CorePlus: A Microsoft Bot Framework v4 Template

Rubén Hinojosa Chapel

10 Dec 2019 CPOL

A MBFv4 template (Node.js and TypeScript) that will let you quickly set up a Transactional, Question and Answer, and Conversational AI chatbot

Download source - 952 KB

View source on GitHub

For millions of years, mankind lived just like the animals
Then something happened which unleashed the power of our imagination
We learned to talk

Stephen Hawking

Table of Contents

- Introduction
- Background and Requirements
- CorePlus Bot Template Features
  - Transactional Chatbot
  - Question and Answer Chatbot
  - Conversational Chatbot
  - Internationalization and Multilingual Conversation
  - Help Handling
  - Joke Handling
  - Profanity Handling
  - Support "*first user interaction*" Best Practices
  - Yes/No Synonyms for Yes/No Answerable Questions
  - Typing Indicator
  - Restart Command
  - Web Chat and Minimizable Web Chat Component
- The .bot File has been Deprecated
- A CorePlus Bot Template PoC
- Conclusion
- History

Introduction

I have always being curious about chatbots since I saw Eliza in action during a demo session at my high school. A computer magazine had published a version in Basic language and one of our computer professors typed the source code on a home computer we had at that time (don't remember if Atari or MSX). So, years later, when Microsoft started talking about their new Bot Framework in 2016, I was seduced by the exciting possibilities it opened. Then, a job opportunity arrived.

After working on a pioneering project with Microsoft Bot Framework v3, I realized the need to restart studying the platform almost from scratch. Microsoft was releasing a new version with lots of breaking changes. Actually, a completely different framework that rendered obsolete all v3 projects. BFv4 is a complete re-write of the framework with new concepts, terminology, documentation, architecture, etc. Quoting Microsoft:

> **Bot Framework SDK V4 is an evolution of the very successful V3 SDK. V4 is a major version release which includes breaking changes that prevent V3 bots from running on the newer V4 SDK.**

Microsoft has developed a number of samples to help you get started with the Bot Builder SDK v4, as well as a set of templates powered by the scaffolding tool Yeoman.

This article introduces **CorePlus**, a Microsoft Bot Framework v4 template that I have created, based on a previous version of the **Core Bot template** (Node.js) supported by the generator-botbuilder Yeoman generator. It's an extended and advanced version, intended as a quick-start for setting up a **Transactional**, **Question and Answer**, and **Conversational** chatbot, all in one, using core AI capabilities. The template proposes a modified project structure and architecture, and provides solutions for the technical and design challenges that arise.

Background and Requirements

Although some basic knowledge on Microsoft Bot Framework: Node.js SDK, LUIS, QnA Maker, Bot Framework Emulator, etc., is recommended, it's not required. The code is fully commented and the article provides lots of external links to samples, documents and other articles that can help you expand your vision and knowledge on Microsoft's framework as well as on chatbots design and development in general. Visual Studio Code is suggested as the code editor of choice. You may use any other one of your preferences, though, such as WebStorm.

The **CorePlus Bot** template is available in two language versions:

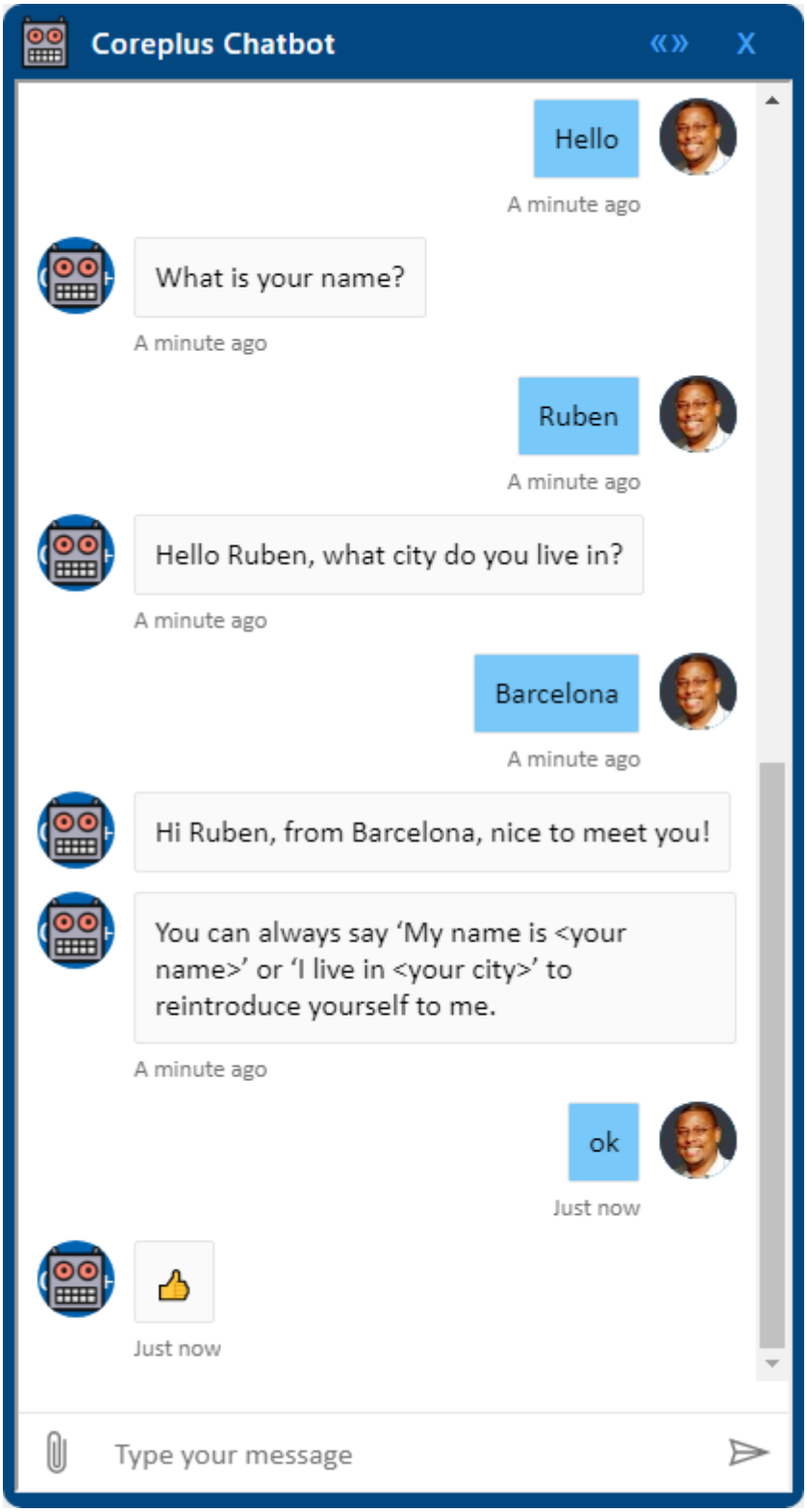
- Node.js
- TyeScript

CorePlus Bot Template Features

CorePlus supports **Transactional**, **Question and Answer**, and **Conversational** Chatbot, all in one. It's able to handle common scenarios ranging from most simple ones to advanced capabilities. Here are the built-in features that you'll get out-of-the-box after downloading and installing the code.

Transactional Chatbot

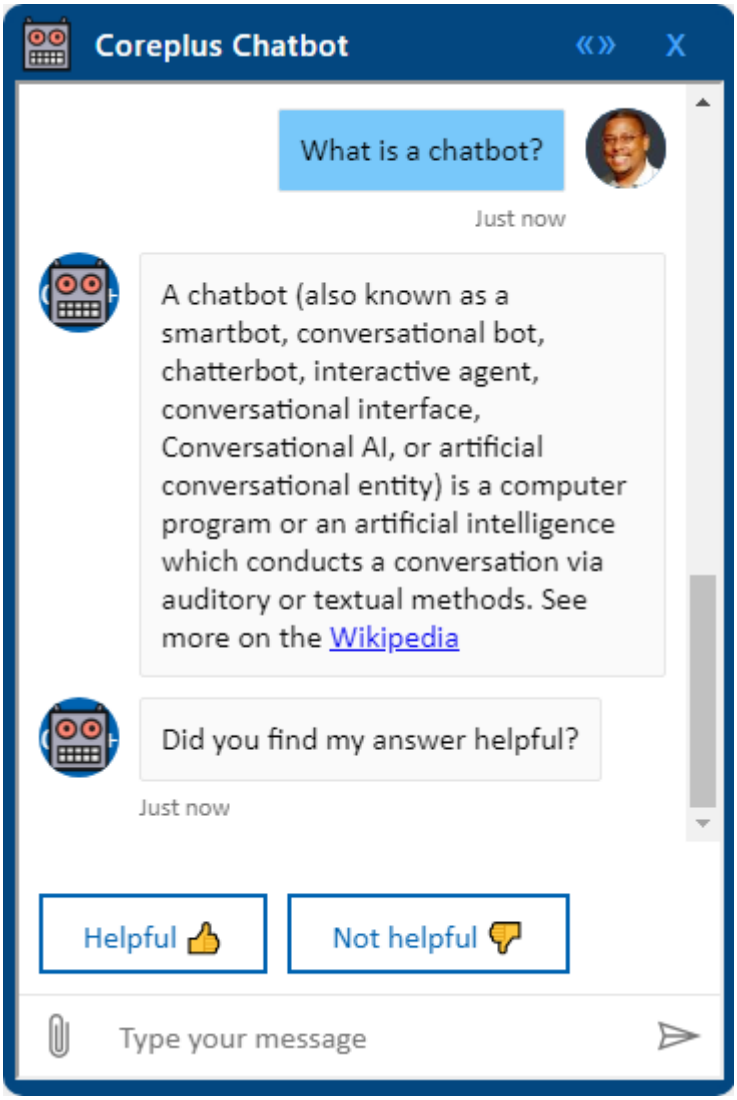
The Microsoft's **Core Bot** template, which is a template version of the core-bot sample (formerly named basic-bot), shows how to gather and validate user's information using multi-steps Waterfall dialogs. **CorePlus Bot** keeps the original `Greeting` dialog code and logic, making little refactoring and modifications, like using a localizer and a Typing indicator.



Though this is a very basic example, it provides the foundation for a transactional task-oriented chatbot, is able to understand requests and perform a bounded number of actions. A **CorePlus** based chatbot can respond to user inquiries to complete a single task at a time, in accordance with the business requirements, such as: **Show me the balance**, **Find me a flight** or **I would like a pizza**.

Question and Answer Chatbot

A **CorePlus** based chatbot can also provide informational answers to user's questions as a Question-Answering conversational system. Powered by Microsoft's QnA Maker, you can train your model and expose existing FAQ (Frequently Asked Questions) through a conversational interface. QnA Maker will classify the questions and will return the answer to a question that best matches with the user query.



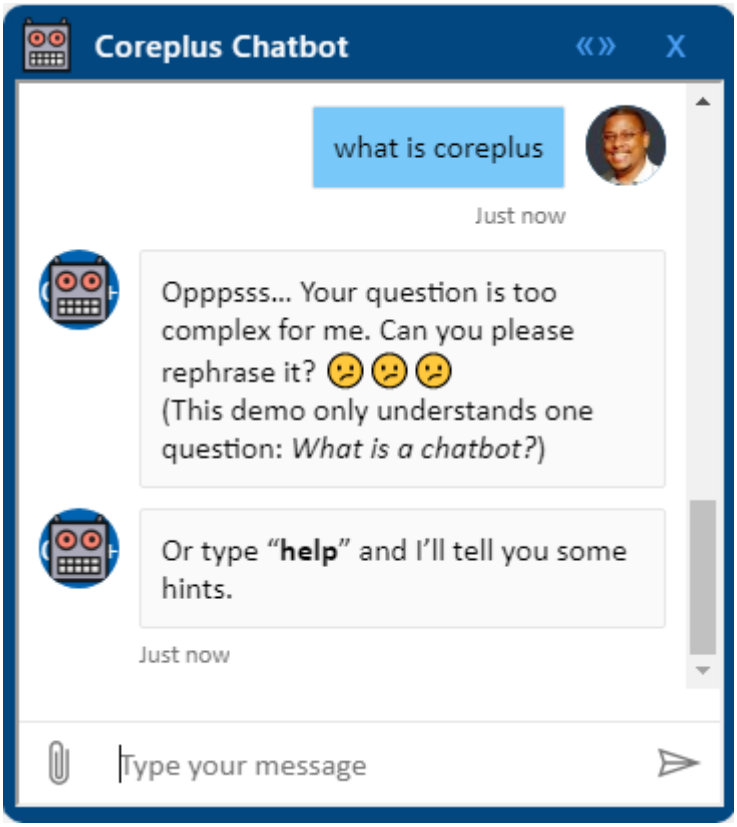
After responding to the user's question, it's a good practice to request an evaluation of the answer (**Helpful**, **Not helpful**). This feedback allows us to log telemetry data about the interaction (asked question, given answer, evaluation...) in **Application Insights**. This way, we can learn from our customers, know what they are asking, the answers they are receiving and how they are evaluated so that we can improve the performance of our bot.

> *The beauty about conversational interfaces is that your users will tell you exactly what they want and what they think of your bot* (Dashbot — Got Bot Metrics?).

So, implement your own analytics and try to extract value from the failures. Certainly, **Application Insights** already logs basic data for you, but it's unable to log custom data without specific coding, like **Helpful** or **Not helpful** user's feedback. The source code for logging custom data to **Application Insights** is out of the scope of the template, though.

On the other hand, when the user evaluates the answer as **Not Helpful** or not answer is found at all, there are a number of response options we can implement:

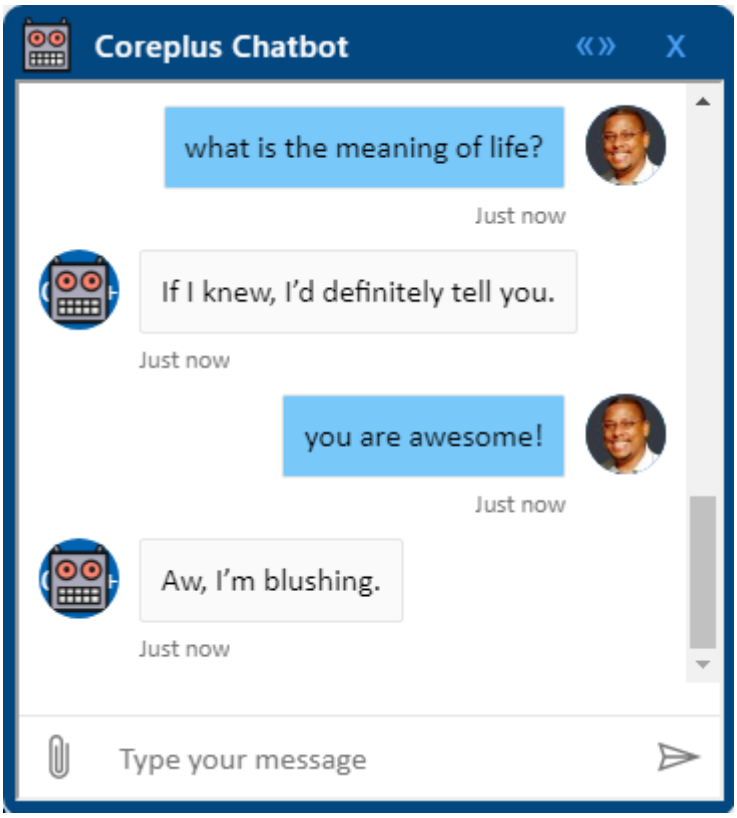
- Reply with a help text, maybe with examples of valid questions
- Look for an answer on another Knowledge Base or external QnA service
- Hand over to a human
- Suggest a link to ask the community
- Suggest an email address to contact a real person
- Ask the user to rephrase the question using other words
- Suggest the user to request help to the chatbot



You should consider, within your context, what solution best fits your business requirements. The template implements the last two ones because, in its context, they are the simpler solutions.

Conversational Chatbot

Besides completing tasks and answering common questions on a trained domain, a **CorePlus** based chatbot is able to chat with customers on random topics. That is to say, it can answer informal questions, also known as chit-chat or small talk.



Microsoft's Project Personality Chat "*easily adds small talk capabilities to a chat agent, in line with a distinct personality*". Personality Chat Datasets can be used with QnA Maker. This is the approach adopted by **CorePlus**. The template comes with a KB file which includes a personality chit-chat dataset, specifically the Friend version. Once the KB is imported into QnA Maker, you can replace the dataset or modify it in accordance with your business requirements.

Using a chit-chat dataset with QnA Maker is a wonderful solution for rapid prototyping or basic bots. If you need to handle specific scenarios in a custom way, you will have to also use LUIS. For instance, out-of-the-box chit-chat dataset takes into account the **Joke** interaction. If the user writes "*tell me a joke*", the same joke will always be returned back if you use the default chit-chat response. Nevertheless, handling this scenario with LUIS will give you the **Joke** intent. So, you will be able to create a dialog that responds with random jokes, just as **CorePlus** does.

CorePlus handles four special cases of chit-chat intents using LUIS:

- `ChitchatCancel`: The user wants to cancel an ongoing transactional dialog
- `ChitchatHelp`: The user asks for help
- `ChitchatJoke`: The user asks for a joke
- `ChitchatProfanity`: LUIS detects inappropriate terms in the user's utterance

In the context of the template, I consider chit-chat questions any user question that is not related to transactional bot capabilities nor QnA questions. This separation of concerns allows to consider special cases and return more elaborated responses. All the other ones are handled using the QnA Maker service. Some of these chit-chat intents can also be understood as Interruptions.

Internationalization and Multilingual Conversation

The **CorePlus Bot** template supports internationalization and multilingual conversation. For localization tasks, in MBFv4, we need to rely on third-party providers. I have chosen i18n, but the code is ready to use any other solution. Let's look at the most important elements.

At first, we need **locale files**. That is, JSON files holding key-value pairs, mapping message IDs to localized text `string`s. A locale file looks like:

en-US.json

```
{
    "welcome": {
        "tittle": "👋 Hello, welcome to Bot Framework!",
    },
    "readyPrompt": "What can I do for you?"
}
```

A useful feature is the ability to use object notation so that we can use keys like `'readyPrompt'` or `'welcome.tittle'` as well. In the *index.(js|ts)* file, we should import and configure the localization package:

Node.js

```
const localizer = require('i18n');
...
localizer.configure({
    defaultLocale: 'en-US',
    directory: path.join(__dirname, '/locales'),
    objectNotation: true
});
```

TypeScript

```
import * as localizer from './dialogs/shared/localizer';
// Configuration is the same as in Node.js
});
```

CorePlus also provides a simplified and unified translation function by mimicking the old session.localizer.gettext() we were used to in MBFv3.

Node.js

```
localizer.gettext = function(locale, key, args) {
    return this.__({ phrase: key, locale: locale }, args);
```

```
};
```

For the **TypeScript** version, we need to do a module augmentation inside `dialogs/shared/localizer` because TypeScript does not allow us to set `localizer.gettext = ...` We must also declare several overloads of the `gettext()` function because of the TypeScript type checking restrictions.

TypeScript

```
import * as i18n from 'i18n'

declare module 'i18n' {
    function gettext(locale: string | undefined, key: string, ...replace: string[]): string;
    function gettext(locale: string | undefined, key: string, replacements: Replacements):
string;
    function gettextarray(locale: string | undefined, key: string): string[];
    function getobject(locale: string | undefined, key: string): {};
}

(i18n.gettext as any) = function(locale: string | undefined, key: string, ...replace:
string[]): string {
    return i18n.__({ phrase: key, locale: locale }, ...replace);
};

(i18n.gettext as any) = function(locale: string | undefined, key: string, replacements:
i18n.Replacements): string {
    return i18n.__({ phrase: key, locale: locale }, replacements);
};

(i18n.gettextarray as any) = function(locale: string | undefined, key: string): string |
string[] {
    return i18n.__({ phrase: key, locale: locale });
};

(i18n.getobject as any) = function(locale: string | undefined, key: string): {} {
    return i18n.__({ phrase: key, locale: locale });
};
```

Should you want to use another localization package, implement your own version of `localizer.gettext()`. After the package has been configured, you can use it throughout your code. For instance:

WelcomeDialog/index.js

```
const localizer = require('i18n');
...
// Restart command should be localized.
const restartCommand = localizer.gettext(locale, 'restartCommand');
```

WelcomeDialog/index.ts

```
import * as localizer from '../shared/localizer';
...
// Restart command should be localized.
const restartCommand: string = localizer.gettext(locale, 'restartCommand');
```

Multilingual conversation can be achieved, at least, by two ways: using automatic translation or using by-language cognitive service instances. I have adopted the latter solution. The following piece of code, located in the *bot.(js|ts)* file, adds **LUIS** and **QnAMaker** recognizers for each locale:

Node.js

```
const luisRecognizers = {};
const qnaRecognizers = {};
```

```
const availableLocales = localizer.getLocales();

// Add LUIS and QnAMaker recognizers for each locale
availableLocales.forEach((locale) => {
    // Add LUIS recognizers
    const luisConfig = botConfig.findServiceByNameOrId(LUIS_CONFIGURATION + locale);

    if (!luisConfig || !luisConfig.appId) {
        throw new Error('Missing LUIS configuration for locale "' + locale + '".\n\n');
    }

    luisRecognizers[locale] = new LuisRecognizer({
        applicationId: luisConfig.appId,
        endpointKey: luisConfig.subscriptionKey,
        endpoint: luisConfig.getEndpoint()
    }, undefined, true);

    // Add QnAMaker recognizers
    const qnaConfig = botConfig.findServiceByNameOrId(QNA_CONFIGURATION + locale);

    if (!qnaConfig || !qnaConfig.kbId) {
        throw new Error(`QnA Maker application information not found in .bot file.\n\n`);
    }

    qnaRecognizers[locale] = new QnAMaker({
        knowledgeBaseId: qnaConfig.kbId,
        endpointKey: qnaConfig.endpointKey,
        host: qnaConfig.hostname
    }, QNA_MAKER_OPTIONS);
});
```

TypeScript

```
const luisRecognizers: LuisRecognizerDictionary = {};
const qnaRecognizers: QnAMakerDictionary = {};
const availableLocales: string[] = localizer.getLocales();

// Add LUIS and QnAMaker recognizers for each locale
availableLocales.forEach((locale) => {
    // Add LUIS recognizers
    const luisConfig: ILuisService = <ILuisService>
botConfig.findServiceByNameOrId(LUIS_CONFIGURATION + locale);

    if (!luisConfig || !luisConfig.appId) {
        throw new Error('Missing LUIS configuration for locale "' + locale + '".\n\n');
    }

    luisRecognizers[locale] = new LuisRecognizer({
        applicationId: luisConfig.appId,
        endpointKey: luisConfig.subscriptionKey,
        endpoint: luisConfig.getEndpoint()
    }, undefined, true);

    // Add QnAMaker recognizers
    const qnaConfig: IQnAService = <IQnAService>
botConfig.findServiceByNameOrId(QNA_CONFIGURATION + locale);

    if (!qnaConfig || !qnaConfig.kbId) {
        throw new Error(`QnA Maker application information not found in .bot file.\n\n`);
    }

    qnaRecognizers[locale] = new QnAMaker({
        knowledgeBaseId: qnaConfig.kbId,
        endpointKey: qnaConfig.endpointKey,
        host: qnaConfig.hostname
    }, QNA_MAKER_OPTIONS);
});
```

As you may infer from the code, we'll need per-locale configurations in the *.bot* file. So, it will look like:

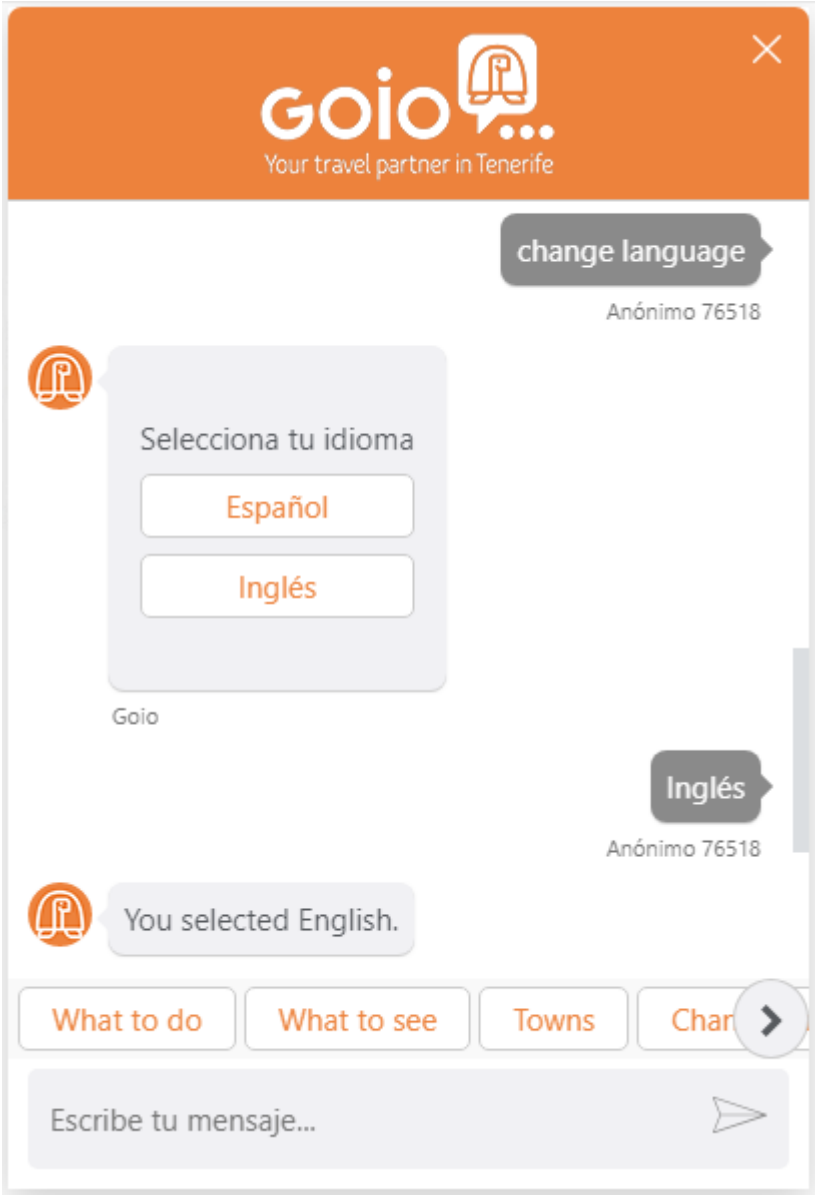
```
{
    "name": "coreplus",
    "services": [
    ...
        {
            "type": "luis",
            "name": "LUIS-en-US",
            ...
        },
        {
            "type": "qna",
            "name": "QNA-en-US",
            ...
        },
        {
            "type": "luis",
            "name": "LUIS-es-ES",
            ...
        },
        {
            "type": "qna",
            "name": "QNA-es-ES",
            ...
        }
    ],
    ...
}
```

Changing Bot's Language

If you need to develop a multilingual chatbot, you should implement your own User Experience solution for gathering the desired language. After obtaining it, all you have to do is update `UserData.locale` with a matching value in your locale files. This is the value used for retrieving localized texts, as well as for querying LUIS and QnAMaker services. Each user has their own associated locale value, which is stored in the user state.

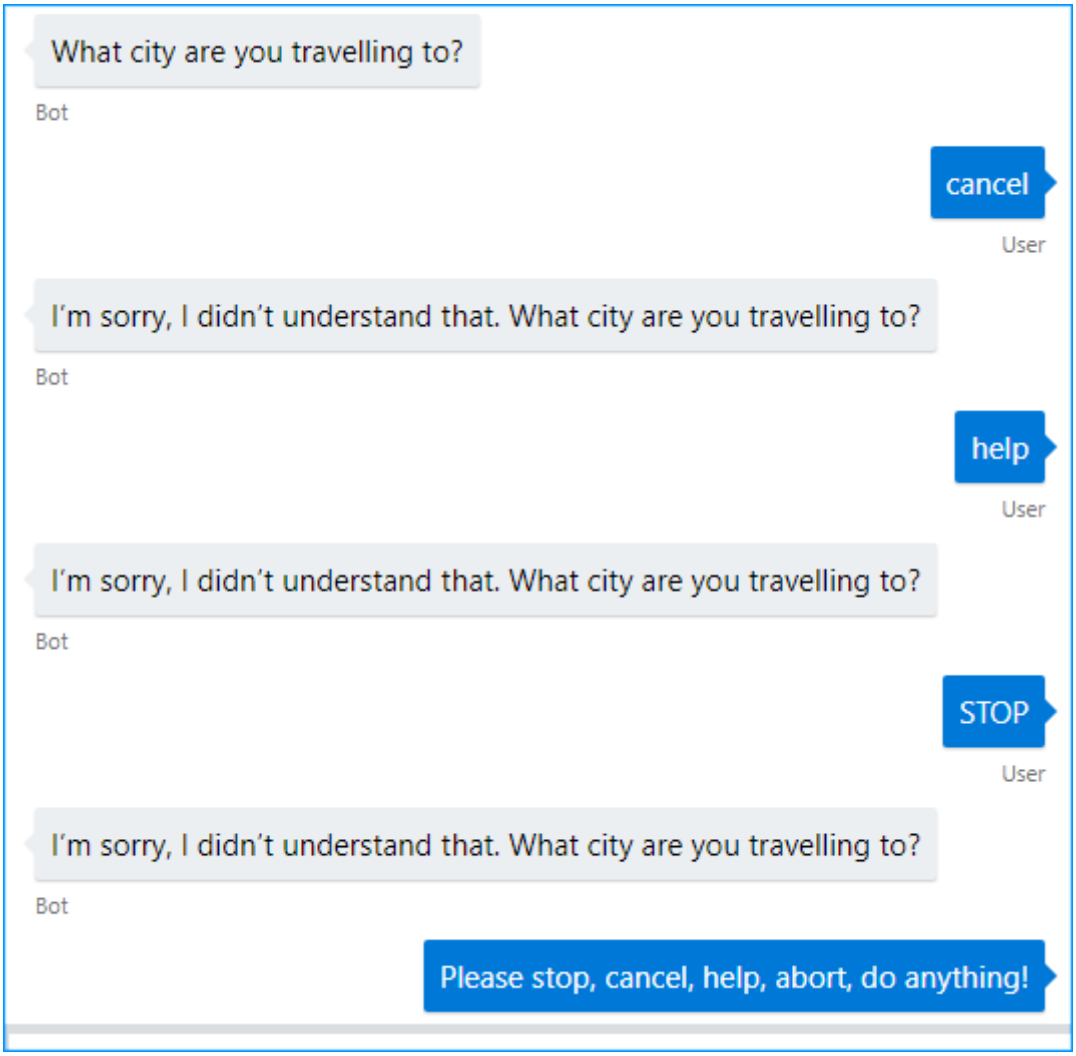
```
const userData = await this.userDataAccessor.get(step.context);
userData.locale = '<the new value>';
```

A possible solution is to train LUIS with the `ChangeLanguage` intent and bind it to a `ChangeLanguage` dialog that displays a menu to the user. This is the approach adopted by Goio, a Spanish chatbot which allows to change the language.



Cancelation and Confirmation of an Ongoing Dialog

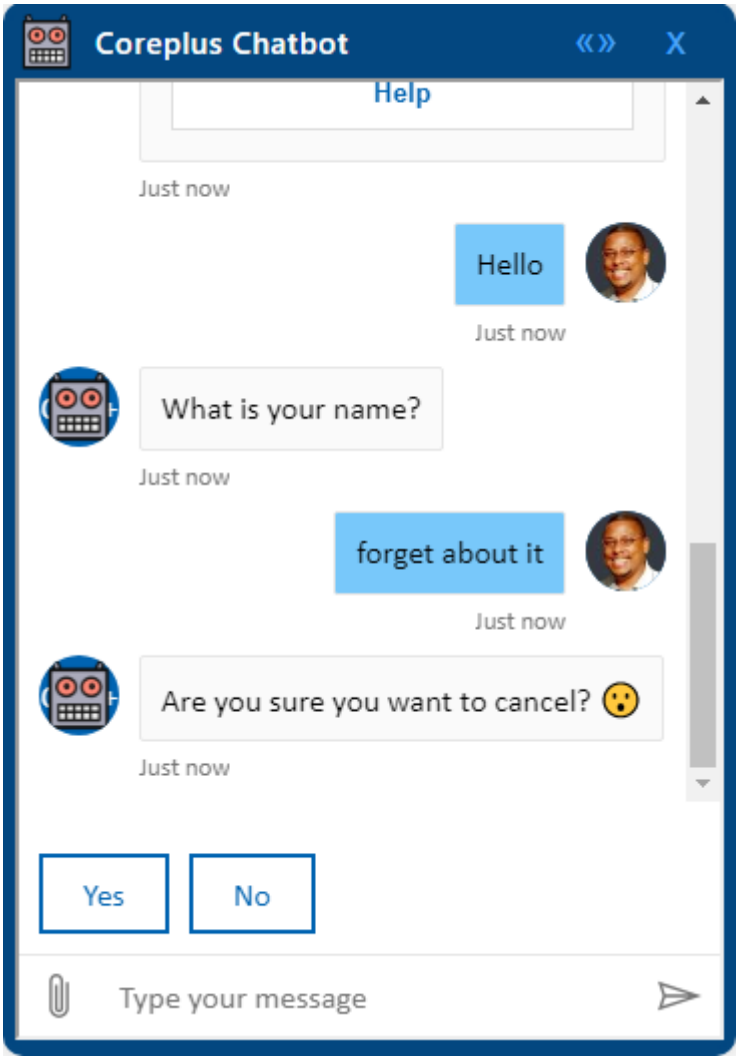
When chatting with a bot, during an ongoing transactional operation, the user may want to cancel the dialog or decide to start over.



That is why a useful recommendation is:

> **Design your bot to consider that a user might attempt to change the course of the conversation at any time.**

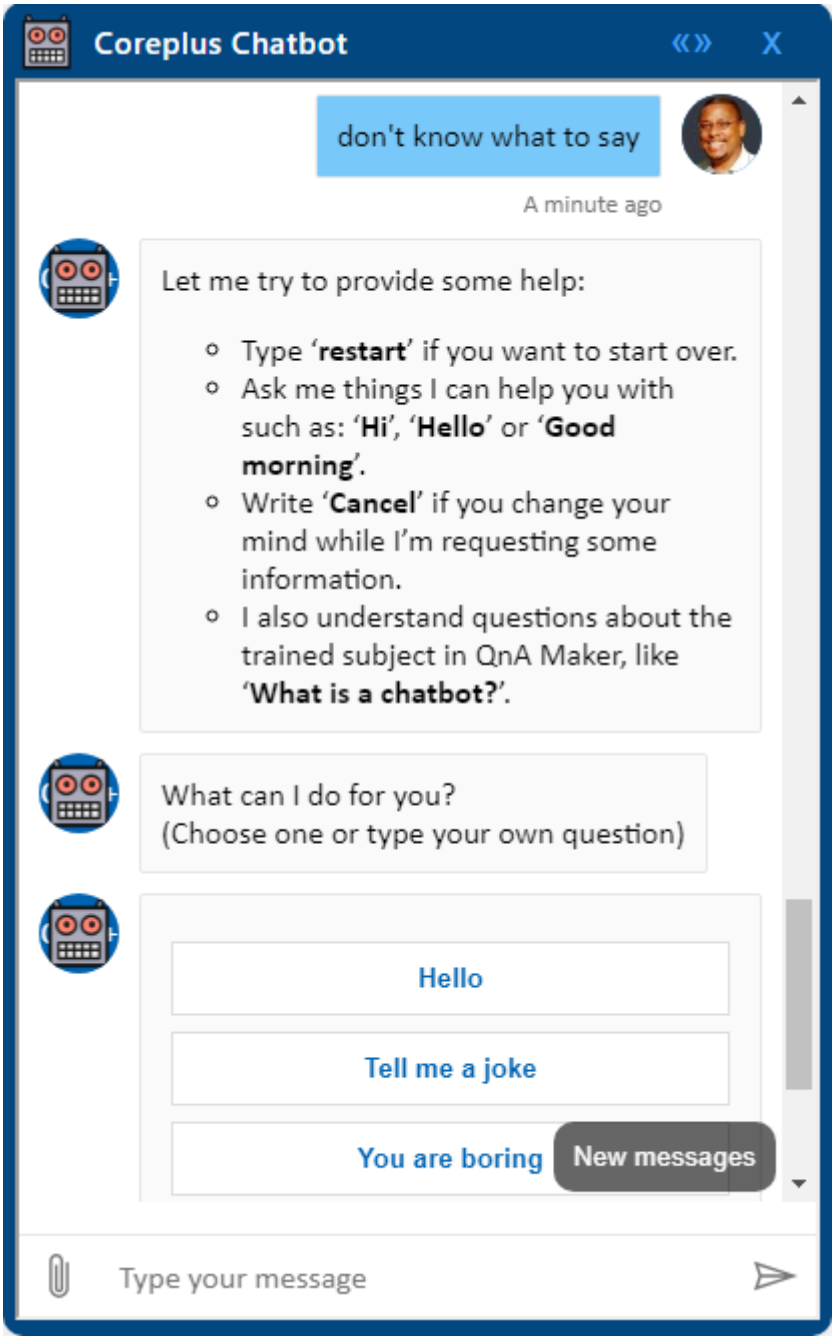
The original Microsoft's template already implements a solution to this issue. **CorePlus** keeps that solution and introduces a confirmation prompt for double checking with the user, while Microsoft's Core Bot immediately cancels the ongoing dialog once it recognizes the **Cancel** intent, without confirmation.



Help Handling

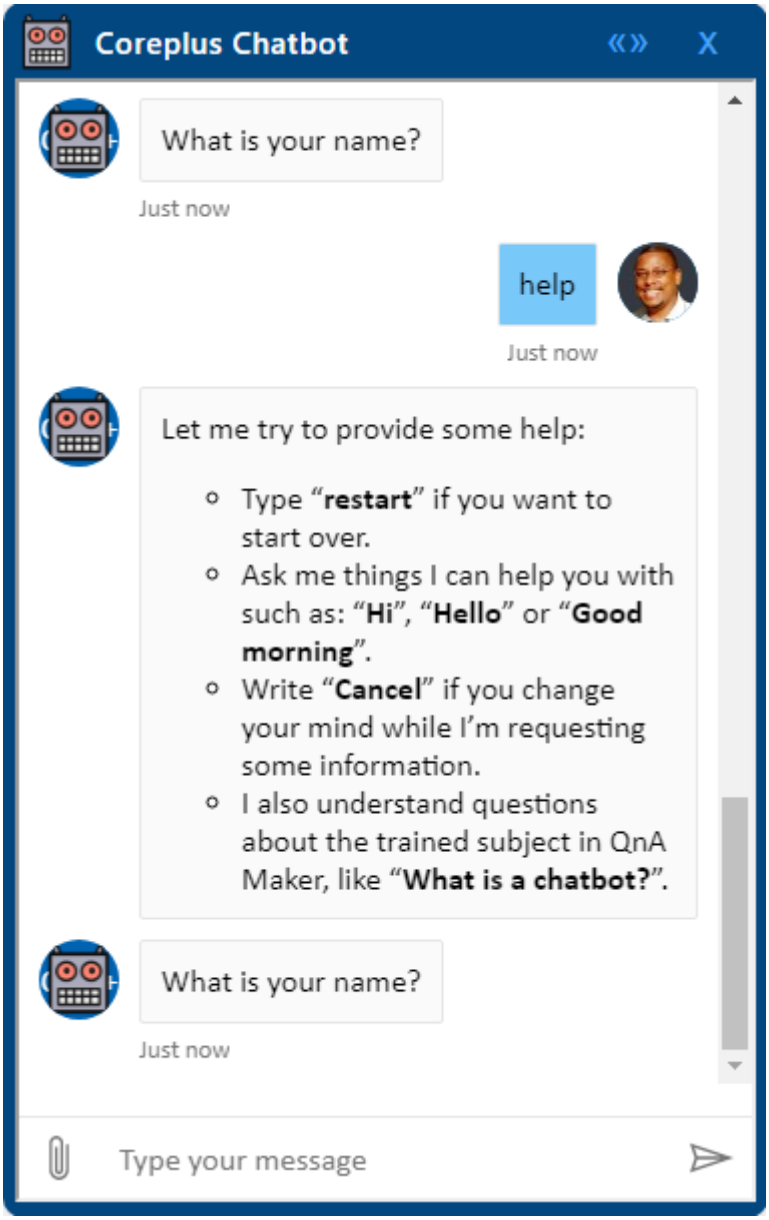
The original Core Bot template already handles the **Help** intent. **CorePlus** refactors and extends this base solution, displaying three elements:

- A list of actual utterances the user can type
- An introduction to a menu
- A menu linked to the chatbot capabilities



The list of actual utterances and menu options are intended here to show the whole set of capabilities the chatbot template exhibits. In a real production bot, this list probably won't be short, so you should design a solution that best supports the User Experience.

The `ChitchatHelp` intent is handled as an interruption so that the user can ask for help at any time:



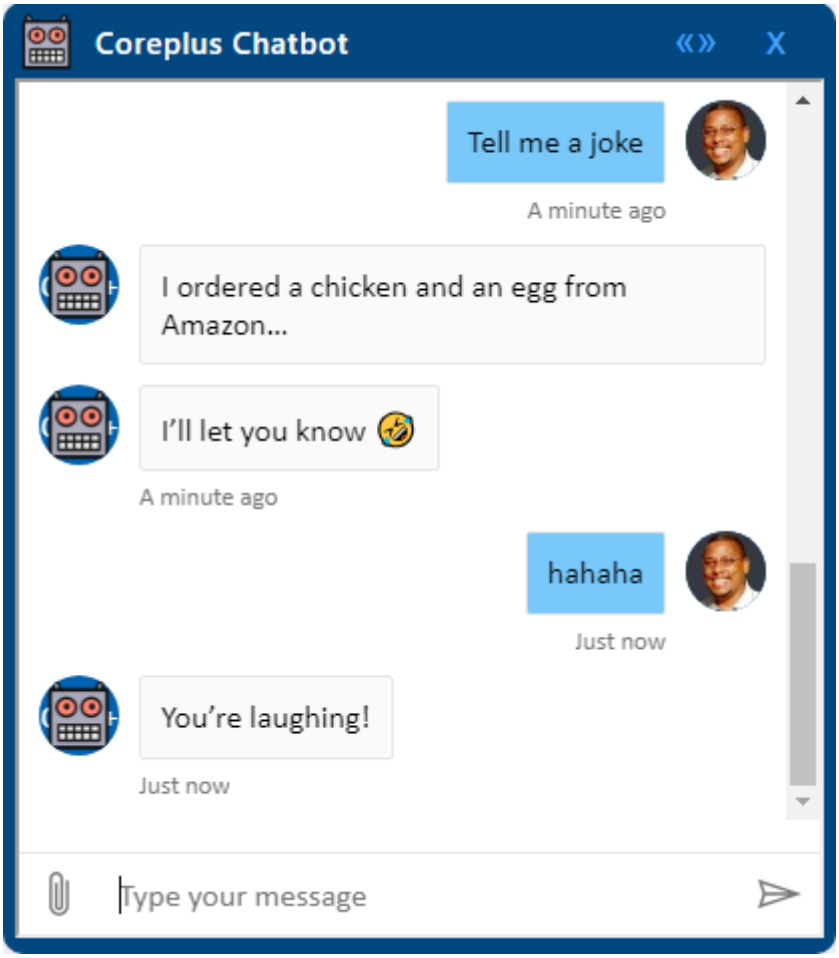
Joke Handling

Users often ask the bot for jokes. According to Arte Merritt, Co-founder and CEO of Dashbot.io, a chatbot analytics company:

> *About 12% of Facebook bots on our platform have had users ask the bot to tell a joke or say something funny.*

CorePlus supports this scenario out-of-the-box with:

- A LUIS model trained with a set of Joke utterances, including those listed by Arte Merritt in his article.
- A specialized dialog bound to the `ChitchatJoke` intent, able to return different random jokes.



Jokes are defined in the locale file. They can be as many as you consider. Multipart jokes can be split with "&&":

en-US.json

```
{
    ...
    "jokes": {
        "0": "My boss told me to have a good day so I went home 😂",
        "1": "What do you call a guy with a rubber toe? && Roberto 🤣",
        ...
        "5": "I ordered a chicken and an egg from Amazon... && I'll let you know 🤣",
        ...
    }
    ...
}
```

The last told joke is never repeated in the next iteration. Here is the dialog code, available in the *dialogs/chitchat* folder, *index.(js|ts)* file:

Node.js

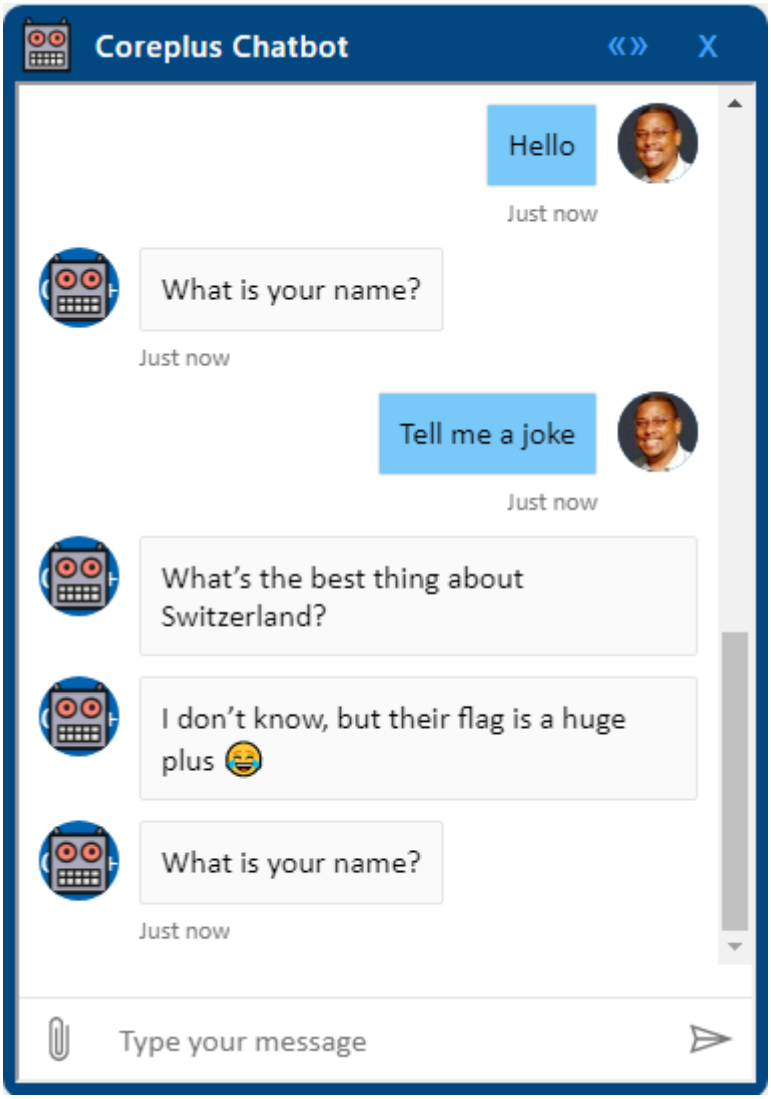
```
async jokeDialog(dc) {
    const userData = await this.userDataAccessor.get(dc.context);

    // Retrieve the jokes list.
    const jokes = localizer.gettext(userData.locale, 'jokes');

    // Randomly select a joke from the list. Do not repeat the last one.
    let jokeNumber;
    do {
        jokeNumber = Utils.getRandomInt(0, Object.keys(jokes).length).toString();
    } while (jokeNumber === userData.jokeNumber);
```

```
    // Save the last joke number.
    userData.jokeNumber = jokeNumber;
    await this.userDataAccessor.set(dc.context, userData);

    const parts = jokes[jokeNumber].split('&&');

    // Send every joke part in a different bubble, leaving a short space time between them.
    for (let i = 0; i < parts.length; i++) {
        await Utils.sendTyping(dc.context);
        await dc.context.sendActivity(parts[i]);
    }

    return true;
}
```

TypeScript

```
async jokeDialog(dc: DialogContext): Promise<boolean> {
    const userData: UserData = await this.userDataAccessor.get(dc.context,
UserData.defaultEmpty);
    const locale: string = userData.locale || localizer.getLocale();

    // Retrieve the jokes list.
    const jokes: StringDictionary = localizer.getobject(locale, 'jokes');

    // Randomly select a joke from the list. Do not repeat the last one.
    let jokeNumber: string;
    do {
        jokeNumber = Utils.getRandomInt(0, Object.keys(jokes).length).toString();
    } while (jokeNumber === userData.jokeNumber);

    // Save the last joke number.
    userData.jokeNumber = jokeNumber;
    await this.userDataAccessor.set(dc.context, userData);

    const parts: string[] = jokes[jokeNumber].split('&&');

    // Send every joke part in a different bubble, leaving a short space time between them.
    for (let i = 0; i < parts.length; i++) {
        await Utils.sendTyping(dc.context);
        await dc.context.sendActivity(parts[i]);
    }

    return true;
}
```

The `ChitchatJoke` intent is handled as an interruption so that the user can request a joke at any time:

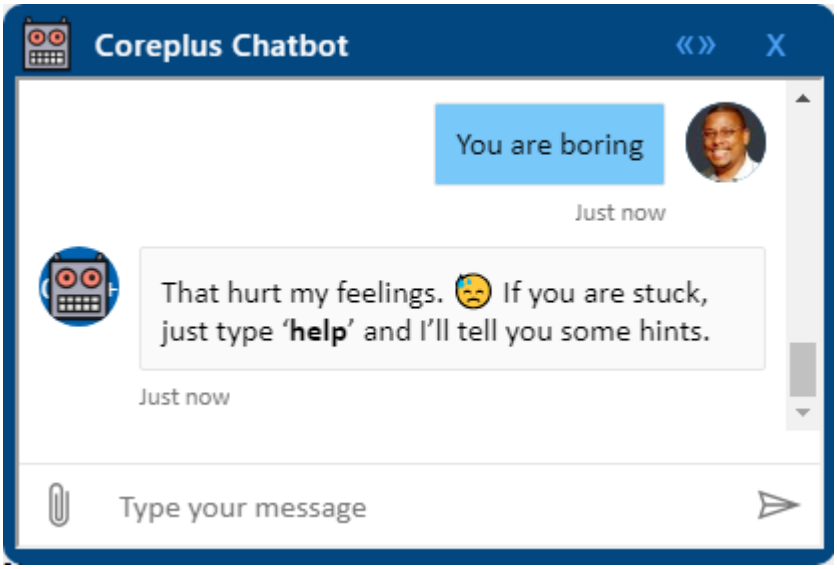


Be cautious when designing jokes for chatbots, as there are some key points to take into account:

> *Humor is subjective and ripe for missteps. One person's laugh is another person's cringe (...). The complexity of what's considered funny is enough to prompt many companies into developing bland personality-free chatbots. But creating a chatbot without humor would defeat the very purpose of most chatbots: creating a human-seeming conversation that people want to have. The trick, then, is finding a balance in creating a conversation that can build bonds and entertain users, but not offend or alienate.* (How Funny Should a Chatbot Be?)

Profanity Handling

It's a known fact that when people have conversations with chatbots, being aware they talk to computers, the occurrences of profanity is greater than when they talk to other humans, including sexually explicit terms and bad words. **CorePlus** comes with built-in profanity recognition and handling.



As was previously exposed, **CorePlus** uses LUIS and the `ChitchatProfanity` intent for filtering unwanted content. Once such content is recognized, the chatbot responds accordingly, suggesting the user to ask for help. Here is the code, available in the *dialogs/chitchat* folder, *index.(js|ts)* file:

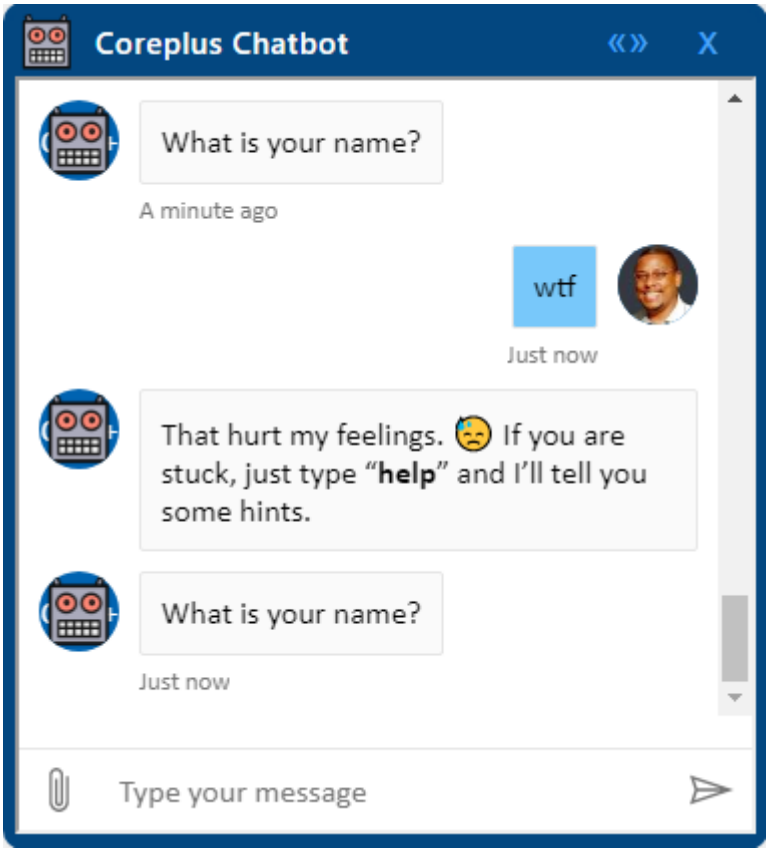
Node.js

```
async profanityDialog(dc) {
    const userData = await this.userDataAccessor.get(dc.context);
    const msg = localizer.gettext(userData.locale, 'profanity');
    await dc.context.sendActivity(msg);

    return true;
}
```

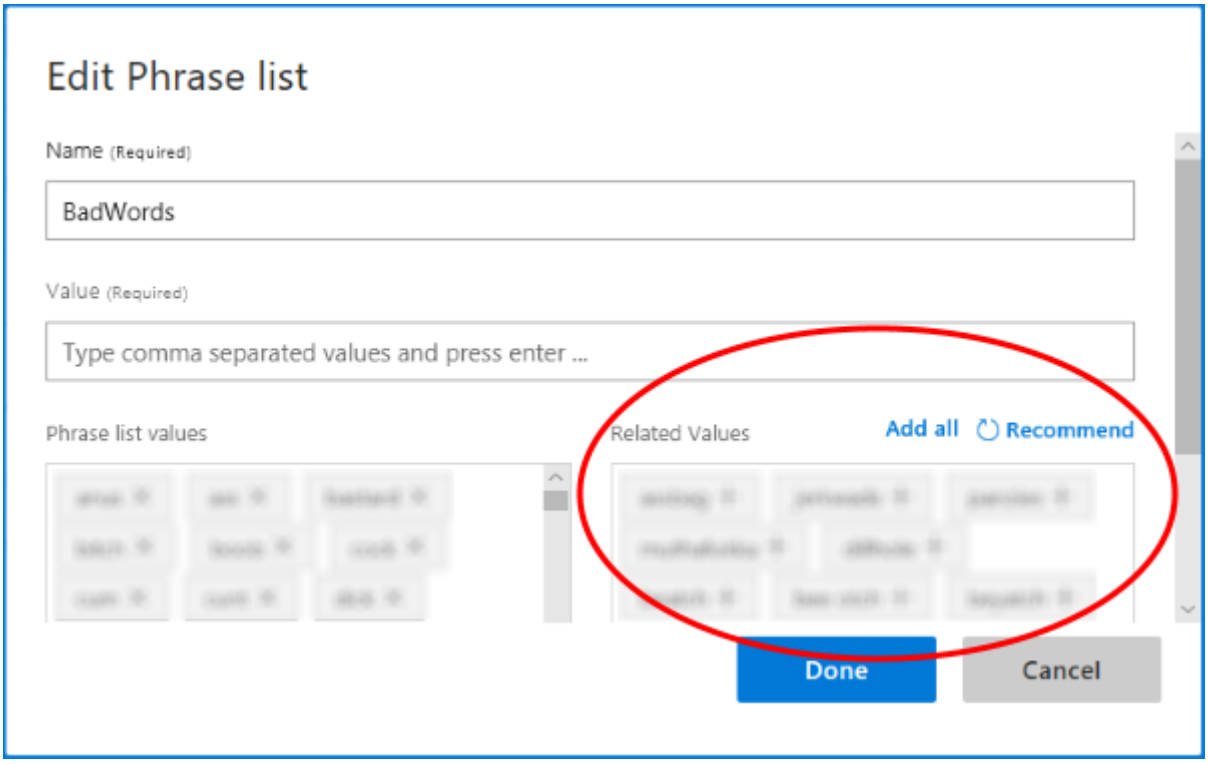
TypeScript

```
async profanityDialog(dc: DialogContext): Promise<boolean> {
    const userData: UserData = await this.userDataAccessor.get(dc.context,
UserData.defaultEmpty);
    const locale: string = userData.locale || localizer.getLocale();
    const msg: string = localizer.gettext(locale, 'profanity');
    await dc.context.sendActivity(msg);

    return true;
}
```

The `ChitchatProfanity` intent is handled as an interruption so that it can be recognized at any time:



The LUIS model is trained with basic phrases such as: "*you're awful*", "*go to hell*" and "*ur stupid*", among others much more offensive and rude. It's also trained with the swear words collected by the BanBuilder project, the Bad Words and Top Swear Words Banned by Google list and the **Related Values** suggested by the LUIS interactive interface.



You may complete or modify the training with the use cases you want to consider as Profanity. An alternative solution for handling inappropriate content and undesirable text is to use a specialized service such as the Content Moderator.

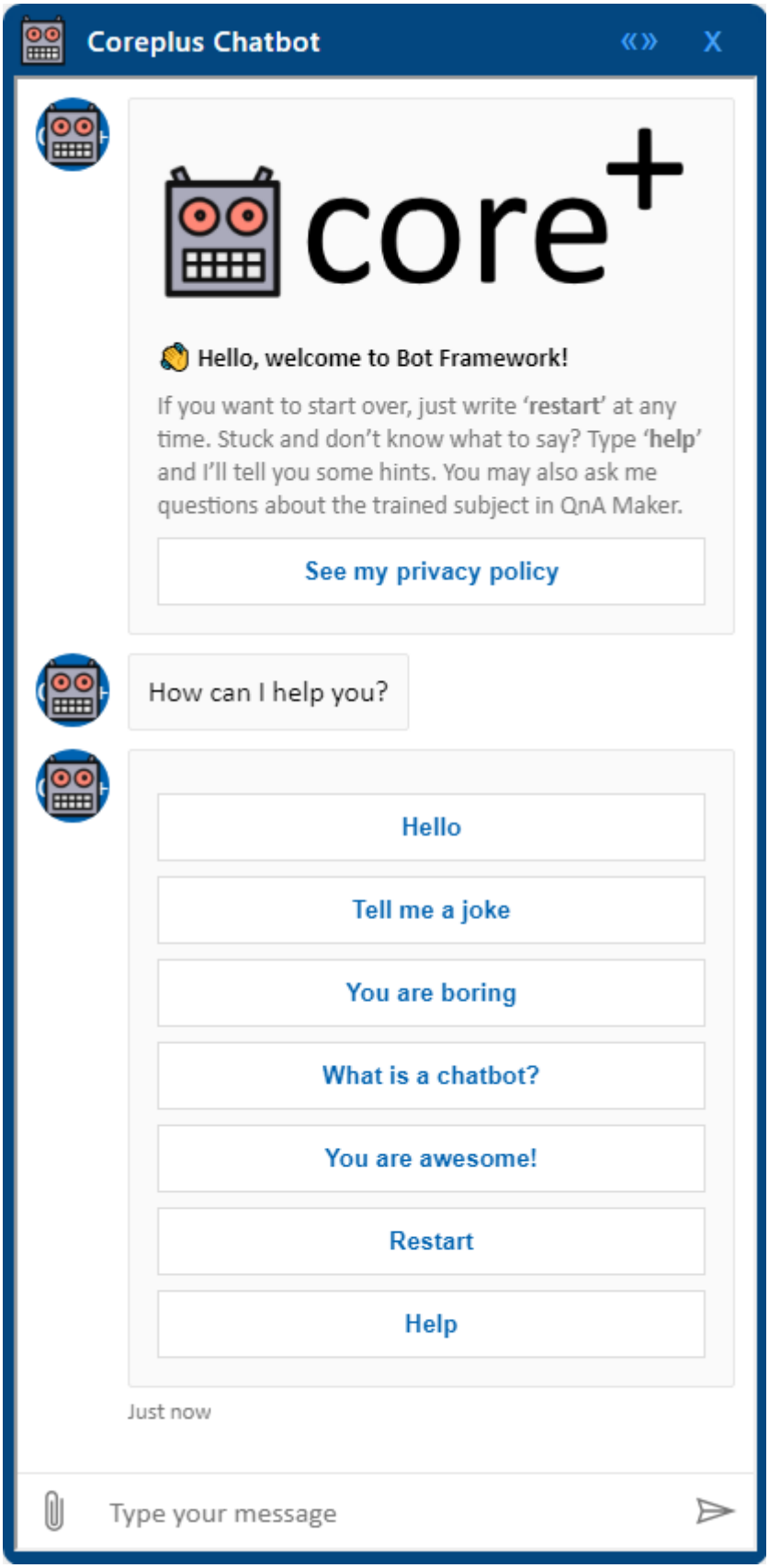
Support "*First User Interaction*" Best Practices

The onboarding interaction is of paramount importance to the success of a chatbot, so that a careful design is critical to the user experience.

> *The very first interaction between the user and bot is critical to the user experience. When designing your bot, keep in mind that there is more to that first message than just saying "hi." When you build an app, you design the first screen to provide important navigation cues. Users should intuitively understand things such as where the menu is located and how it works, where to go for help, what the privacy policy is, and so on. When you design a bot, the user's first interaction with the bot should provide that same type of information.* (Design a bot's first user interaction)

CorePlus comes with placeholders that enforce some UX best practices:

- Set user's expectation by providing bot capabilities, that is, what the bot can do
- Inform about how to get help, cancel an ongoing operation or start over
- Show actual examples of what the user can say
- Display a menu with 3–5 key capabilities
- Provide access to privacy policy and terms of use



As was explained in "**Help handling**", *the list of menu options is intended here to show the whole set of capabilities the chatbot template exhibits. In a real production bot, this list probably won't be short, so you should design a solution that best supports the User Experience*. A widely accepted rule of thumb is to only show 3–5 key capabilities.

CorePlus has a consistent way to show the main menu. There are three places or moments where it is displayed:

- At the end of the welcome dialog
- After the user cancels an ongoing transactional dialog
- At the end of the help dialog, when the root dialog (`MainDialog`) is the active one.

"*Bot menu actions/commands should be always invokable, regardless of the state of the conversation or the dialog the bot is in*". During an ongoing transactional dialog, if the user asks for help, we should provide some guidance. Nevertheless, if we show the

main menu here, at this very moment, we'll be distracting the user from the current task by offering parallel tasks that are not handled as interruptions. That is why the main menu is only displayed from the root dialog.

Here is the code for showing the main menu from any dialog:

Node.js

```
const { Utils } = require('../shared/utils');
...
await Utils.showMainMenu(context, locale);
```

TypeScript

```
import { Utils } from '../shared/utils';
...
await Utils.showMainMenu(context, locale);
```

While the main menu code, located in the *dialogs/shared* folder, *utils.(js|ts)* file is:

Node.js

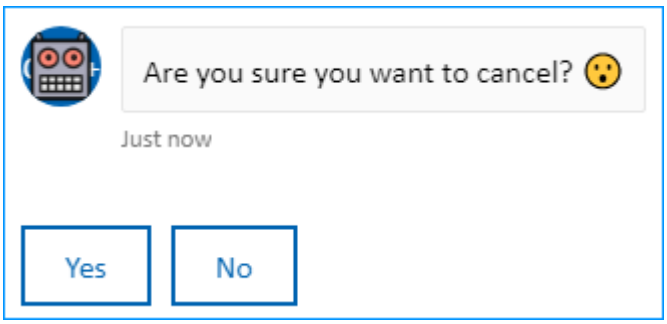
```
static async showMainMenu(context, locale) {
    const hints = localizer.gettext(locale, 'hints');
    const buttons = [];

    Object.values(hints).forEach(value => {
        buttons.push(value);
    });

    await context.sendActivity(this.getHeroCard(buttons));
}
```

TypeScript

```
static async showMainMenu(context: TurnContext, locale: string | undefined): Promise<void> {
    const hints: StringDictionary = localizer.getobject(locale, 'hints');
    const buttons: string[] = [];

    Object.values(hints).forEach(value => {
        buttons.push(value);
    });

    await context.sendActivity(this.getHeroCard(buttons, ''));
}
```

The `showMainMenu()` function loads a `hints` object from the locale file, fills an array with its values and passes it to a function, `getHeroCard()`, also in the same *utils.js* file, which returns a `HeroCard` (the menu) with the options in it.

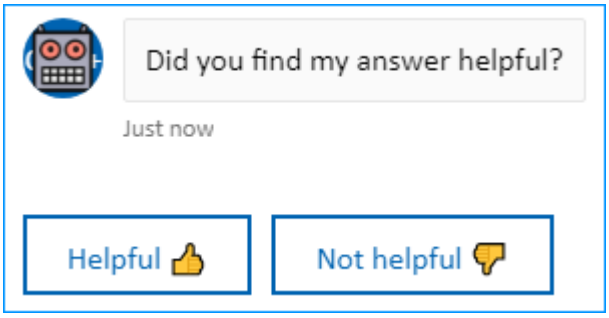
Yes/No Synonyms for Yes/No Answerable Questions



The Bot Framework v4 SDK comes with built-in specialized dialogs classes for managing conversations. One of them, which "*Prompts a user to confirm something with a "yes" or "no" response*", is the ConfirmPrompt class. As a specialized component, this class does a good job at a basic level. At the time of this writing, there are a couple of tasks I've found hard or impossible to do, though:

- Buttons with custom texts
- Recognize a wide number of utterances as synonyms of **Yes** or **No**

I came up with a solution by using a similar and more generic dialog class: ChoicePrompt. Both classes accept a Choice list, by the way, and its use should be interchangeable.



The idea is to implement a pair of functions which return a Choice object: one with "**Yes**" data and the other one... well, you may guess, with "**No**" data. For instance, the "**Yes**" version follows, located in the *dialogs/shared* folder, *utils.(js|ts)* file.

Node.js

```
static getChoiceYes(locale, titleKey, moreSynonymsKey) {
    const title = localizer.gettext(locale, titleKey);
    let yesSynonyms = localizer.gettext(locale, 'synonyms.yes');

    if (moreSynonymsKey) {
        const moreSynonyms = localizer.gettext(locale, moreSynonymsKey);
        yesSynonyms = yesSynonyms.concat(moreSynonyms);
    }

    return {
        value: 'yes',
        action: {
            type: 'imBack',
            title: title,
            value: title
        },
        synonyms: yesSynonyms
    };
}
```

TypeScript

```
static getChoiceYes(locale: string | undefined, titleKey: string, moreSynonymsKey?: string):
Choice {
    const title: string = localizer.gettext(locale, titleKey);
    let yesSynonyms: string[] = localizer.gettextarray(locale, 'synonyms.yes');

    if (moreSynonymsKey) {
        const moreSynonyms: string[] = localizer.gettextarray(locale, moreSynonymsKey);
        yesSynonyms = yesSynonyms.concat(moreSynonyms);
    }

    return {
        value: 'yes',
        action: {
            type: 'imBack',
            title: title,
            value: title
        },
```

```
        synonyms: yesSynonyms
    };
}
```

Yes/No `synonyms` are defined in the locale JSON file as arrays of `string`s. The prompt dialog will recognize any of them, as well as the button tittle and little variations.

en-US.json

```
{
    ...
    "synonyms": {
        "yes": ["yes", "y", "yeah", "yay", "👍", "awesome", "great",
        "cool", "sounds good", "works for me", "bingo", "go ahead",
            "yup", "yes to that", "you're right", "that was right",
            "that was correct", "that's accurate", "accurate", "ok",
            "yep", "that's right", "that's true", "correct",
            "that's right", "that's true", "sure", "good", "confirm", "thumbs up"],
        "no": ["no", "n", "nope", "👎", "ko", "uh-uh", "nix", "nixie",
               "nixy", "nixey", "nay", "nah", "no way",
            "negative", "out of the question", "for foul nor fair",
            "not", "thumbs down", "pigs might fly", "fat chance",
            "catch me", "go fish", "certainly not",
            "by no means", "of course not", "hardly"],
        "cancel": ["cancel", "abort"]
    },
    ...
}
```

And here is how to build a custom Confirm dialog using the `Choice` functions:

Node.js

```
const { Utils } = require('../shared/utils');
...
    async promptFeedbackStep(step) {
        const userData = await this.userDataAccessor.get(step.context);
        const locale = userData.locale;

        const prompt = localizer.gettext(locale, 'qna.requestFeedback');
        const choiceYes = Utils.getChoiceYes(locale, 'qna.helpful');
        const choiceNo = Utils.getChoiceNo(locale, 'qna.notHepful');

        return await step.prompt(ASK_FEEDBACK_PROMPT, prompt, [choiceYes, choiceNo]);
    }
```

TypeScript

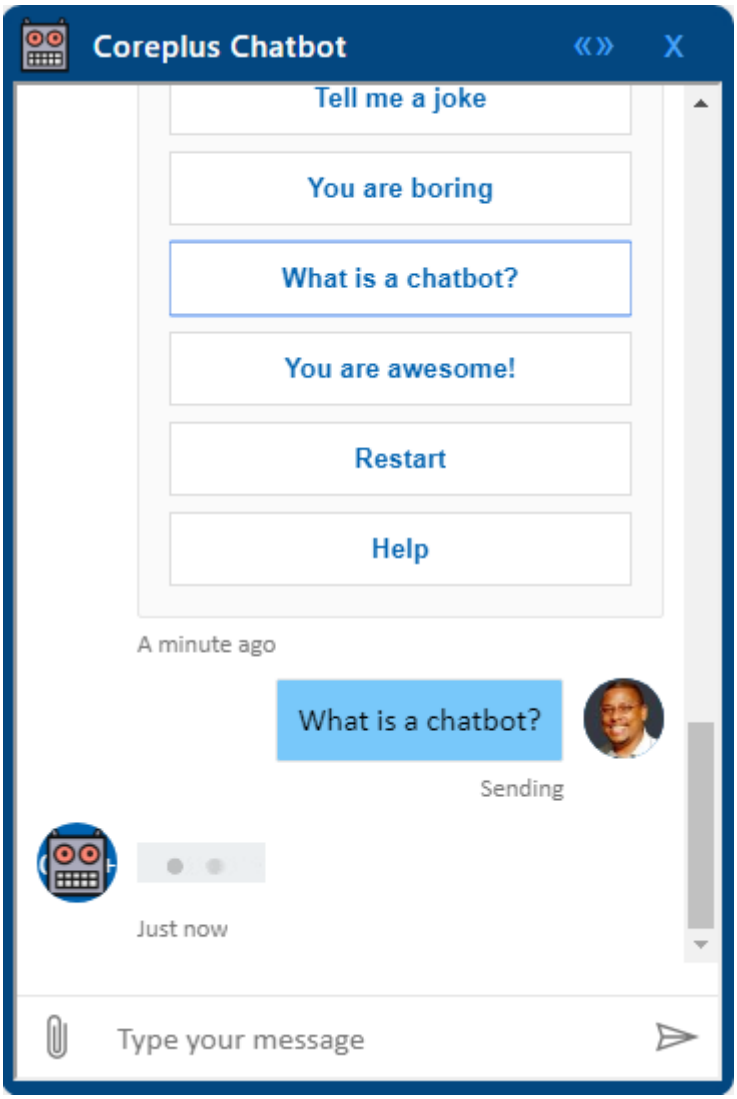
```
import { Utils } from '../shared/utils';
...
    async promptFeedbackStep(step: WaterfallStepContext): Promise<DialogTurnResult> {
        const userData: UserData = await this.userDataAccessor.get(step.context,
UserData.defaultEmpty);
        const locale: string = userData.locale || localizer.getLocale();

        const prompt: string = localizer.gettext(locale, 'qna.requestFeedback');
        const choiceYes: Choice = Utils.getChoiceYes(locale, 'qna.helpful');
        const choiceNo: Choice = Utils.getChoiceNo(locale, 'qna.notHepful');

        return await step.prompt(ASK_FEEDBACK_PROMPT, prompt, [choiceYes, choiceNo]);
    }
```

For the sake of the UX and usability, it's really important that your bot be able to understand more utterances than the ones displayed in the buttons. Be aware that users don't use bots like apps — they want to chat, not click on buttons. As a rule of thumb, a hybrid approach to chatbots that utilizes both NLP and buttons is recommended.

Typing Indicator

For the chatbot UX, the Typing indicator is as important as the Busy or Loading indicators for the web and mobile apps. It tells the user that something is happening on the other side of the screen. That the bot has listened to the user's request and now is "thinking" and elaborating the expected response.

> *Design your bot to immediately acknowledge user input, even in cases where the bot may take some time to compile its response. (...) By immediately acknowledging the user's input, you eliminate any potential for confusion as to the state of the bot. If your response takes a long time to compile, consider sending a "typing" message to indicate your bot is working. (Design bot navigation - The "mysterious bot")*



Typing indicator can also be useful to increase the wait time of additional messages and give users more time to read them when a long message is split into shorter ones.

At the time of writing this article, the Node.js version of the Bot Framework v4 lacks a built-in feature to send Typing indicator at will, when and where the developer decides to send the event. It can be implemented with just a few lines of code, though. **CorePlus** provides the feature out-of-the-box. You may find the implementation, which is self-explanatory, located in the *dialogs/shared/utils.(js|ts)* file.

Node.js

```
static async sendTyping(context) {
    await context.sendActivities([
        { type: 'typing' },
        { type: 'delay', value: this.getRandomInt(1000, 2200) }
```

```
    ]);
}
```

TypeScript

```
static async sendTyping(context: TurnContext): Promise<void> {
    await context.sendActivities([
        { type: 'typing' },
        { type: 'delay', value: this.getRandomInt(1000, 2200) }
    ]);
}
```

Here is an example for sending a Typing indicator message:

Node.js

```
// Import Utils for sendTyping() function
const { Utils } = require('../shared/utils');
...
    async promptForNameStep(step) {
        await Utils.sendTyping(step.context);
        // Do more things
    }
```

TypeScript

```
// Import Utils for sendTyping() function
import { Utils } from '../shared/utils';
...
    async promptForNameStep(step: WaterfallStepContext): Promise<DialogTurnResult> {
        await Utils.sendTyping(step.context);
        // Do more things
    }
```

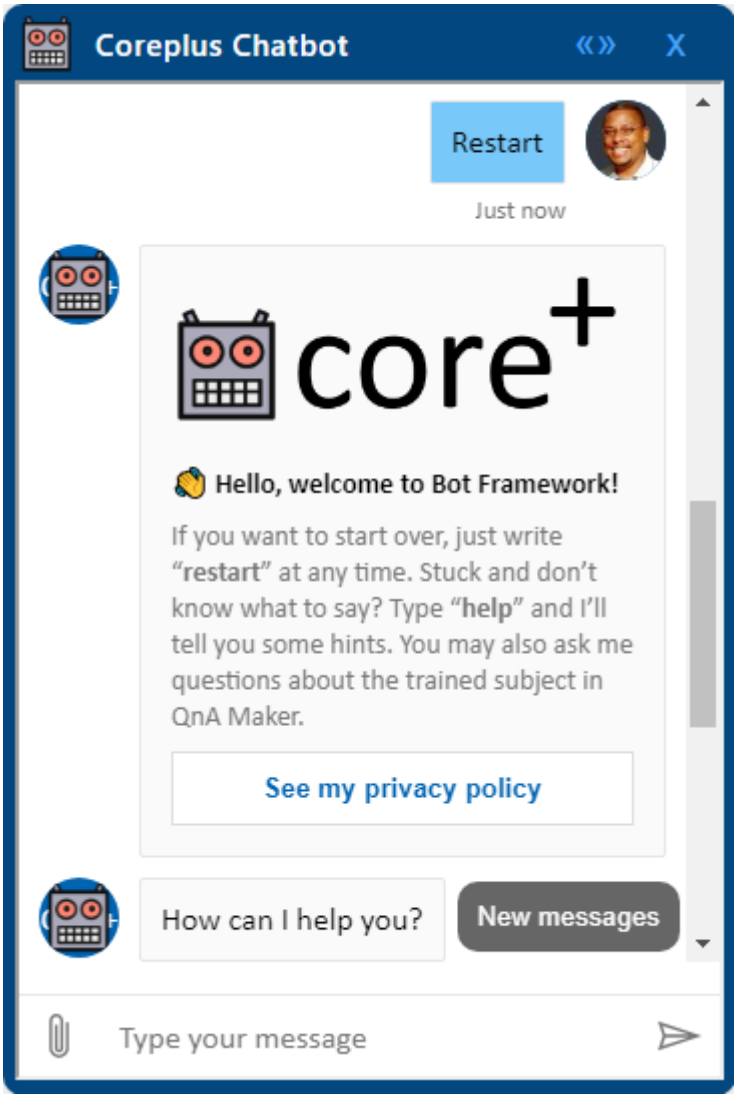
The current Typing indicator animation is `static`, it's always the same. Perhaps we'll see a more dynamic implementation in the near future because Microsoft has filed a patent that describes such improvement:

> *Technology is disclosed herein that improves the user experience with respect to is-typing animations. In an implementation, a near-end client application receives an indication that a user is typing in a far-end client application. The near-end client application responsively selects an animation that is representative of a typing pattern. The selection may be random in some implementations (or pseudo random), or the selection may correspond to a particular typing pattern. The near-end client then manipulates the ellipses in its user interface to produce the selected animation.*

Restart Command

Sometimes, the user may feel trapped in the conversation and wants to start over from scratch. So, the chatbot should provide a way to tackle this situation. A common solution is to implement a **Restart** command ("`restart`", "`start over`", "`reset`", etc.) that tells the chatbot to resend the Welcome message and repeat the "first user interaction", which may also involve clearing some gathered user data. The availability of the **Restart** command should be informed earlier, in the Welcome dialog, as well as in the Help dialog, so that the user is aware of that superpower. We have previously seen examples of both.

CorePlus provides built-in logic by handling a restart keyword, that can be modified at will in accordance with your business requirements. A more flexible approach would be to use LUIS and intent recognition with a number of different utterances. Nevertheless, intent recognition always has a variable grade of confidence, while the confidence of a single keyword is 1. If the user types **restart**, we can assume he or she wants to start over the conversation.



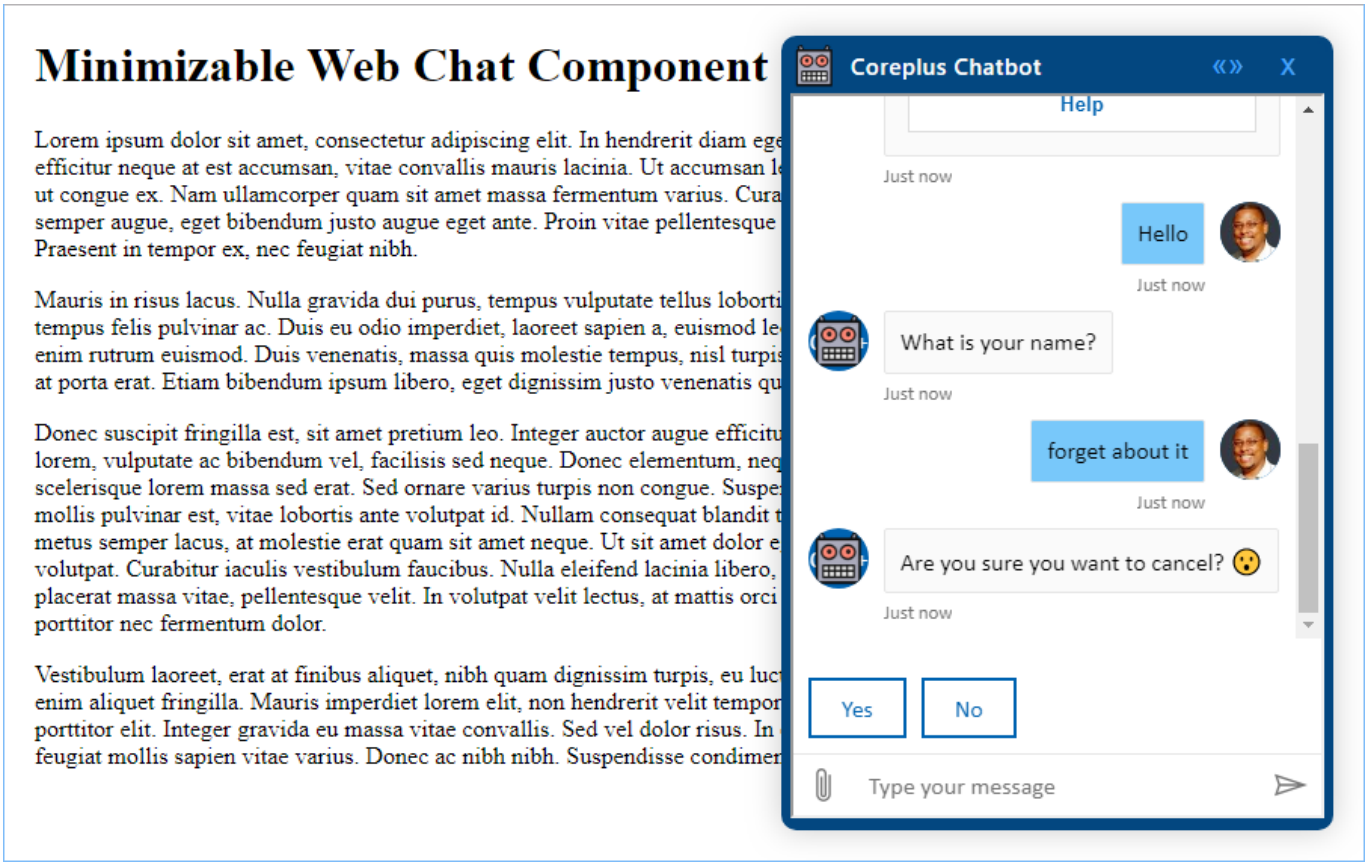
The **Restart** command is handled in the root dialog (`MainDialog`), located in the *dialogs/main/index.(js|ts)* file. Note that the command should be localized.

```
...
        // Handle commands
        if (utterance === localizer.gettext(locale, 'restartCommand')) {
            let userData = new UserData();
            // Save locale and any other data you need to persist between resets
            userData.locale = locale;
            await this.userDataAccessor.set(dc.context, userData);
            await dc.cancelAllDialogs();
            turnResult = await dc.beginDialog(WelcomeDialog.name);
        }
...
```

Web Chat and Minimizable Web Chat Component

Based on the MBFv4 Web Chat samples, this template comes with a ready-to-use custom version that shows how to add a Web Chat control to your website. The sample is available in the *public* folder, *webchat.html* file. You need to replace **YOUR_DIRECT_LINE_TOKEN** with your bot secret key. The HTML file can be straight open with your preferred internet browser or can be loaded from *http://localhost:3978/public/webchat.html* once your bot is running locally on your computer.

On the other hand, the template also comes with a minimizable version of the Web Chat component, located in the *webchat* folder, *minwebchat.html* file.



This sample takes some ideas from the customization-minimizable-web-chat official sample. All the logic is packaged into a single HTML file, so there is no need to use React. There, you will find two iframes: one will point to your bot url, remotely hosted in Azure (*https://**your-bot-handle**.azurewebsites.net/public/webchat.html*). The other one will point to the locally running instance (*http://localhost:3978/public/webchat.html*). In both cases, the previous Web Chat file will be loaded. The two iframes are meant to work one instead of the other, so you are able to decide which one to use and test.

Note that, for the local version, there is a tricky way of loading the iframe content to override the cached version by the browser. This forces the browser to re-fetch the iframe content each time the web page is loaded.

```
<iframe id='botiframe' src='' style='width: 100%; height: 100%;'></iframe>
<script>
  const iframe = document.getElementById('botiframe');
  iframe.src = 'http://localhost:3978/public/webchat.html?r=' + new Date().getTime();
</script>
```

The .bot File has been Deprecated

With the 4.3 release of the Bot Framework, Microsoft moved away from using *.bot* files. **The .bot file has been deprecated**. The new recommended solution for managing resources is to use ***appsettings.json*** *or* ***.env*** file instead of the ***.bot*** file. So, I'll be updating the project in the near future accordingly.

A CorePlus Bot Template PoC

Want to see a functional Proof of Concept created with this template? Watch a video of **HAL Fintech Chatbot**, a Personal Finance Management assistant that can track income and expenses. It can also retrieve information from the previous transactions on the following concepts:

- Account balance
- Income and Expenses data
- Largest income and expenses
- Budget evaluation at any given time

You can use query filters like:

- Date (current or a specific date, or a date range)
- Combined with source, concept, amount, place, item and/or category

HAL Fintech Chatbot can also answer common financial questions and can have small talks. All interactions are carry out in free-form text conversations with NLP.

Conclusion

Sitting on the foundation of an official Microsoft Bot Framework v4 template named Core Bot (Node.js), I have created an advanced version (Node.js and TypeScript) intended as a quick-start for setting up **Transactional**, **Question and Answer**, and **Conversational** chatbots using core AI capabilities, all in one, while supporting design best practices. This article introduces the template, describes its most important features and discusses why they are there and how you can leverage its benefits. I hope it's helpful and fulfill its goals.

The **CorePlus Bot** template is the result of learning from the scattered set of Microsoft's samples and templates over Github, as well as my own experience and findings writing bots. Actually, the template is the summarized version of a much more complex chatbot I designed and developed. Here are Microsoft's Github links:

- Bot Framework samples repository
- BotFramework-Samples
- Bot Framework Solutions repository
- Web Chat hosted samples

How should a chatbot be designed? How many things to keep in mind? What are the best practices? As an emerging technology, the wide commercial use of chatbots and conversational interfaces is still in its infancy, although they have been around here for years. People involved in the field, like product owners, project managers, UX designers, developers, copywriters, Computer Science and Artificial Intelligence researchers, are still learning and establishing the rules that should drive the design and development of this exciting field, while sharing their findings. Here is a selection of some articles that might help you think and understand the collective knowledge of the community.

- Chatbot Do's and Don'ts – These Are the Best and Worst Chatbot Practices
- The code of ethics for AI and chatbots that every brand should follow
- The User Experience of Chatbots
- Why chatbots fail
- Chatbots were the next big thing: what happened?

The template is ready to use, although it's an unfinished work. The development of the underlying framework is in progress and so should also be **CorePlus**. It should be adapted to the changes and improvements of the MBFv4 over the time. It can and should also be improved with the contributions of the community. Your opinions, suggestions and contributions are more than welcome. So, where do we go from here? *All we need to do is make sure we **keep talking**.*

History

- 20th May, 2019: Version 1.0 - Initial article submitted
- 27th November, 2019: Version 2.0 - TypeScript version added
- 10th December, 2019: Version 2.1 - Video section (PoC) added

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

Rubén Hinojosa Chapel

Software Developer (Senior)
Spain

Microsoft Development Consultant. Mobile Software Engineer with Apple, Google and Microsoft technologies (Xamarin, C#). Chatbots Software Engineer with Microsoft Bot Framework (Node.js, Azure). Collaborates with companies from different sectors in the development or maintenance of their computer systems, helping them in identifying their needs and proposing technical solutions that provide value and solve their problems.

Comments and Discussions

6 messages have been posted for this article Visit **https://www.codeproject.com/Articles/4254785/CorePlus-A-Microsoft-Bot-Framework-v4-Template** to post and view comments on this article, or click **here** to get a print view with messages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Article Copyright 2019 by Rubén Hinojosa Chapel
Everything else Copyright © CodeProject, 1999-2020

Web01 2.8.200113.1