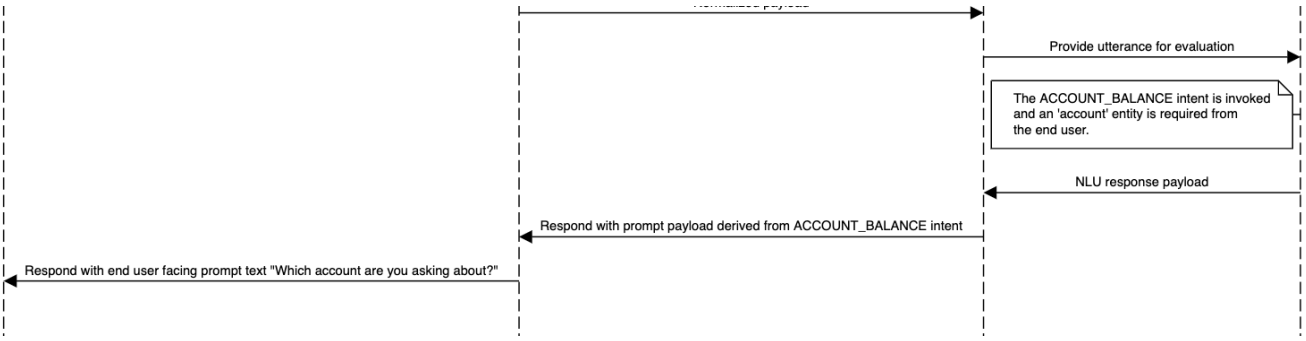
If an end user requests provides the utterance “hello” an Agent would simply respond with response text derived from an intent that is trained on similar utterance data to “hello”, and the request/response cycle would cease at that point.



Simple Intent Prompt (Gather Mode)

Similarly, if an end user said “What is my account balance?” the Agent would determine that an account entity is required from the user (the account they want a balance for) and put itself into gather mode. The Agent would then prompt an account entity from the user via gather mode.





Any subsequent requests by the user would need to provide a valid account entity (account name, account last four digits), or the Agent would continue to prompt them for a valid entity. Just as with the first example, this is a linear exchange. Either the user gives a correct entity to facilitate the original intent invocation, or the Agent will continue to prompt the user via gather mode.

Linear Intent Invocation: Multiple Intents

Unlike the previous examples, the Journey functionality allows for complex entity gathering across multiple intents to facilitate high level functionality such as moving money between accounts.

Practical Example Thesis

Suppose an end user wanted had two accounts “Total Savings” which is their main savings account, and “Consumer Checking” which is their day to day checking account. If the end user wanted to move money from Total Savings to Consumer checking the Agent would need an assortment of entities.

- The source account: an entity representing the account the user wants to move money from.
- The destination account: an entity representing the account the user wants to move money into.
- The transfer amount: a numeric entity representing the amount of money the user wants to move from their source account to their destination account.
- The transfer date: a date entity representing the specific date the money movement should occur.
- The transfer confirmation: a final yes/no confirmation from the user that the details pertaining to the money movement are accurate.

Using a simple linear response generation pipeline as described in previous sections would not be enough to achieve this flow as it is designed merely for request/response handling. The functionality Journeys provides allows for these complex interactions.

Journey Anatomy

It makes sense that a Journey would be a set of intents invoked serially, but the arrangement of these intents is complex. At its core the Journey structure is a set of “phases” which can be viewed as individual steps in the Journey.

Using the previous money movement Journey example we can imagine money movement steps as “gather source account”, “gather destination account”, “gather amount”, “gather date”, “gather user confirmation”. While this isn’t the exact structure, the macro concepts are well represented by these hypothetical Journey phases.

These phases are defined by Journey configuration files written in JavaScript, and maintained within the Integration Services repositories as siblings of Intent Methods.

Example Journey Configuration

```
// Journey Configuration containing Journey state constructor.
{
  MOVE_MONEY_JOURNEY: function(...) {
    // Initial Journey state.
    return {
      yesNo: {}, // confirmation metadata
      cache: { ... }, // Journey metadata cache
      phases: [ // Journey phase set
        phases.userContext(initialValues),
        phases.payee(initialValues),
        phases.amount(initialValues),
```

```
        phases.additionalPrincipal(),
        phases.additionalEscrow(),
        phases.amountPastDue(),
        phases.sourceAccount(initialValues),
        phases.date(initialValues),
        phases.reviewAndConfirm(initialValues),
      ],
    };
  },
}
```

Phases: A closer look

Each phase in a Journey is comprised of a set of intents that must be fulfilled before the Journey can continue. While there is some magic that Conversate must perform to keep a conversation in the Journey state, the fulfillment of each intent in a phase of a Journey is identical to the Simple Intent Invocation examples previously provided.

Example Phase Configuration

```
// Root configuration for 'sourceAccount' phase of above Journey example
{
    displayName: 'Source Account',
    phaseName: 'SOURCE_ACCOUNT',
    intents: [ // Intent set that must be fulfilled to complete phase.
      {
        intent: 'SOURCE_ACCOUNT',
        parameters: {
          sourceAccount: initialValues.sourceAccount || null,
        },
      },
    ]
}
```

Given this explanation, a Journey can be viewed as a set of intents broken down into steps (phases) that must be invoked in sequence. If any intent is not fulfilled, or if the end user invokes a 'cancel' intent, the Journey will be terminated, and the conversation will be put back into the 'general' state and consume incoming requests following the End User Request flow.

The practical application of Journeys is generally to gather sets of entities for a greater cause. Using the example from previous sections, a set of details required to necessitate the movement of money from one account to another.

While it may be clear, it's worth pointing out that any intent methods being utilized by the phases of a Journey must be fully defined and implemented within the Integration Service being used to facilitate the Journey.

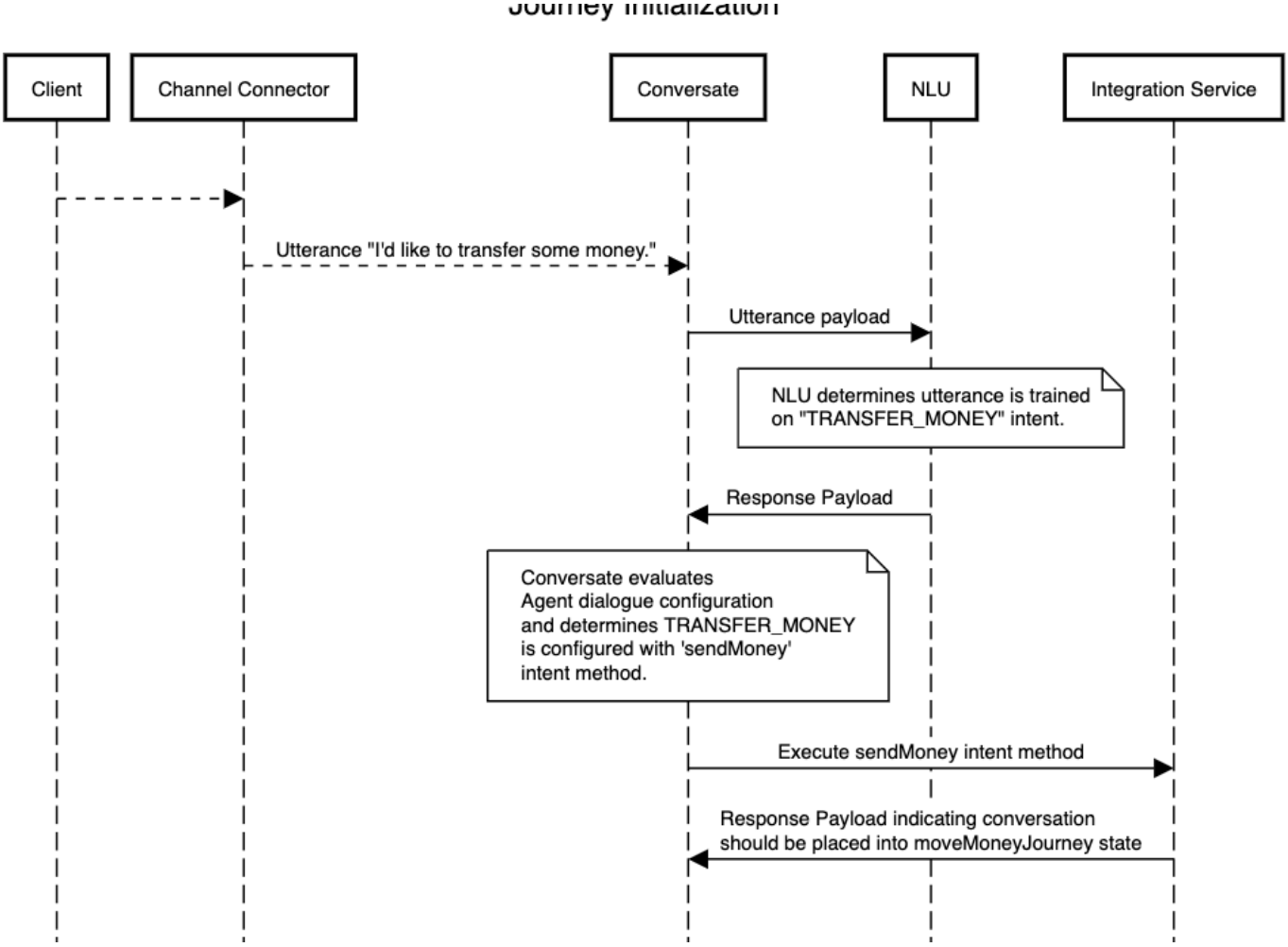
Happy Path Journey Flow

Let's summarize our understanding of a Journey by observing the flow of a user through our example money movement Journey.

This is a "happy path" execution path meaning it is assumed that when an entity is required from a user (source/destination accounts, amount, date, confirmation) it is correctly given during the Agent's gather mode flow. Additionally, we will omit the phases "additionalPrincipal", "additionalEscrow", and "amountPastDue" since those deal with special edge cases for involved accounts.

Journey Initialization

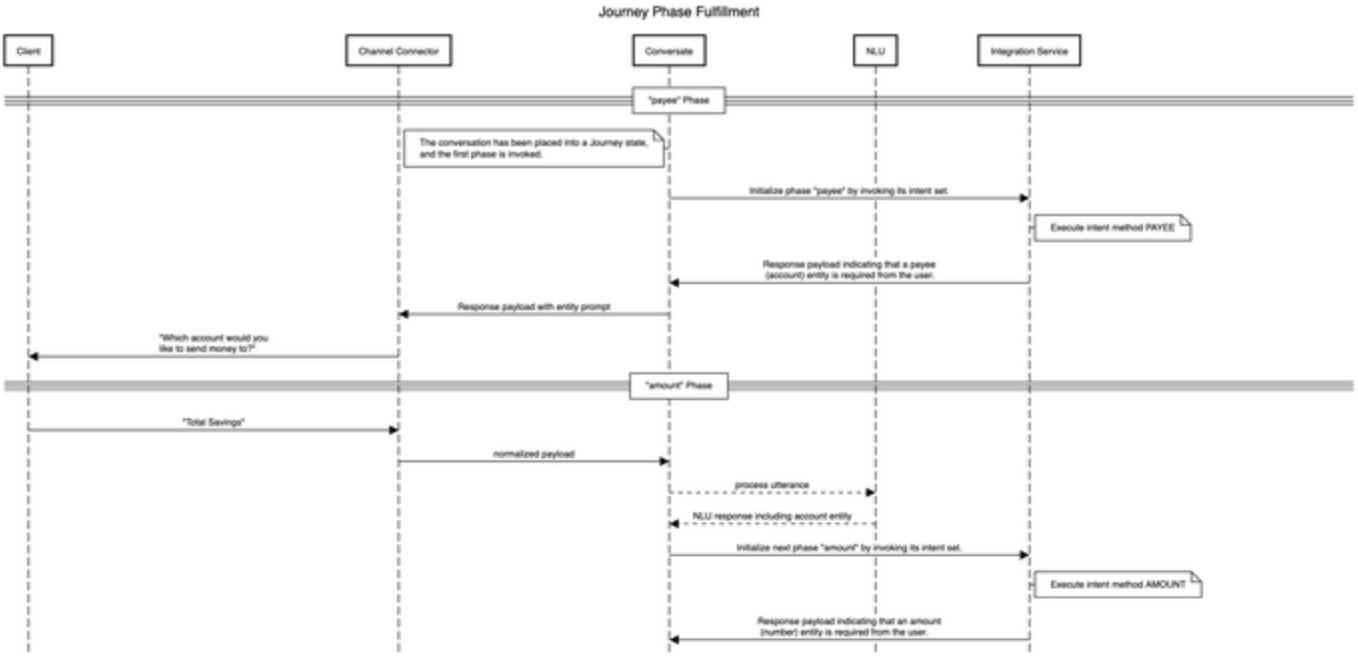
A Journey is invoked by a single intent method who's job is to initialize the internal Journey state object, and to notify Conversate that the conversation should be placed into a Journey state. This is achieved by configuring an intent to utilize a "front door" intent method who's job is to initialize a Journey state object and to notify Conversate that the conversation should be put into a Human Handoff state.

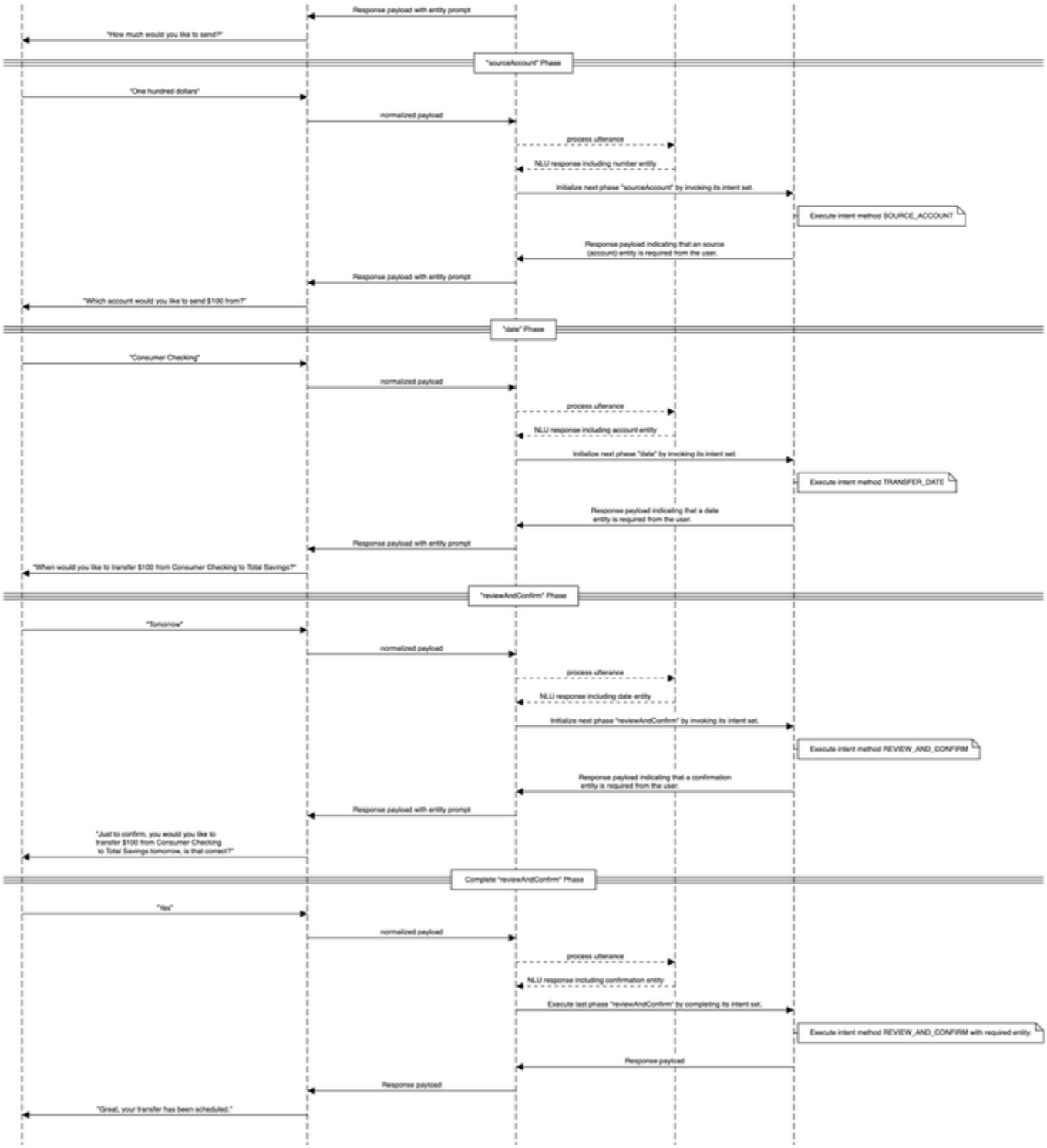


Journey Phase Fulfillment

Once Conversate puts a conversation into a Journey state the end user will be directed through the sets of intents defined by the Journey's phases. These steps are managed by Conversate, the intent's configured Integration Service, and the Journey state object created by the intent method which initialized the Journey.

Again, using the money movement example, we can walk through the fulfillment of a Journey's phases.



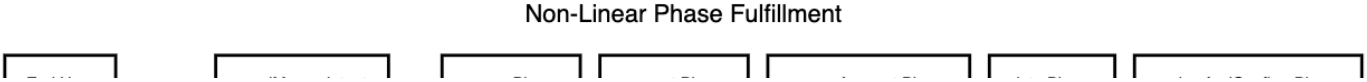


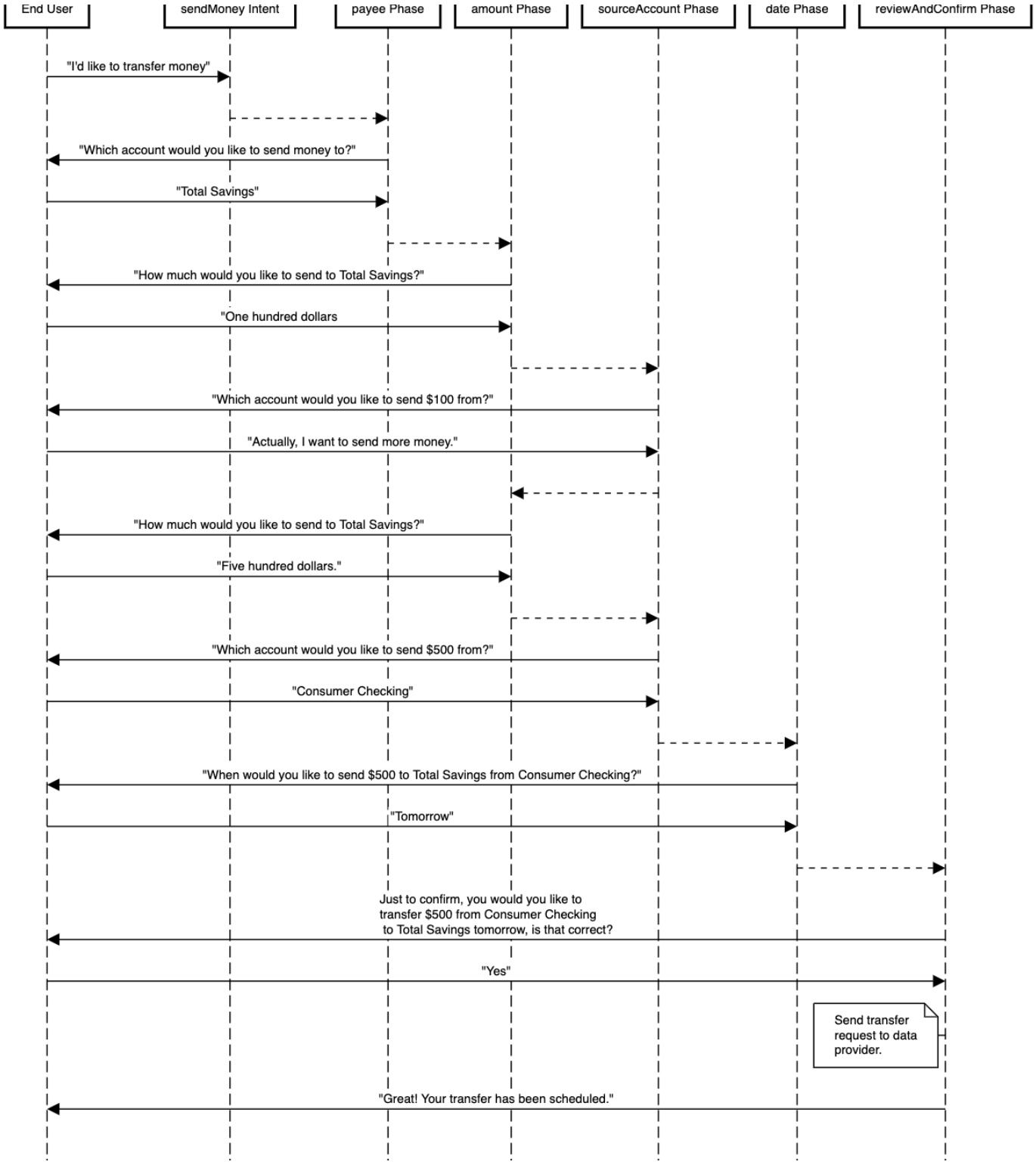
Non-Linear Phase Fulfillment

An important feature of Journeys and their phased architecture is that end users are able to navigate non linearly between phases by providing utterances to the Agent.

Using the move money example Journey, if the end user changed their mind about the amount they wanted to transfer to Total Savings they could simply tell the agent that they want to send a different amount. An intent configuration that is given proper utterance training data will be able to redirect the user back to the desired phase intent.

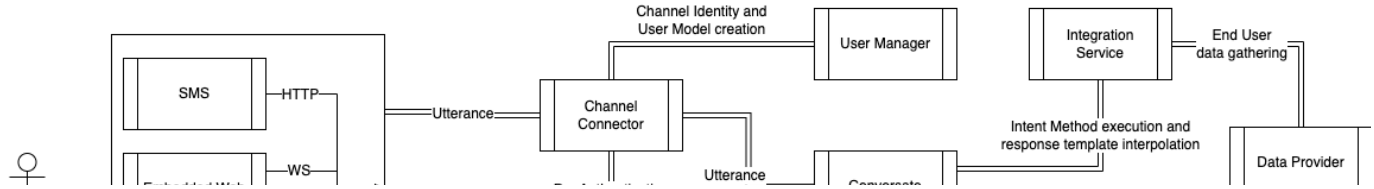
Once the desired previous intent phase is fulfilled the conversation will redirect back to the phase it left off on before the non-linear phase redirect.

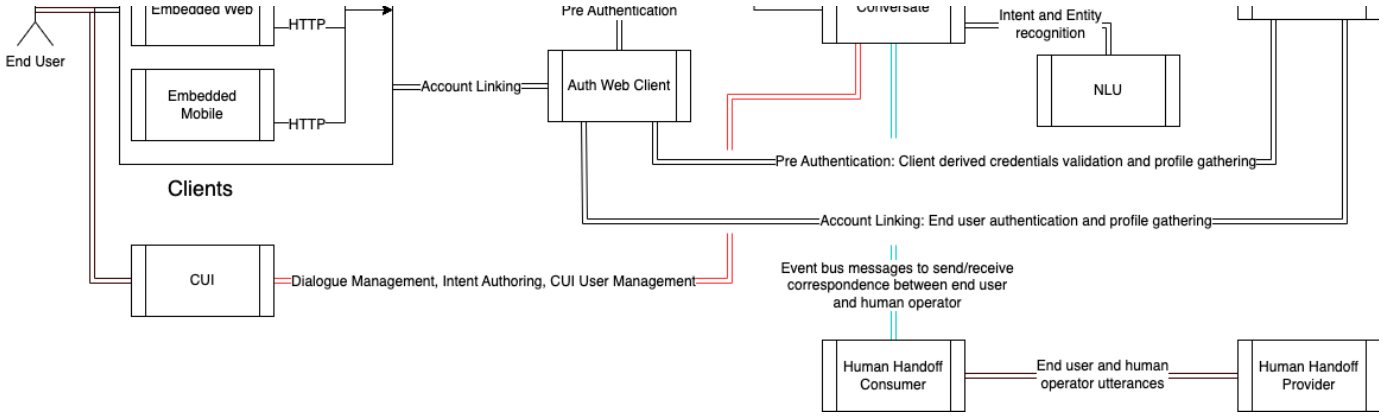




Architecture Overview

The diagram below is an holistic overview to how the Clients and Services communicate with one another. This should better illustrate how Features and the overall End User Request flow are facilitated by the various system components and micro services.




End User
Embedded Web
HTTP
Embedded Mobile
HTTP
Clients
Account Linking
Pre Authentication
Auth Web Client
Converse
Intent and Entity recognition
NLU
Pre Authentication: Client derived credentials validation and profile gathering
Account Linking: End user authentication and profile gathering
CUI
Dialogue Management, Intent Authoring, CUI User Management
Event bus messages to send/receive correspondence between end user and human operator
Human Handoff Consumer
End user and human operator utterances
Human Handoff Provider