# Get started with Teams AI library

Article • 12/13/2023

Teams AI library streamlines the process to build intelligent Microsoft Teams applications by using the AI components. It provides APIs to access and manipulate data, as well as a range of controls and components to create custom user interfaces.

You can easily integrate Teams AI library, prompt management, and safety moderation into your apps and enhance the user experience. It also facilitates the creation of bots that uses an OpenAI API key or Azure OpenAI to provide an AI-driven conversational experience.

## Initial setup

Teams AI library is built on top of the Bot Framework SDK and uses its fundamentals to offer an extension to the Bot Framework SDK capabilities. As part of initial setup, it's important to import the Bot Framework SDK functionalities.

> **Note**
>
> The adapter class that handles connectivity with the channels is imported from **Bot Framework SDK**.

JavaScript

Sample code reference

JavaScript

```
// Import required bot services.
// See https://aka.ms/bot-services to learn more about the different
parts of a bot.
import {
    CloudAdapter,
    ConfigurationBotFrameworkAuthentication,
    ConfigurationServiceClientCredentialFactory,
    MemoryStorage,
    TurnContext
} from 'botbuilder';

// Read botFilePath and botFileSecret from .env file.
const ENV_FILE = path.join(__dirname, '..', '.env');
config({ path: ENV_FILE });
```

```
const botFrameworkAuthentication = new
ConfigurationBotFrameworkAuthentication(
    {},
    new ConfigurationServiceClientCredentialFactory({
        MicrosoftAppId: process.env.BOT_ID,
        MicrosoftAppPassword: process.env.BOT_PASSWORD,
        MicrosoftAppType: 'MultiTenant'
    })
);

// Create adapter.
// See https://aka.ms/about-bot-adapter to learn more about how bots
work.
const adapter = new CloudAdapter(botFrameworkAuthentication);
```

## Import Teams AI library

Import all the classes from `@microsoft/teams-ai` to build your bot and use the Teams AI library capabilities.

Sample code reference

JavaScript

```
// import Teams AI library
import {
    AI,
    Application,
    ActionPlanner,
    OpenAIModerator,
    OpenAIModel,
    PromptManager,
    TurnState
} from '@microsoft/teams-ai';
import { addResponseFormatter } from './responseFormatter';
import { VectraDataSource } from './VectraDataSource';
```

# Create AI components

Add AI capabilities to your existing app or a new Bot Framework app.

**OpenAIModel**: The OpenAIModel class provides a way to access the OpenAI API or any other service, which adheres to the OpenAI REST format. It's compatible with both OpenAI and Azure OpenAI language models.

**Prompt manager**: The prompt manager manages prompt creation. It calls functions and injects from your code into the prompt. It copies the conversation state and the user state into the prompt for you automatically.

**ActionPlanner**: The ActionPlanner is the main component calling your Large Language Model (LLM) and includes several features to enhance and customize your model. It's responsible for generating and executing plans based on the user's input and the available actions.

JavaScript

Sample code reference

JavaScript

```
/// Create AI components
const model = new OpenAIModel({
    // OpenAI Support
    apiKey: process.env.OPENAI_KEY!,
    defaultModel: 'gpt-3.5-turbo',

    // Azure OpenAI Support
    azureApiKey: process.env.AZURE_OPENAI_KEY!,
    azureDefaultDeployment: 'gpt-3.5-turbo',
    azureEndpoint: process.env.AZURE_OPENAI_ENDPOINT!,
    azureApiVersion: '2023-03-15-preview',

    // Request logging
    logRequests: true
});

const prompts = new PromptManager({
    promptsFolder: path.join(__dirname, '../src/prompts')
});

const planner = new ActionPlanner({
    model,
    prompts,
    defaultPrompt: 'chat',
});
```

# Define storage and application

The application object automatically manages the conversation and user state of your bot.

- **Storage**: Create a storage provider to store the conversation and the user state for your bot.
- **Application**: The application class has all the information and bot logic required for an app. You can register actions or activity handlers for the app in this class.

JavaScript

Sample code reference

JavaScript

```
// Define storage and application
const storage = new MemoryStorage();
const app = new Application<ApplicationTurnState>({
    storage,
    ai: {
        planner,
        // moderator
    }
});
```

The `MemoryStorage()` function stores all the state for your bot. The `Application` class replaces the Teams Activity Handler class. You can configure your `ai` by adding the planner, moderator, prompt manager, default prompt and history. The `ai` object is passed into the `Application`, which receives the AI components and the default prompt defined earlier.

# Register data sources

A vector data source makes it easy to add RAG to any prompt. You can register a named data source with the planner and then specify the name[s] of the data sources to augment the prompt within the prompt's `config.json` file. Data sources allow AI to inject relevant information from external sources into the prompt, such as vector databases or cognitive search. You can register named data sources with the planner and then specify the name[s] of the data sources they wish to augment the prompt within the prompt's `config.json` file.

Sample code reference

TypeScript

```
// Register your data source with planner
planner.prompts.addDataSource(new VectraDataSource({
    name: 'teams-ai',
    apiKey:  process.env.OPENAI_API_KEY!,
    indexFolder: path.join(__dirname, '../index'),
}));
```

## Embeddings

An Embedding is a kind of Vector generated by an LLM that represents a piece of text. The text could be a word, sentence, or an entire document. Since the model understands the syntax and semantics of language the Embedding can capture the semantic meaning of the text in a compact form. Embeddings are often used in natural language processing tasks, such as text classification or sentiment analysis, but also be used for search.

The model for generating Embeddings is different from the foundational LLMs. For example, OpenAI provides an embedding model called **text-embedding-ada-002**, which returns a list of 1536 numbers that represents the input text. The system creates embeddings for text within the documents and stores them in a Vector Database. Now from our Chat application we can implement the RAG pattern by first retrieving relevant data about the documents from the Vector Database, and then augmenting the Prompt with this retrieved information.

▼ The following is an example of a VectraDataSource and OpenAIEmbeddings:

TypeScript

```
import { DataSource, Memory, RenderedPromptSection, Tokenizer } from
'@microsoft/teams-ai';
import { OpenAIEmbeddings, LocalDocumentIndex } from 'vectra';
import * as path from 'path';
import { TurnContext } from 'botbuilder';

/**
 * Options for creating a `VectraDataSource`.
 */
export interface VectraDataSourceOptions {
    /**
     * Name of the data source and local index.
     */
    name: string;

    /**
     * OpenAI API key to use for generating embeddings.
     */
```

```
    apiKey: string;

    /**
     * Path to the folder containing the local index.
     * @remarks
     * This should be the root folder for all local indexes and the index
itself
     * needs to be in a subfolder under this folder.
     */
    indexFolder: string;

    /**
     * Optional. Maximum number of documents to return.
     * @remarks
     * Defaults to `5`.
     */
    maxDocuments?: number;

    /**
     * Optional. Maximum number of chunks to return per document.
     * @remarks
     * Defaults to `50`.
     */
    maxChunks?: number;

    /**
     * Optional. Maximum number of tokens to return per document.
     * @remarks
     * Defaults to `600`.
     */
    maxTokensPerDocument?: number;
}

/**
 * A data source that uses a local Vectra index to inject text snippets into
a prompt.
 */
export class VectraDataSource implements DataSource {
    private readonly _options: VectraDataSourceOptions;
    private readonly _index: LocalDocumentIndex;

    /**
     * Name of the data source.
     * @remarks
     * This is also the name of the local Vectra index.
     */
    public readonly name: string;

    /**
     * Creates a new `VectraDataSource` instance.
     * @param options Options for creating the data source.
     */
    public constructor(options: VectraDataSourceOptions) {
        this._options = options;
        this.name = options.name;
```

```
        // Create embeddings model
        const embeddings = new OpenAIEmbeddings({
            model: 'text-embedding-ada-002',
            apiKey: options.apiKey,
        });

        // Create local index
        this._index = new LocalDocumentIndex({
            embeddings,
            folderPath: path.join(options.indexFolder, options.name),
        });
    }

    /**
     * Renders the data source as a string of text.
     * @param context Turn context for the current turn of conversation with
the user.
     * @param memory An interface for accessing state values.
     * @param tokenizer Tokenizer to use when rendering the data source.
     * @param maxTokens Maximum number of tokens allowed to be rendered.
     */
    public async renderData(context: TurnContext, memory: Memory, tokenizer:
Tokenizer, maxTokens: number): Promise<RenderedPromptSection<string>> {
        // Query index
        const query = memory.getValue('temp.input') as string;
        const results = await this._index.queryDocuments(query, {
            maxDocuments: this._options.maxDocuments ?? 5,
            maxChunks: this._options.maxChunks ?? 50,
        });

        // Add documents until you run out of tokens
        let length = 0;
        let output = '';
        let connector = '';
        for (const result of results) {
            // Start a new doc
            let doc = `${connector}url: ${result.uri}\n`;
            let docLength = tokenizer.encode(doc).length;
            const remainingTokens = maxTokens - (length + docLength);
            if (remainingTokens <= 0) {
                break;
            }

            // Render document section
            const sections = await
result.renderSections(Math.min(remainingTokens,
this._options.maxTokensPerDocument ?? 600), 1);
            docLength += sections[0].tokenCount;
            doc += sections[0].text;

            // Append do to output
            output += doc;
            length += docLength;
            connector = '\n\n';
```

```
        }

        return { output, length, tooLong: length > maxTokens };
    }

}
```

# Prompt

Prompts are pieces of text that can be used to create conversational experiences. Prompts are used to start conversations, ask questions, and generate responses. The use of prompts helps reduce the complexity of creating conversational experiences and make them more engaging for the user.

A new object based prompt system breaks a prompt into sections and each section can be given a token budget that's either a fixed set of tokens, or proportional to the overall remaining tokens. You can generate prompts for both Text Completion and Chat Completion style APIs.

The following are a few guidelines to create prompts:

- Provide instructions, examples, or both.
- Provide quality data. Ensure that there are enough examples and proofread your examples. The model is smart enough to see through basic spelling mistakes and give you a response, but it also might assume that the input is intentional and it might affect the response.
- Check your prompt settings. The temperature and top_p settings control how deterministic the model is in generating a response. Higher value such as 0.8 makes the output random, while lower value such as 0.2 makes the output focused and deterministic.

Create a folder called prompts and define your prompts in the folder. When the user interacts with the bot by entering a text prompt, the bot responds with a text completion.

- `skprompt.txt`: Contains the prompts text and supports template variables and functions. Define all your text prompts in the `skprompt.txt` file.

- `config.json`: Contains the prompt model settings. Provide the right configuration to ensure bot responses are aligned with your requirement.

  Sample code reference

  JSON

```
{
   "schema": 1.1,
   "description": "A bot that can turn the lights on and off",
   "type": "completion",
   "completion": {
       "model": "gpt-3.5-turbo",
       "completion_type": "chat",
       "include_history": true,
       "include_input": true,
       "max_input_tokens": 2800,
       "max_tokens": 1000,
       "temperature": 0.2,
       "top_p": 0.0,
       "presence_penalty": 0.6,
       "frequency_penalty": 0.0,
       "stop_sequences": []
   },
   "augmentation": {
       "augmentation_type": "sequence"
       "data_sources": {
            "teams-ai": 1200
    }
   }
 }
```

## Query parameters

The following table includes the query parameters:

Expand table

| Value | Description |
|---|---|
| `model` | ID of the model to use. |
| `completion_type` | The type of completion you would like to use for your model. Given a prompt, the model will return one or more predicted completions along with the probabilities of alternative tokens at each position. Supported options are `chat` and `text`. Default is `chat`. |
| `include_history` | Boolean value. If you want to include history. Each prompt gets its own separate conversation history to make sure that the model doesn't get confused. |
| `include_input` | Boolean value. If you want to include user's input in the prompt. How many tokens for the prompt. |
| `max_input_tokens` | The maximum number of tokens for input. Max tokens supported is 4000. |

| Value | Description |
| --- | --- |
| `max_tokens` | The maximum number of tokens to generate in the completion. The token count of your prompt plus max_tokens can't exceed the model's context length. |
| `temperature` | What sampling temperature to use, between 0 and 2. Higher values like 0.8 makes the output more random, while lower values like 0.2 makes it more focused and deterministic. |
| `top_p` | An alternative to sampling with temperature, called nucleus sampling, where the model considers the results of the tokens with top_p probability mass. Therefore, 0.1 means only the tokens comprising the top 10% probability mass are considered. |
| `presence_penalty` | Number between -2.0 and 2.0. Positive values penalize new tokens based on whether they appear in the text so far, increasing the model's likelihood to talk about new topics. |
| `frequency_penalty` | Number between -2.0 and 2.0. Positive values penalize new tokens based on their existing frequency in the text so far, decreasing the model's likelihood to repeat the same line verbatim. |
| `stop_sequences` | Up to four sequences where the API stops generating further tokens. The returned text won't contain the stop sequence. |
| `augmentation_type` | The type of augmentation. Supported values are `sequence`, `monologue` and `tools`. |

## Prompt management

Prompt management helps adjust the size and content of the prompt sent to the language model, considering the available token budget and the data sources or augmentations.

If a bot has a maximum of 4,000 tokens where 2,800 tokens are for input and 1,000 tokens are for output, the model can manage the overall context window and ensure that it never processes more than 3,800 tokens. The model starts with a text of about 100 tokens, adds in the data source of another 1,200 tokens, and then looks at the remaining budget of 1,500 tokens. The system allocates the remaining 1,500 tokens to the conversation history and input. The conversation history is then condensed to fit the remaining space, ensuring the model never surpasses 2,800 tokens.

## Prompt actions

Plans let the model perform actions or respond to the user. You can create a schema of the plan and add a list of actions that you support to perform an action and pass arguments. The OpenAI endpoint figures out the actions required to be used, extracts all the entities, and passes those as arguments to the action call.

```text
The following is a conversation with an AI assistant.
The assistant can turn a light on or off.

context:
The lights are currently {{getLightStatus}}.
```

## Prompt template

Prompt template is a simple and powerful way to define and compose AI functions using plain text. You can use prompt template to create natural language prompts, generate responses, extract information, invoke other prompts, or perform any other task that can be expressed with text.

The language supports features that allow you to include variables, call external functions, and pass parameters to functions. You don't need to write any code or import any external libraries, just use the curly braces {{...}} to embed expressions in your prompts. Teams parses your template and executes the logic behind it. This way, you can easily integrate AI into your apps with minimal effort and maximum flexibility.

- `{{function}}`: Calls a registered function and inserts its return value string.
- `{{$input}}`: Inserts the message text. It gets its value from state.temp.input.
- `{{$state.[property]}}`: Inserts state properties.

# Actions

Actions handle events triggered by AI components.

`FlaggedInputAction` and `FlaggedOutputAction` are the built-in action handlers to handle the moderator flags. If the moderator flags an incoming message input, the moderator redirects to the `FlaggedInputAction` handler and the `context.sendActivity` sends a message to the user about the flag. If you want to stop the action, you must add `AI.StopCommandName`.

Sample code reference

JavaScript

```
// Register other AI actions
app.ai.action(
    AI.FlaggedInputActionName,
    async (context: TurnContext, state: ApplicationTurnState, data:
Record<string, any>) => {
        await context.sendActivity(`I'm sorry your message was flagged:
${JSON.stringify(data)}`);
        return AI.StopCommandName;
    }
);

app.ai.action(AI.FlaggedOutputActionName, async (context: TurnContext,
state: ApplicationTurnState, data: any) => {
    await context.sendActivity(`I'm not allowed to talk about such
things.`);
    return AI.StopCommandName;
});
```

## Register Action Handlers

Action handlers help users achieve the goals, which is shared in the user intents.

One of the key aspects in action handlers is that you must first register the actions in the prompts and then help user achieve the goal.

You must register a handler for each action listed in the prompt and also add a handler to deal with unknown actions.

In the following example of a light bot, we have the `LightsOn`, `LightsOff`, and `Pause` action. Every time an action is called, you return a `string`. If you require the bot to return time, you don't need to parse the time and convert it to a number. The `PauseParameters` property ensures that it returns time in number format without pausing the prompt.

JavaScript

Sample code reference

JavaScript

```
// Register action handlers
app.ai.action('LightsOn', async (context: TurnContext, state:
ApplicationTurnState) => {
    state.conversation.lightsOn = true;
    await context.sendActivity(`[lights on]`);
```

```
    return `the lights are now on`;
});

app.ai.action('LightsOff', async (context: TurnContext, state:
ApplicationTurnState) => {
    state.conversation.lightsOn = false;
    await context.sendActivity(`[lights off]`);
    return `the lights are now off`;
});

interface PauseParameters {
    time: number;
}

app.ai.action('Pause', async (context: TurnContext, state:
ApplicationTurnState, parameters: PauseParameters) => {
    await context.sendActivity(`[pausing for ${parameters.time / 1000}
seconds]`);
    await new Promise((resolve) => setTimeout(resolve,
parameters.time));
    return `done pausing`;
});
```

If you use either `sequence`, `monologue` or `tools` augmentation, it's impossible for the model to hallucinate an invalid function name, action name, or the correct parameters. You must create a new actions file and define all the actions you want the prompt to support for augmentation. You must define the actions to tell the model when to perform the action. Sequence augmentation is suitable for tasks that require multiple steps or complex logic. Monologue augmentation is suitable for tasks that require natural language understanding and generation, and more flexibility and creativity.

In the following example of a light bot, the `actions.json` file has a list of all the actions the bot can perform:

JSON

```
[
    {
        "name": "LightsOn",
        "description": "Turns on the lights"
    },
    {
        "name": "LightsOff",
        "description": "Turns off the lights"
    },
    {
        "name": "Pause",
        "description": "Delays for a period of time",
        "parameters": {
```

```
            "type": "object",
            "properties": {
                "time": {
                    "type": "number",
                    "description": "The amount of time to delay in millisec-
onds"
                }
            },
            "required": [
                "time"
            ]
        }
    }
]
```

- `name`: Name of the action. Required.
- `description`: Description of the action. Optional.
- `parameters`: Add a JSON schema object of the required parameters.

Feedback loop is a model's response to validate, correct, or refine the answer to your question. If you're using a `sequence` augmentation, you can disable looping to guard against any accidental looping in the following ways:

- You can set `allow_looping?` to `false` in the `AIOptions` definition.
- You can set `max_repair_attempts` to `0` in the `index.ts` file.

### Manage history

You can use the `MaxHistoryMessages` and `MaxConversationHistoryTokens` arguments to allow the AI library to automatically manage your history.

## Feedback loop

A feedback loop allows you to monitor and improve the bot's interactions over time, leading to more effective and user-friendly applications. The feedback received can be used to make adjustments and improvements, ensuring that the bot consistently meets user needs and expectations.

A feedback loop consists of the following:

**Repair Loop**: If the model's response falls short of expectations, it triggers a repair loop. The conversation history forks, enabling the system to try various solutions without impacting the main conversation.

**Validation**: Validation verifies the corrected response. If it successfully passes validation, the system unforks the conversation and reinserts the repaired structure into the main conversation.

**Learn from Mistakes**: Once the model sees an example of correct behavior, it learns to avoid making similar mistakes in the future.

**Handle Complex Commands**: Once the model has learned from its mistakes, it becomes capable of handling more complex commands and returning the desired plan.

## Next step

Teams AI library quick start guide