OIT Chatbot Support Documentation

Add Headings (Format > Paragraph styles) and they will appear in your table of contents.

Installation/Setup

Install Prerequisites

IDE/Text editor
For Windows OS, it is use Visual Studio Community 2017
https://www.visualstudio.com/downloads/

For other operating systems, use Visual Studio Code
https://code.visualstudio.com/

Project frameworks and libraries
.Net Core 2.0
https://www.microsoft.com/net/download/windows

Node.Js 8.X.X LTS (installing node js should install npm with it)
https://nodejs.org/en/

Webpack CLI (for bundling using the webpack configuration files)
After installing node and npm: `$ npm install -g webpack-cli`

Get from Github

Clone to your local repository and make subsequent pushes to OIT's SVN or a new github repository under your group's github account.

```
$ git clone https://github.com/lee6346/OITChatSupport.git
[yourDirName]
```

Install Project dependencies

1. Make sure all projects are loaded in src/ (except OITChatBotSupport.ChatBot) and right-click the OITChatBotSupport.Web -> set as startup project
2. Visual Studio should automatically install the .NET Core dependencies listed in each of the project's *.csproj files. If you see yellow triangles under the Dependencies/ directory icon, right-click the project -> build
3. In the OITChatBotSupport.Web root directory, install the node_modules/ dependencies: `$ npm install`

Bundle the angular application

1. OITChatBotSupport.Web contains 2 different Angular application configuration files: ClientApps/AgentPortal/config/, ClientApps/Chatbot/config/
2. Change to each of the above directories and type: `$ webpack --config webpack.dev.js`
3. `Once successfully completed, the bundles will appear in the wwwroot/ directory`

Run the application

1. CTRL+F5 on Visual Studio will start the application without debug mode
2. A blank HTML page with a ‘click here’ text should be displayed
3. Click it to server the chatbot application (the chat window students use to talk to the chatbot)
4. Navigate to <url:port>/home/agent to serve the agent portal application

View Angular code documentation

Under the OITChatBotSupport.Web root directory, the package.json file contains a set of scripts. To generate documents, then serve it on your machine to view:

```
$ npm run generatechatbotdocs
$ npm run renderchatbotdocs
```

Go to the localhost:<portno> to view
Repeat the npm commands for the agent portal documents

Glossary

Agent	UTSA’s OIT agents who handle student questions
Agent Transfer Requests	Requests made by students to talk to live agents in the case that the bot cannot correctly answer the student’s question
Chat Session	The conversation/thread information and all messages exchanged in the conversation
Conversation/Thread	Contextual information about a chat session between student and bot/agent. Contains a unique identifier, created time,
Direct Line	Microsoft’s real-time messaging channel to connect to the chatbot.

Student/User	Primary user of the bot services

Useful Links/Terms (3rd party libraries/frameworks)

Angular 5

A client-side javascript framework used to build single page applications

Terms

Component: a decorated class with HTML template and CSS styles representing a patch of the web page. (Similar to partial views)

Directives: a decorated class embedded into HTML tags (including components) for adding effects, additional styles, etc. (Components are a special type of directive)

Pipes: a decorated class that takes some data input and returns a new output (similar to linux pipes). Used in HTML templates to transform component data (Ex: date pipe to transform the date format)

Services / DI (dependency injection): decorated class providing services for components (handling http requests, etc). Angular registers all services with singleton lifetime scope in its DI container, where a service is requested/resolved most commonly through the requesting component's constructor

Module: a decorated class consisting of a collection of components, services, pipes, directives, etc. to represent application features (Ex: chat module consists of chat window, input bar, http services to send and retrieve messages, etc.)

Links

Official documentation
https://angular.io/docs

ASP.NET Core / .NET Core

.NET Core is the open source library built on top of .NET standard for C# applications. ASP.NET Core is a web framework built on .NET Core.

Terms

Controllers/Actions:

- By default ASP.NET Core maps URLs with the controller class name and method (action) name
    - Ex: localhost:5000/test/A maps to the TestController class' A() method
- Use attribute filters on controllers to change the URL it maps to
    - Ex: [Route('....')]
- Controllers returning IActionResult can return view pages, JSON data types, and HTTP response code
- By default, returning a View() will map to the subdirectory under View/ and the file that matches the controller action
    - Ex: TestController class' A() method will in the Views/Test/ directory for A.cshtml file

Dependency Injection:
- IServiceCollection is used as the IOC container to register service components to use in the application
- Service components are registered in the Startup.cs file in the ConfigureServices method.
- Services lifetime scopes can be configured as singleton (one throughout application lifetime), scoped (one per each web request), or transient (one for each time it is resolved)
- Services can be resolved by injecting them in the constructors or methods requesting the service components. They can also be resolved explicitly from the service collection container

Middleware:
- Http request/response handlers configured as a pipeline.
- Middleware components are added to the pipeline in the Startup.cs file under Configure method. Order matters.
- All Http requests go through the middleware pipeline prior to hitting a web controller
- All Http responses go through the middleware pipeline in reverse order before sent back to the client.
- The pipeline can configure middleware components to handle all requests, or requests based on specific web URLs

Configuration:
- AppSettings.json: application settings (logging, URIs, etc.). If you store connection strings and API keys/secrets, make sure to remove if pushing to public repository
- MSBuild files: all projects have a *.csproj file (right-click project to view if using Visual Studio IDE). Contains assembly information, the target framework, other project references, and other libraries used
- LaunchSettings.json: under Properties/ directory. Configures settings like authentication, URL and port numbers, etc for development, staging and production environments

Public static files
- Any static file under the wwwroot/ directory are publicly accessible
- Store public javascript bundles, css files, vendors, and assets (images, etc.)

Links

Official documentation
https://docs.microsoft.com/en-us/aspnet/core/

Dapper ORM

A micro-orm for mapping SQL columns to objects in C#. Dapper is extremely lightweight and its CRUD operations are identical in speed to using the ADO.NET SQL client connector by itself

Links

Official documentation
http://dapper-tutorial.net/api

MediatR

A mediator library to dispatch messages to appropriate handlers to process data

Terms

Message
- Messages can be request commands (present) or notifications (past event)
- Messages can contain data payloads (DTOs, etc.)
- Messages are dispatched using the mediator

Message Handlers
- Each message type is handled using request / notification handlers
- The handler services must be registered with the service collection
- The handlers are used as a facade layer to coordinate different service components and validators to perform CRUD operations in persistence, make API calls, etc.
- Special processor handlers can be used to create pipeline of handlers for each message type (similar to the middleware pipeline for web requests)
- Preprocessors and postprocessors are hooks to handle messages before and after the main message handler handles the message
-

Links

Official documentation
https://github.com/jbogard/MediatR/wiki

Example applications
https://github.com/jbogard/ContosoUniversityCore

Rxjs (Reactive Javascript)

A reactive programming library for javascript applications. Used in angular and NGRX

Terms

Observable: emits items for observers to subscribe to (item producer)

- Cold observables: each time the observable is subscribed to, the observer gets its own execution thread to handle the items emitted
- Hot observables: all observers that subscribe to the observable share the same execution thread (multicasting)

Observer/Subscriber: subscribes to observables to handle the items (item consumer)

- Each observer handles 3 scenarios for the observable (when item is emitted, when an error is thrown, when the observable finishes and stops emitting items)

Operators: functions that operate on items emitted by observables.

- Operators map, filter, and reduce items emitted before returning them. (Ex: map a model to a view-model)
- Operators can be piped

Subjects: observable + observer. It can emit items and subscribe to other observables

- Regular subjects/observables emit over time which means that a new observer cannot subscribe to items emitted prior to subscribing (the tree that fell emitted a noise, we were just too late to hear it)
- Special subjects (behavior subjects, replay subjects, etc.) cache past items so later observers can subscribe to them
- Subjects are often used as a proxy between observers and observables. The subject will subscribe to the original observable and handle the emitted item to emit a new item using itself as the observable, then the end observer will subscribe to the subject.
- You can create subjects and use it to manually emit items

Links

Reactive programming concepts:
http://reactivex.io/documentation/observable.html

Official documentation:
http://reactivex.io/rxjs/manual/overview.html

Rx Marbles (play with observable operators to view how it is emitted and changes)
http://rxmarbles.com/

Misc. articles
https://netbasal.com/javascript-observables-under-the-hood-2423f760584
https://medium.com/@benlesh/learning-observable-by-building-observable-d5da57405d87
https://medium.com/@benlesh/on-the-subject-of-subjects-in-rxjs-2b08b7198b93

NGRX

A functional-reactive library for angular applications used to create a store-based architecture (similar to Redux)

Terms

Actions: Messages with optional data payload that are dispatched to the store to modify the application state

Effects: Services that handle side effects from dispatched actions (Ex: handler HTTP server requests)

Selectors: special functions used to get slices of the application state.
- Selectors are like SQL commands (WHERE, JOIN, etc) where you can filter and combine data from different feature states to produce a new state
- Selectors use memoization to track the most recent state update for efficiency

State: represents the application data
- Application state: the collection of all feature states (like SQL schema)
- Feature states: data for a specific feature of the application (like SQL table)

Store: holds the application state and operators to modify the state

Links

Official documentation (documentation is in markdown files in each directory)
https://github.com/ngrx/platform/tree/v4.1.1/docs

Ngrx under the hood (recommended to get an in depth understanding)
https://gist.github.com/btroncone/a6e4347326749f938510

Example applications
https://github.com/ngrx/platform/tree/v4.1.1/example-app

Misc. articles
https://blog.angular-university.io/angular-ngrx-store-and-effects-crash-course/
https://blog.nrwl.io/ngrx-patterns-and-techniques-f46126e2b1e5

SignalR

A real time data transmission library that uses websockets, polling, and server-sent events protocols to push data to clients.

Terms

Hubs

- Manages that manage real time connections with clients and receive/send data using message invocation
- Each Hub is configured to map to a URL
- Hubs can be secured like web controllers using authentication and authorization policies
- A Hub can create groups where messages are broadcast by group
- A Hub can also broadcast to all connected to that particular hub
- A hub can also send to specific clients by the connection ID or a custom user ID

Transport

- Manages the data transmission protocol (web socket, polling, etc) and serialization type (JSON, message pack, protobuf)

Links

Official documentation
https://github.com/aspnet/SignalR/tree/dev/specs

Example applications
https://github.com/aspnet/SignalR/tree/dev/samples
https://github.com/damienbod/AspNetCoreAngularSignalR/tree/master/src/AspNetCoreAngularSignalR
https://github.com/moozzyk/SignalRCoreAuction/tree/master/src/AuctionR

Misc articles
https://radu-matei.com/blog/real-time-aspnet-core/ (understand how web sockets work)

Webpack

A module bundler for javascript applications. Webpack takes javascript-based modules, static files, HTML templates, and stylesheets and compiles, extracts, and bundles into a single static file to be served to the client

Terms

Module/Graph

- Webpack uses graph dependency to locate all the files/modules to be bundled
- It takes an entry point module and tracks other modules from it recursively using ES imports

Entry

- The location for bundles (ex: application bundle, polyfill bundle, vendor bundle)

Output

- The output directory, names, and hash configurations to put the final bundle

Loaders

- Used to transform modules/files as required for the final bundle
- Multiple loaders can be used for the same module/file in pipeline
- Each loader type manages a specific file/module type (Ex: typescript loaders will transpile typescript code into javascript)

Plugins

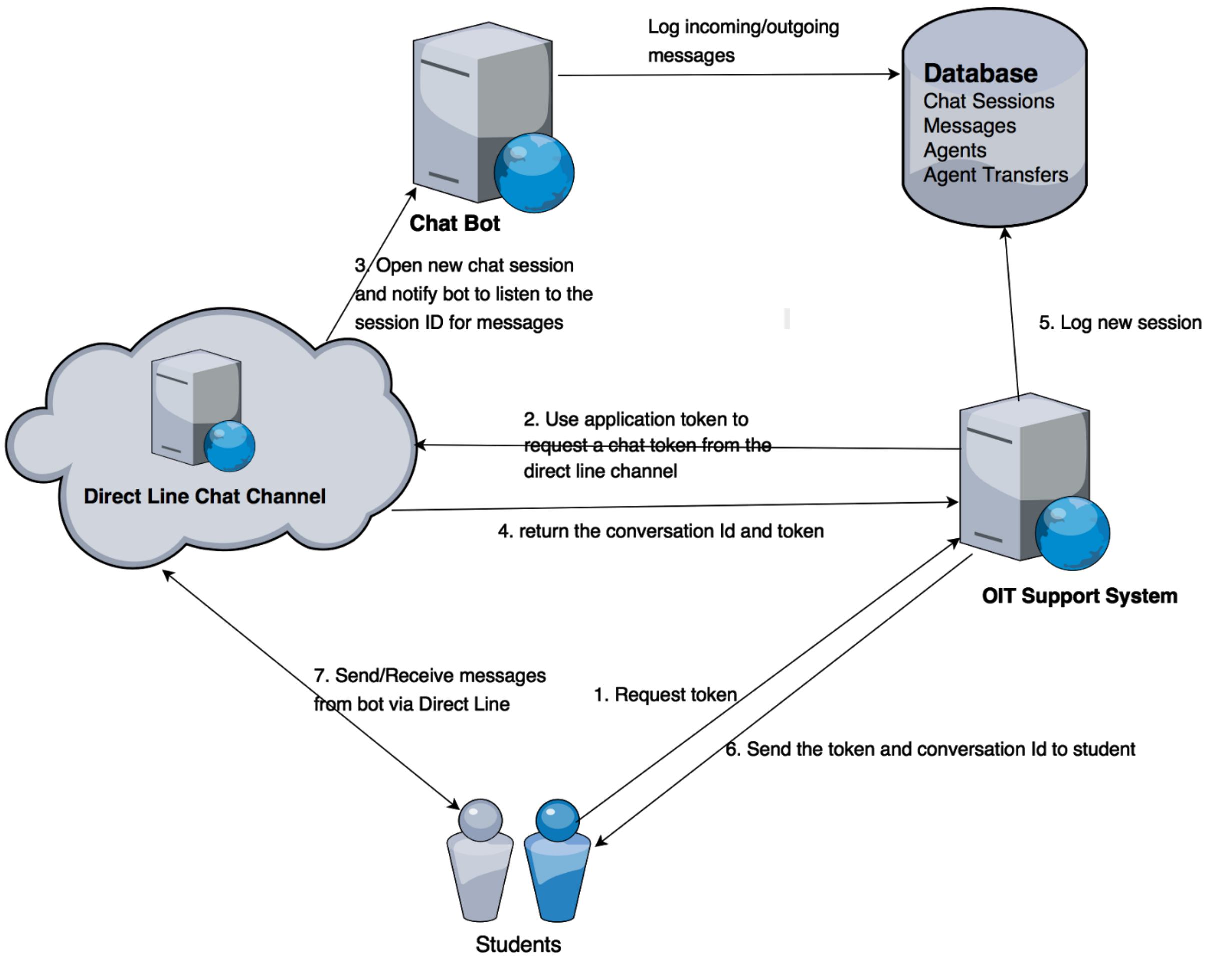
- Optional features to integrate with webpack for bundle optimization, output file extractions, etc.

Links

Official documentation
https://webpack.js.org/concepts/

Network Architecture

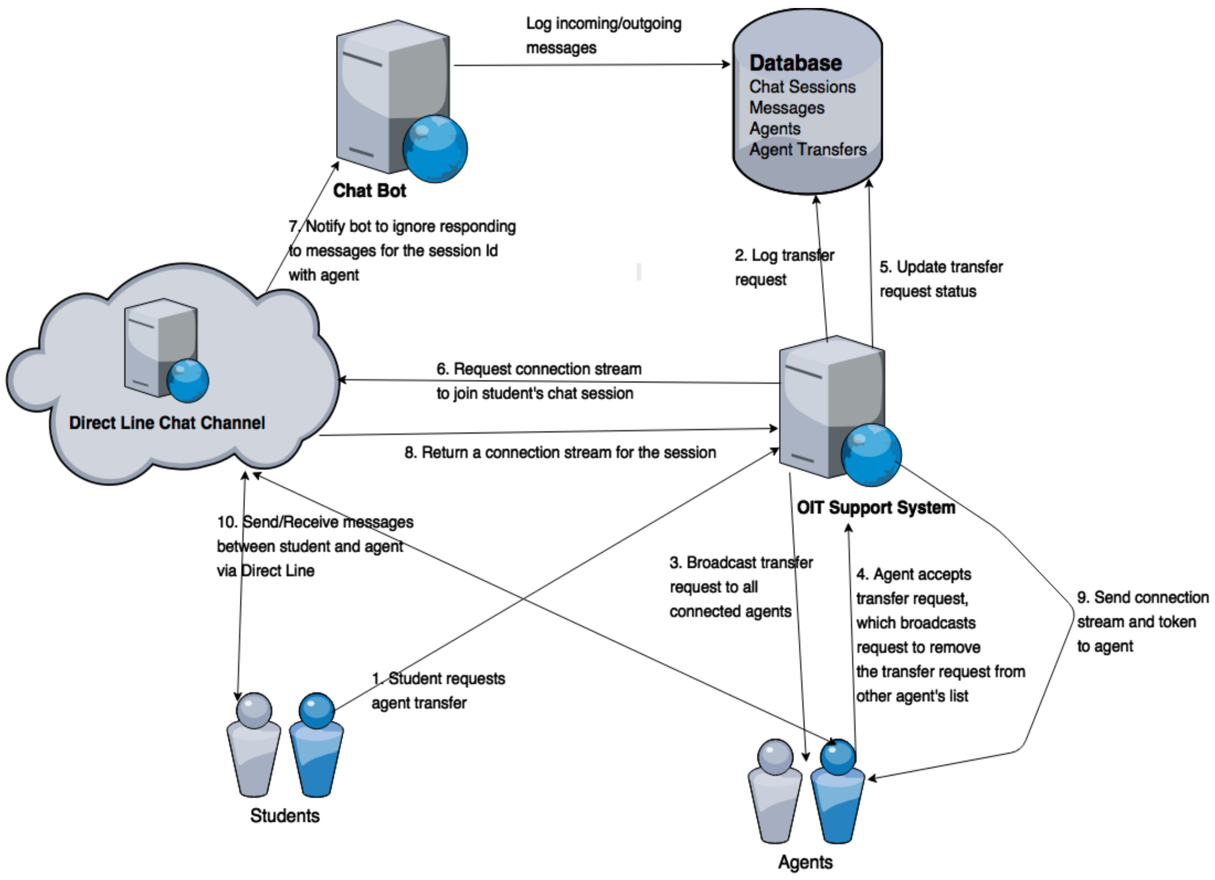
Chatbot interaction


Log incoming/outgoing messages
Database
Chat Sessions
Messages
Agents
Agent Transfers
Chat Bot
3. Open new chat session and notify bot to listen to the session ID for messages
5. Log new session
2. Use application token to request a chat token from the direct line channel
Direct Line Chat Channel
4. return the conversation Id and token
OIT Support System
7. Send/Receive messages from bot via Direct Line
1. Request token
6. Send the token and conversation Id to student
Students

1. Once the chat window initializes, it sends an HTTP Get request to the server to start a chat session.
2. The web server uses the direct line secret to request a token from the direct line server.
3. The Direct Line server creates a new chat session with a session ID. The chatbot is notified to start listening to this session to respond to messages.
4. The Direct Line returns the session Id and a token with expiration time.
5. The new chat session is logged in persistence

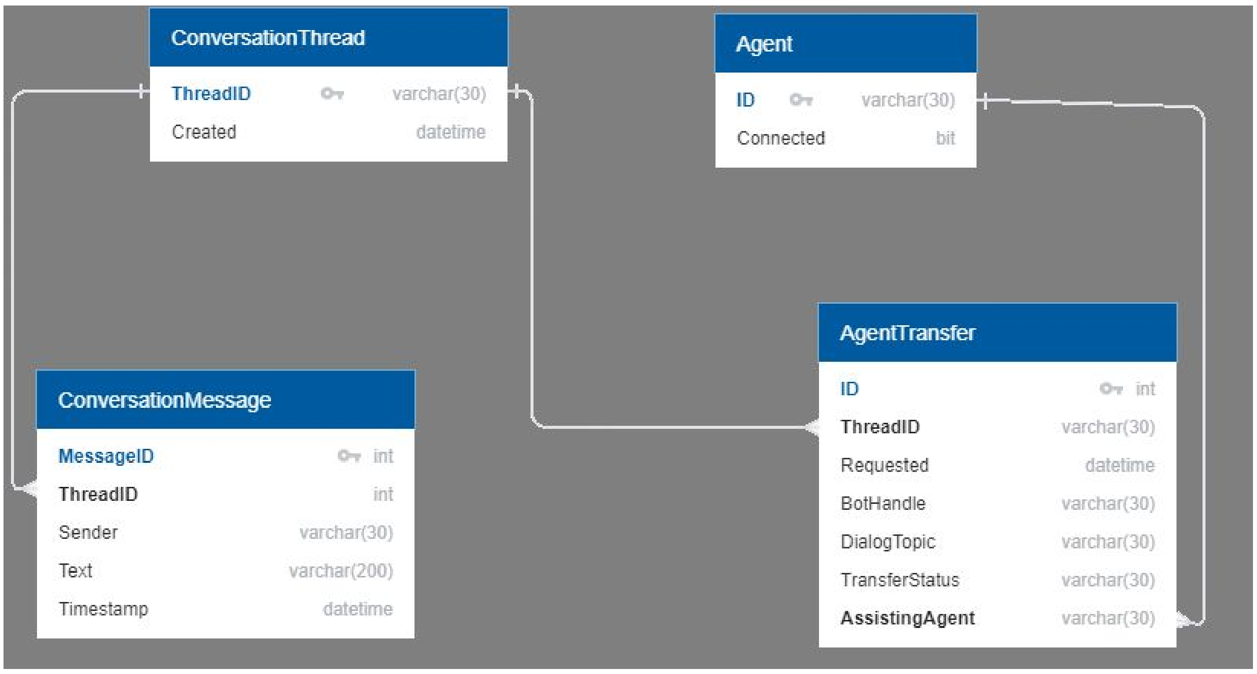
6, 7) The token and session Id are sent to the student/user who will then start sending messages and receiving replies from the chat bot via the Direct Line server

Agent Transfer interaction


Chat Bot
Log incoming/outgoing messages
Database
Chat Sessions
Messages
Agents
Agent Transfers
7. Notify bot to ignore responding to messages for the session Id with agent
2. Log transfer request
5. Update transfer request status
Direct Line Chat Channel
6. Request connection stream to join student's chat session
8. Return a connection stream for the session
OIT Support System
10. Send/Receive messages between student and agent via Direct Line
3. Broadcast transfer request to all connected agents
4. Agent accepts transfer request, which broadcasts request to remove the transfer request from other agent's list
9. Send connection stream and token to agent
1. Student requests agent transfer
Students
Agents

1) The student/user hits the agent transfer button sending an HTTP Post request with the chat session ID and contextual data (bot handle, last message where bot failed, etc) to the server.
2) The cache of agents are first checked to see if any agents are available. If not, the appropriate message is returned to the student/user to try at a later time. Otherwise, the request data is added to the cache for agents who login after to access, then it is logged in persistence
3) The transfer request is broadcast to all connected agents using web sockets, and the student/user is notified to wait for the next available agent.
4) An agent selects the request which sends an HTTP Post request with the chat session ID.
5) If the session ID exists as a key in the cache, the request is removed from it to prevent multiple agents from joining the same conversation. The agent transfer status in persistence is updated, and the session ID is broadcast the other agents which removes it from their list of transfer requests
6) The session ID is used to make an API call to the Direct line server to retrieve a connection stream for that session.
7) The bot is notified to stop responding to messages received from that chat session. But it will still continue logging both agent/student messages in persistence
8, 9, 10) The stream url is retrieved and sent to the agent who uses it to start conversing with the student

Database Schema



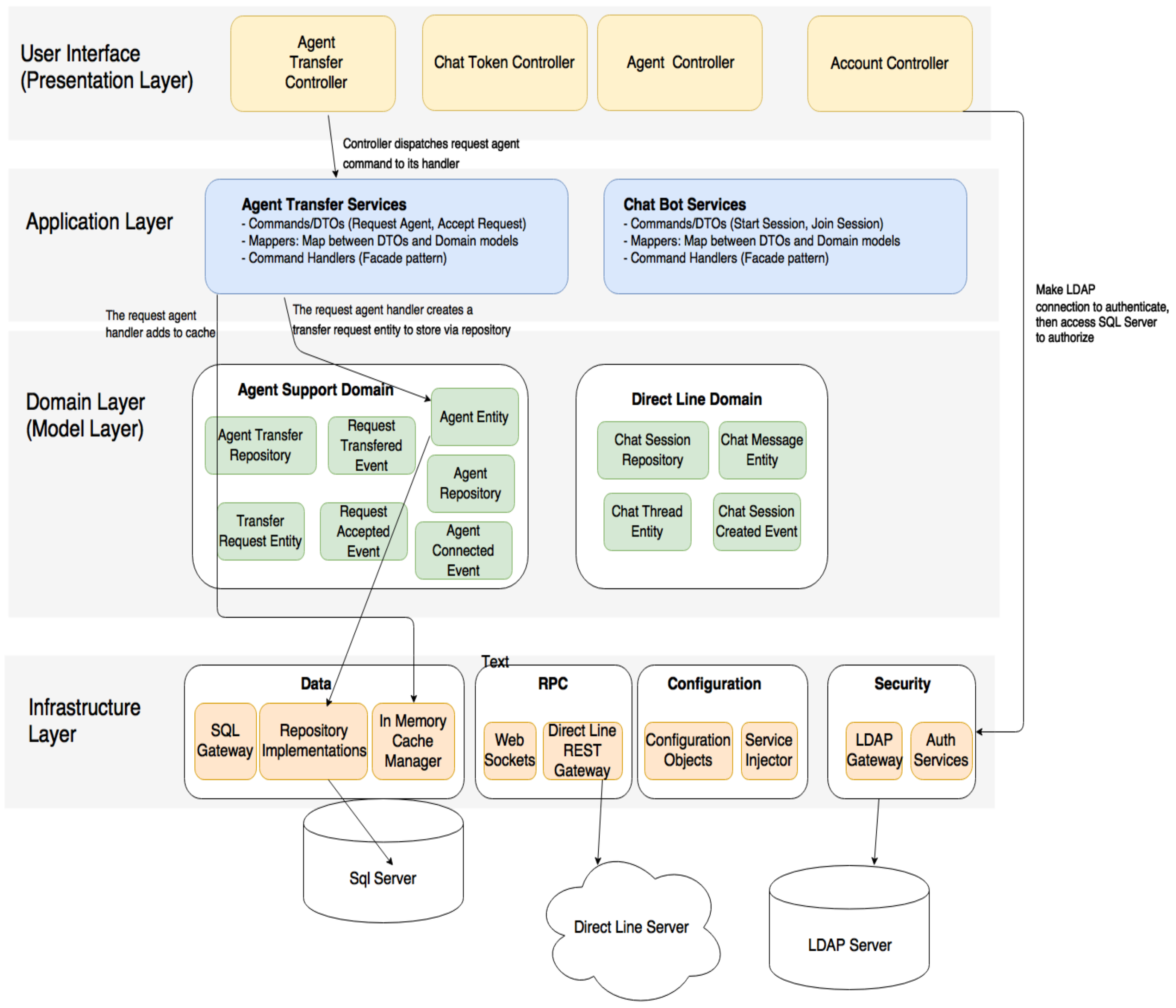
Agent: Used for authorizing agents for specific features such as accepting transfer requests, sending group messages, etc. (Note: agent group messages are not persisted per our client's requests)

Conversation Thread: Stores every new chat session created. Has a one to many relation with conversation messages and a one to one relation with agent transfers. Used to assess frequency of application use

Conversation Message: Stores every message from students, bots and agents via Direct line channel. Used to assess average chat session duration by time and number of messages

Agent Transfer: a log to store every agent transfer requested by students. Has a one to zero or one relation with agents, which depends if an agent accepts the request or the student cancels/exits prior to an agent being able to respond

Server Side (N-tier architecture)


User Interface (Presentation Layer)
Agent Transfer Controller
Chat Token Controller
Agent Controller
Account Controller
Controller dispatches request agent command to its handler
Application Layer
Agent Transfer Services
- Commands/DTOs (Request Agent, Accept Request)
- Mappers: Map between DTOs and Domain models
- Command Handlers (Facade pattern)
Chat Bot Services
- Commands/DTOs (Start Session, Join Session)
- Mappers: Map between DTOs and Domain models
- Command Handlers (Facade pattern)
Make LDAP connection to authenticate, then access SQL Server to authorize
The request agent handler adds to cache
The request agent handler creates a transfer request entity to store via repository
Domain Layer (Model Layer)
Agent Support Domain
Agent Transfer Repository
Request Transfered Event
Agent Entity
Agent Repository
Transfer Request Entity
Request Accepted Event
Agent Connected Event
Direct Line Domain
Chat Session Repository
Chat Message Entity
Chat Thread Entity
Chat Session Created Event
Infrastructure Layer
Text
Data
SQL Gateway
Repository Implementations
In Memory Cache Manager
RPC
Web Sockets
Direct Line REST Gateway
Configuration
Configuration Objects
Service Injector
Security
LDAP Gateway
Auth Services
Sql Server
Direct Line Server
LDAP Server

Presentation Layer

Handles all web requests through the web middleware pipeline. Serves static/bundled angular applications and JSON data via API controllers. Also uses SignalR Hubs to retrieve and push messages between clients. The Presentation layer uses data transfer objects to map to/from the JSON data, and the mediator to dispatch messages, and return Http responses

Application Layer

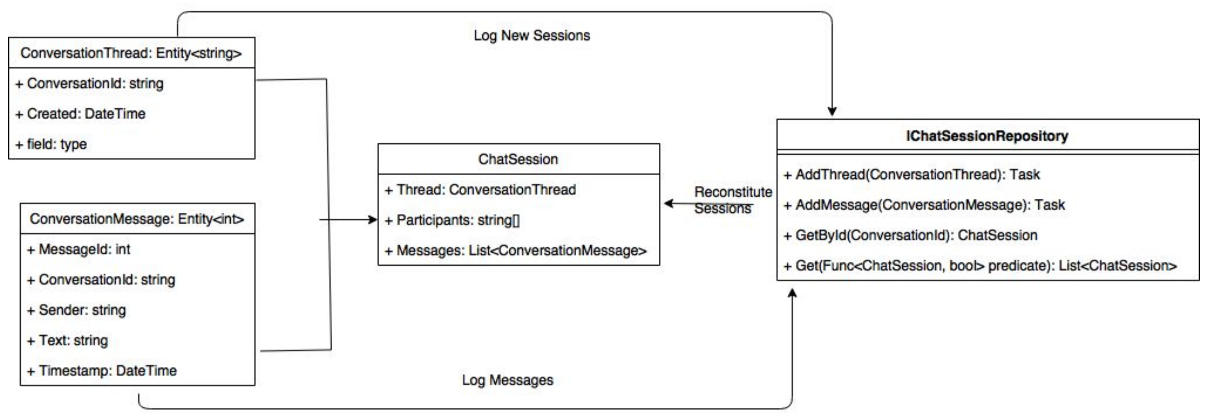
Handles all messages dispatched by the mediator. This layer uses repositories to save / query data from SQL, map it to/from DTOs and return to the mediator

The purpose of this layer is to abstract the services away being used in the web layer's controllers. This makes it easier to integrate the application with clients via native phone applications

Domain Layer

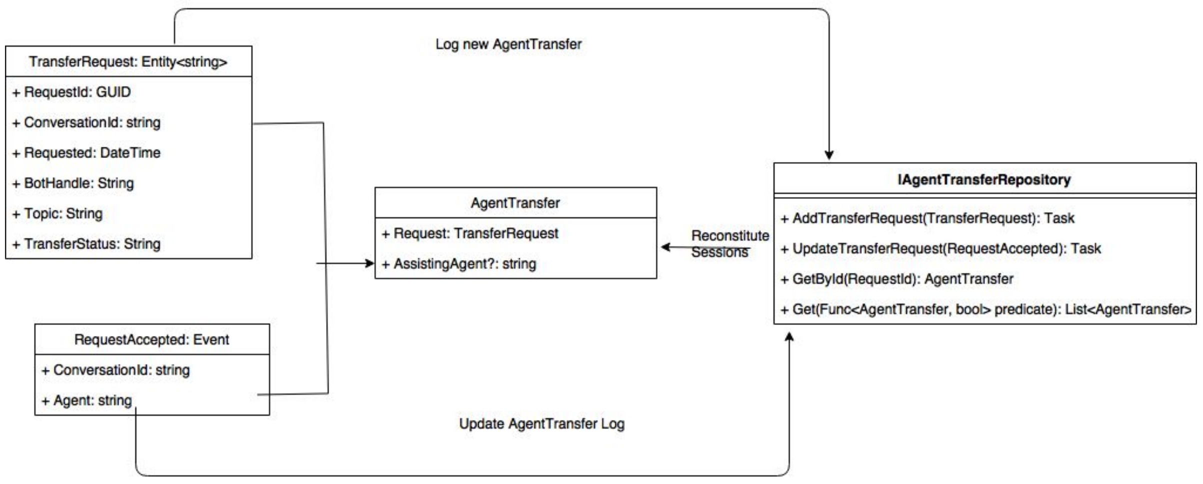
Consists of entity models, value objects, and repositories for saving and retrieving the models.

Direct Line Chat Context
The Direct Line chat context represents the chat message and chat thread models with their respective validators.
The Chat session is the aggregate model containing the Chat session ID, created time, list of messages, and participants, thus representing whole chat session information for administrative use.
A chat session repository interface defines the contract to access necessary data to create and reconstitute the sessions in persistence.



ConversationThread: Entity<string>
+ ConversationId: string
+ Created: DateTime
+ field: type
ConversationMessage: Entity<int>
+ MessageId: int
+ ConversationId: string
+ Sender: string
+ Text: string
+ Timestamp: DateTime
ChatSession
+ Thread: ConversationThread
+ Participants: string[]
+ Messages: List<ConversationMessage>
IChatSessionRepository
+ AddThread(ConversationThread): Task
+ AddMessage(ConversationMessage): Task
+ GetById(ConversationId): ChatSession
+ Get(Func<ChatSession, bool> predicate): List<ChatSession>
Log New Sessions
Reconstitute Sessions
Log Messages

Agent Support Context
The Agent support context represents agent and agent request models with their respective validators.
 The agent transfer event log is the aggregate model containing the Request ID, request time, contextual data (current topic, the bot handle name, etc), the transfer status (waiting, canceled, accepted) and the agent who accepted the request. This aggregate model is used for subsequent analysis for assess the success rate of bots.
The agent transfer repository interface defines the contract to access necessary data to create and reconstitute the transfer events



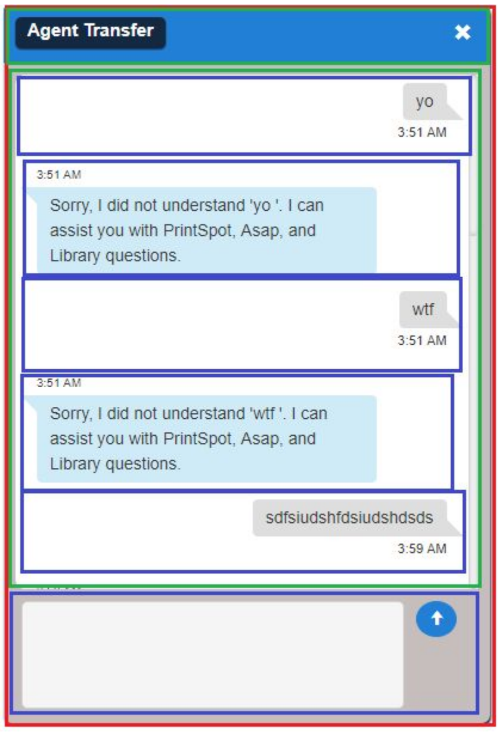
Infrastructure Layer

Manages application configurations, security, consuming from external resources, and other cross-cutting concerns

NGRX Store-Based Architecture

Components and Component interaction

An Angular component is a decorated class that contains styles and html mark up to represent a patch of the view page in the UI interactions inside it (like widgets). Components can have child components which are structured as a tree. Interaction between components is done by emitting events or passing data through input DOM markers


Agent Transfer

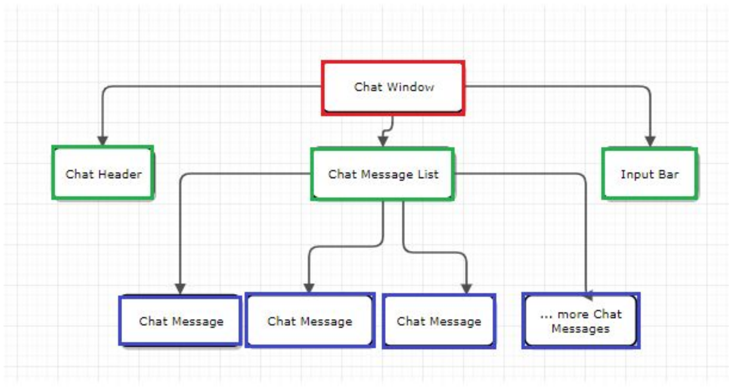
yo
3:51 AM

3:51 AM
Sorry, I did not understand 'yo '. I can assist you with PrintSpot, Asap, and Library questions.

wtf
3:51 AM

3:51 AM
Sorry, I did not understand 'wtf '. I can assist you with PrintSpot, Asap, and Library questions.

sdfsiudshfdsiudshdsds
3:59 AM

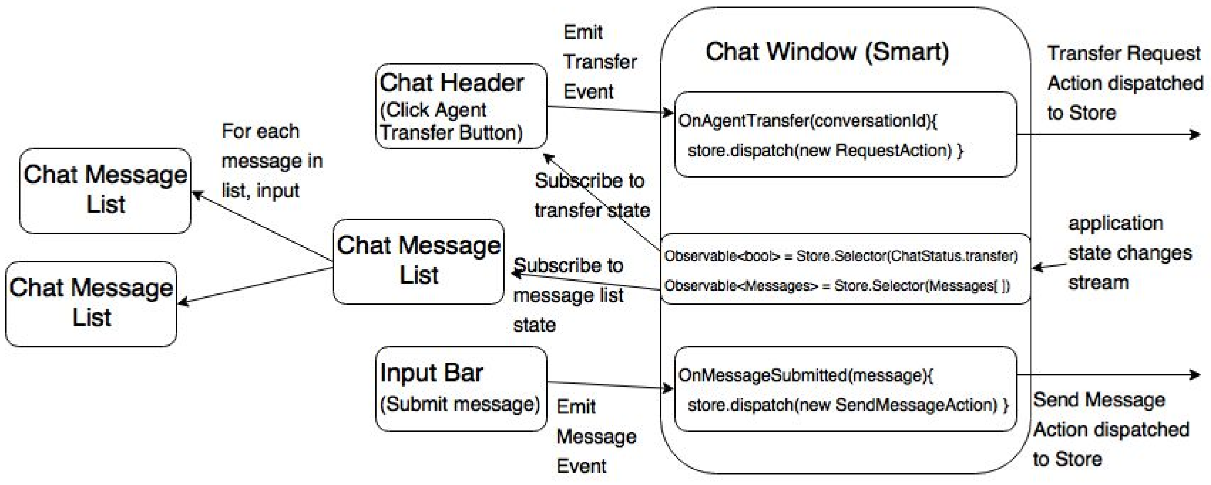


Smart Components vs Dumb Components

The root component is the smart component. It is the facade listens for all changes to the application state in the store and sends those changes to the child componets. All events emitted by the child and descendant components end at the smart components which then dispatch the appropriate actions to change the state based on event emitted

The child components are dumb components. Their only job is to receive changed state data from its input by the root component, and to emit events to the root component. These components are style and markup heavy since they represent the real layout and UI design of the features and widgets

- Data is input from smart to dumb component (downstream)
- Events are output from dumb to smart component (upstream)


Chat Window (Smart)
OnAgentTransfer(conversationId){ store.dispatch(new RequestAction) }
Observable<bool> = Store.Selector(ChatStatus.transfer)
Observable<Messages> = Store.Selector(Messages[ ])
OnMessageSubmitted(message){ store.dispatch(new SendMessageAction) }
Transfer Request Action dispatched to Store
application state changes stream
Send Message Action dispatched to Store
Chat Header (Click Agent Transfer Button)
Emit Transfer Event
Subscribe to transfer state
Chat Message List
Subscribe to message list state
Input Bar (Submit message)
Emit Message Event
For each message in list, input
Chat Message List
Chat Message List

Actions, Reducers

Actions are types with options for carrying a payload. For instance, the action to send message carries the message text as its payload, and a message to request agent transfer has the chat session ID as its payload.

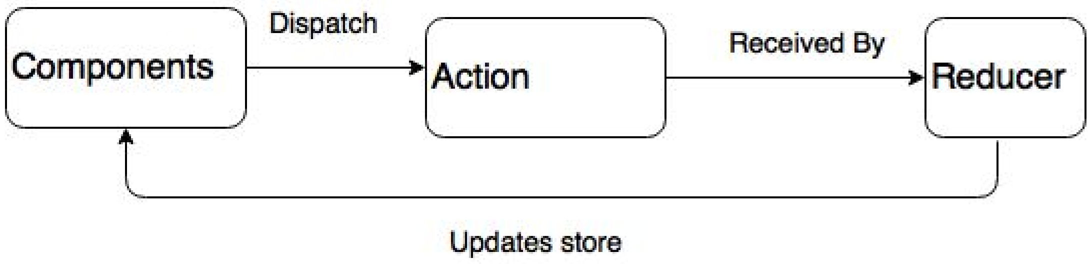
The actions are dispatched using the smart components, and received by the reducers in the store which uses the action type and payload to make the changes to the application state Reducers are pure functions that listen for specific actions. When those actions are received, it returns a copy of the store.

Store, Selectors

Each store is a javascript object where each property represents a some data type. For instance, a chat message store can have the array of messages, the chat session ID, and connection status. Another store can hold agent transfer UI data such as transfer status, which is used to disable or enable the agent transfer button. A Root store encapsulates each store as a property.

Selectors are functions to selector properties from different stores to make different combinations. These functions use memoization to optimize selector results. The selectors allow your smart components to subscribe to them to only track changes to relevant data based on the UI components that need them

One analogy is to see the Root store as the SQL database, each store as a database table, and each property of a store object as a column attribute with its value. Then the selectors would be SQL queries like JOIN, etc

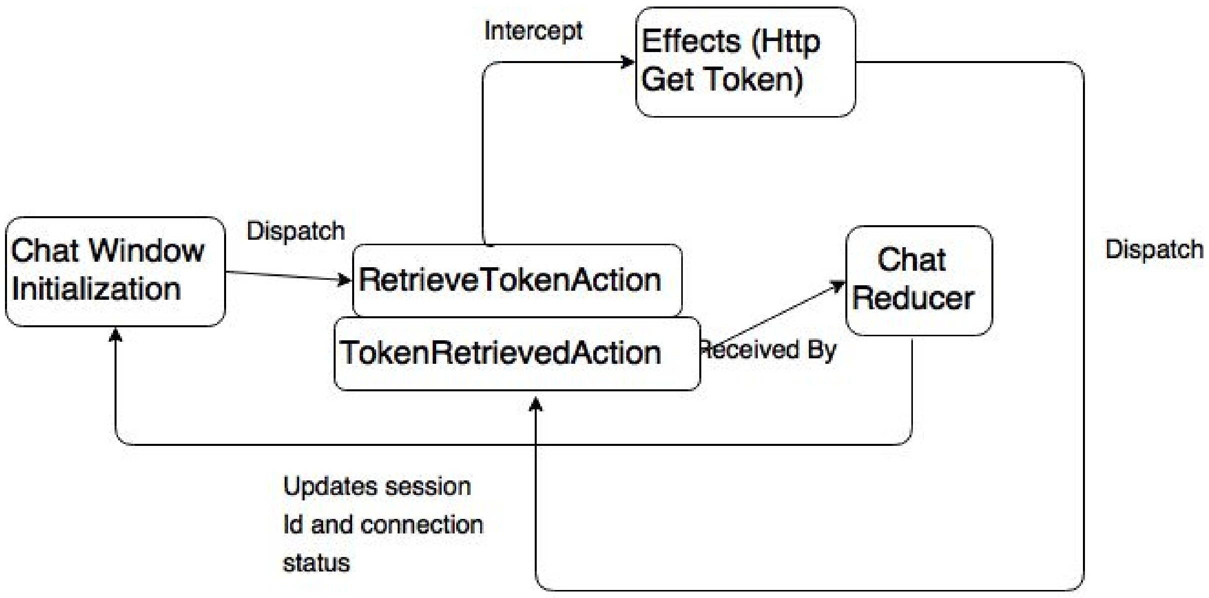


Effects

Even using NGRX's functional-reactive programming, we may want to produce side effects to make HTTP requests to a web server or database and get results. Effects are decorated service components that intercept specific actions to make these calls. Upon receiving data from the server, it would dispatch a new action to the store with the data as its payload

Example: When the agent application is initialized, it sends an action to retrieve all pending agent transfer requests. The effects intercepts to make the HTTP call, receives the list of requests, then dispatches a new action with the list to the store

Even though we produce side effects, any resulting data from it must be loaded into an action and dispatched. This preserves the uni-directional data flow.


Intercept
Effects (Http Get Token)
Chat Window Initialization
Dispatch
RetrieveTokenAction
TokenRetrievedAction
Received By
Chat Reducer
Dispatch
Updates session Id and connection status