Cortana

Skills:

* To access Cortana’s skills, go to [Cortana Skills Page](https://www.microsoft.com/en-us/cortana/skills/all);
* **Personalize UX:**
  + You can personalize UX using user insights, if the user grants certain permissions;
* **Add intelligence to your skill:**
  + You can add intelligence to Cortana using [Microsoft Cognitive Services](https://docs.microsoft.com/en-us/azure/bot-service/bot-service-concept-intelligence?view=azure-bot-service-3.0&viewFallbackFrom=azure-bot-service-4.0);
* **Natural language understanding:**
  + You can use regEX but you should use the Language Understanding Intelligent Services (LUIS);
  + LUIS uses Node.js or .NET stack;

Cortana Skills Design

Principle of Cortana skills design:

* **Cases to consider when building skills:**
  + **When using voice and more natural to interact;**
    - When user has a disability;
    - When user is on device that is difficult to type on;
    - When user needs to share content or set a reminder;
  + **When the user’s hands are busy:**
    - When the user is cooking;
  + **When using voice is a more efficient experience:**
    - When using voice reduces number of steps required to do something;
    - Example: “Play the latest House of Cards episode”;
  + **When using voice assists with multitasking:**
    - When you want, for example, to control music while reading or replying to messages;
  + **When user is driving, walking, or otherwise distracted:**
    - Easier to speak when navigating and safer, for example;
* **Skill design thought process:**
  + Will it solve the user’s problem with the minimum number of steps?
  + Will it solve the user’s problem better, faster, or more easily than alternatives?
  + Is it intuitive? Will users naturally know what to say when using it?
  + Does it avoid complexity?
  + Can it use default values when the user is not specific?
* **Design for the right scenario:**
  + **Define the key scenarios you want your skill to target:**
    - What are the high value scenarios?
    - Do these scenarios have clearly defined goals?
    - Are the scenarios relatively simple?
  + **Of these, which work well with voice?**
    - Which scenarios don’t rely on visual elements?
    - Which scenarios are relatively quick to get through, with minimal steps?
* **Design the conversation, think about:**
  + Essential questions that the skills must ask;
  + Questions the user will likely ask;
  + Skill’s reply to the user’s question;
  + Information required; Which’s optional;
  + Anything Cortana can infer or remember from prior interaction in this session to use as defaults;
  + Which answers should you confirm before taking any action; Kinds of confirmation you should use;
  + What actions might the conversation trigger on the backend?
  + Will the skill need a directed dialogue;
  + Handle requests for help; For errors;
  + How will the skill react if it reaches a dead end;
  + What environments could the skill be used in;
* **Identify the intents, entities, and utterances:**
  + **Utterances:**
    - Input from the user that your app needs to interpret;
  + **Intents:**
    - Actions that the user wants to perform;
    - **Categories:**
      * Full intent;
      * Partial intent;
      * No intent;
  + **Entities:**
    - Data required to perform action;
* **Identity:**
  + **Full intent:**
    - When the user fully expresses what they want in a single utterance;
    - Should respond directly and either propose further interaction, or end the conversation;
    - **Example:**
      * User: Hey Cortana, ask Mileage Wizard if I have enough miles to travel.

Mileage Wizard: You currently have 30,000 miles. This is enough to travel domestically but not internationally. Would you like to purchase additional miles, or book a domestic trip?

* + - Need to develop as many variations of the intent as possible;
  + **Partial Intent:**
    - When a user partially expresses what they want, but the utterance is missing a required entity;
    - Should detect the missing element and automatically provide a prompt that ask for the missing entity;
    - **Example:**
      * User: Hey Cortana, ask Mileage Wizard if I have miles.

Mileage Wizard: Do you mean miles you can use to travel?

* + **No intent:**
    - When a user gives minimum info, not enough to engage in a conversation;
    - Your skill should consider the first-time user and help them get started;
    - Should present a list of three options to choose from
    - **Example:**
      * User: Hey Cortana, ask Mileage Wizard.

Mileage Wizard: Do you want available miles, used miles or discounts?

* **Ask the user questions:**
  + **Be conversational:**
    - Interact like a person would;
    - Sounding natural > grammatical accuracy;
    - Use the implied first-person tense where possible and natural;
    - Use contractions. Write for ear, not for eye;
  + **Use variations:**
    - Vary your responses to make your app sound natural;
  + **Use phrases like “Ok” and “All right” in response with restraint:**
    - They can repetitive if used too often and without variation;
    - Give acknowledgment in TTS only;
* **Interactions:**
  + Efficient;
  + Relevant;
  + Clear;
  + Trustworthy;
* **Present options:**
  + You should make sure the user understands what you are asking, and that a response is expected;
  + For example, follow a list of options with a question;
  + Keep questions simple and concise;
* **Ask questions with directed or open prompts:**
  + **Directed prompt:**
    - Lists specific choices for the user;
    - Minimize confusion;
  + **Open prompt:**
    - Lets user decide their own choice;
    - Good when answer is obvious, or user is familiarized with the subject;
  + **Good uses of directed prompts are:**
    - A lot of users use the skill, or they use it infrequently;
    - There are never more than three options;
    - **Form:**
      * Use “Please select X, Y or Z” over “Would you like X, Y or Z”;
* **Confirm the user’s answer:**
  + Think about where in the conversation flow the user needs confirmation;
  + You can use a variety of techniques for confirmation and correction;
  + You need to strike a balance with the skill and protection from lost data or wasted time;
  + **Confirmation strategies:**
    - Explicit confirmation;
    - Implicit confirmation;
  + **Explicit confirmation:**
    - Slows conversation because of the extra prompt;
    - Used for situations with high misunderstanding cost (Ex: banking);
  + **Implicit confirmation:**
    - Combines the confirmation with your next question;
    - Faster;
    - The grammar for implicit interaction is subtly different that the explicit;
    - The grammar for the implicit one combines acceptance or denial of the previous prompt with supplying information for the next prompt;
* **Other design considerations:** 
  + **Present help:**
    - Design should include welcome prompts;
    - It should include response to “Help” prompts, by showing your skill’s capabilities and options for that area;
  + **Show progress:**
    - Keep the user informed while you’re working on their request;
    - If it takes longer than three seconds to process, you should let the user know;
  + **Use default values:**
    - Use default values when user in not specific;
  + **Identify the skill when invoked:**
    - If asked without utterance, identify the skill and display help content or ask what to do with leading questions;
  + **Break lists into manageable pieces:**
    - Limit voice interactions to what is absolutely required;
    - Present only three items at one time maximum;
  + **Understand abbreviations and symbols:**
    - Cortana TSP does this automatically:
* **Design your skill’s visual:**
  + Principal goal is designing voice-oriented skills;
  + For some skills, pure voice functionality isn’t practical;
  + Visuals shouldn’t be required to communicate, they should only support intent;
  + **Things to consider:** 
    - How can the visual elements enhance the voice experience;
    - Don’t overload with visual info;
    - Keep tasks glanceable, users shouldn’t need much visual attention;
  + **Add visual elements to your skill:**
    - **Cortana supports Bot Framework cards, such as:**
      * **Adaptive card:**
        + A used-designed card that contains elements you specify;
        + Supports single or carrousel layouts;
      * **Hero card:**
        + A card with one big image;
        + Supports single or carrousel layouts;
      * **Thumbnail card:**
        + With a single small image;
        + Supports single or carrousel layouts;
      * **Receipt card:**
        + Lets user deliver invoice or receipt;
        + Supports single layout;
      * **Sign-in card:**
        + Lets the skill initiate a sign-in procedure;
        + Supports single layout;
    - **Card design tips:**
      * Limit the card’s title to 84 characters or less;
      * Create brief but meaningful responses;
      * Try to fit your content within the height of Cortana’s canvas;
      * Use cards to provide details;
      * Direct users to another screen only when needed;
      * Tailor the experience based on the device the user is using;
      * Use horizontal lists;

Building hybrid user interfaces:

* **Building hybrid interfaces:**
  + Hybrid means, how to blend a Voice interface with either a Graphical Interface or Textual Interface or both;
  + **Cortana as the Primary experience:**
    - **When to pick a voice experience:** 
      * If it’s more natural to interact with voice;
      * If the user’s hands are busy/ support for multitasking;
      * If it’s more effective and/or effective;
  + **The design process:**
    - **1.** Define the process or problem you want to simplify/optimize;
    - **2.** Design the conversation/ the improvement plan to fit the purpose;
    - **3.** Define the performance metrics;
    - **4.** Supplement the conversation with visuals where/if they make sense;
    - **5.** Perform a Usability study and experiment;
    - **6.** Refine the experience based on user feedback;
    - **Design your conversation, keep it:**
      * Efficient;
      * Relevant;
      * Clear;
      * Trustworthy;
  + **When to use rich cards:**
    - Ideally, Cortana skills should be able to stand alone;
    - The cards should be used to support your skills;
    - Cortana supports dee-linking to apps;
  + **Simple text styles with Markdown:**
    - **For text formatting in a Cortana client, use:**
      * italics (\*italic\* or \_italic\_);
      * bold (\*\*bold\*\*);
      * Markdown links ([link text] (URL));
  + **Being a bot – support short conversations:**
    - Conversations should work the same with text or voice;
    - A bot automating a task should make it easier;
    - Where there’s conversation, make it as simple as possible;
    - Bot should feel natural;
    - Bot should be able to handle interruptions;
  + **Using buttons, choices and links:**
    - On rich cards;
    - Buttons should be used to confirm actions;
    - Options should be spoken;
    - **Radio buttons combined with usual “Ok” and “Cancel” buttons are used when:**
      * There 5 or more options;
      * Need to view additional info;
      * Need to interact with choices;
      * Need to view choices as options;
    - Use buttons to let user initiate an immediate action;
    - Instead of using a button to let the user navigate to another page, use a Hyperlink button;
  + **From bot to assistant – follow up:**
    - Consider saving the state of the conversation to allow the end user to change devices or try something out without blocking the conversation;
* **Dealing with large amounts of data, do this:**
  + Refine the search results;
  + Return the top N results (N <= 3);
  + Paginate results (ex: Carousel layouts);
  + Delegate viewing of results elsewhere (ex: ask if user want results emailed);

Invocation name guidelines:

* **Invocation name guidelines:**
  + Avoid the term “bot” in the invocation name;
  + Avoid trademarked or copyrighted names;
  + Must not be longer than three words;
  + Avoid duplicate names;
  + Avoid the use of common proper nouns;
* **Invocation name recommendations:**
  + Specify clear and easily recognized invocation name;
  + Avoid words that are homophones;
  + Use words with multiple syllables or/and multiple words;
  + Avoid compound words unless required for branding;
  + Avoid hard to pronounce words;
  + Avoid names that combine multiple words into one;
  + Prime speech recognition for your skill;
* **Invocation phrases**
  + Can be many (check documentation);

Skills performance guidelines

* **General performance tips:**
  + Load resources at start up;
  + Avoid large images;
  + Avoid long texts;
  + Avoid long running processes;
* **Azure performance guidelines:** 
  + **Same region deployment:**
    - Deploy ALL your services in the same Azure region;
  + **LUIS deployment:**
    - Create a key in the same Azure region as the bot;
  + Avoid cold starts in Azure:
  + **Disable ARR affinity cache:**
    - Disable app request routing if not using the automatic scaling feature;

Bots:

How bots work:

* When starting a conversation, you see two conversation update activities (one for the user joining and one for the bot);
* Message activity carries info between parties;
* **HTTP Details:**
  + Activities arrive at the bot via HTTP POST requests;
  + Bots respond via 200 HTTP status code;
  + Typically, these requests are nested;
* **Turns:**
  + A turn consists of a user’s incoming activity to the bot and the bot and any activity the bot sends back to the user as an immediate response;
  + The turn *context* object provides info about the activity such as the sender and the receiver and allows the addition of info during the turn across layers of the bot;
  + This object carries inbound activity to all the middleware components and the app logic and provides mechanism where the middleware components and the app logic can send outbound activities;
* **Activity processing stack:**
  + Processing starts with an HTTP POST request;
  + Activity info is carried as a JSON payload arriving as a web server;
  + In C#, will be an ASP.NET project and in JS Node.js, will be Express or Restify;
  + The *adapter* is the core of the SDK runtime;
  + The activity is carried as JSON in the HTTP POST body, which is then deserialized to create the Activity object that is then handed to the adapter with a call to *process activity method*;
  + The adapter on receiving creates a *turn context* and calls middleware;
  + To achieve this, the *turn context* provides *send, update,* and *delete* activity response methods in an asynchronous process;
* **Activity handler:**
  + When a bot receives an activity, it passes it on to *its activity handlers*;
  + Then the *turn handler* calls the individual activity handler for whatever type of activity received;
  + **C#:**
    - When bot receives a message activity, turn handler sees it and sends it on the *OnMessageActivityAsync* activity handler;
    - Bot logic for handling and responding to messages goes in the *OnMessageActivtyAsync* handler*;*
    - Bot logic for handling members being added to the conversation will go in your *OnMembersActivityAsync* handler*;*
    - Logic for the handler must be implemented by overriding the methods in your bot;
    - When overriding, call *await base.OnTurnAsync(turnContext, cancellationToken);* to make *TurnAsync* run before your additional code;
    - That base implementation is responsible for example for the calling the *OnMessageActivityAsync;*
  + **JavaScript:**
  + **Middleware**
    - Linear set of components executed in order, giving each a chance to operate;
    - In the final stage of the middleware pipeline there’s a callback to the *turn handler* on the bot class, with the *process activity method;*
    - The *turn handler* is **OnTurnAsync for C#** and onTurn for JS;
    - It takes a turn context as argument, and then the logic running in the turn handler function processes the activity content and generates one or more activities in response, sending them with the *send activity* function on the turn context;
    - Calling send activity on it causes middleware components to be invoked to on the outbound activities;
    - Middleware components execute before and after the turn handler function;
    - This is nested;
  + **Bot structure:**
    - The VSIX Template generates a ASP.NET Core web app;
    - Has the Program.cs and the Startup.cs files;
    - **appsettings.json:**
      * Specifies the config info for the bot;
      * App ID, password, etc.;
    - **Bot Logic:**
      * **C#:**
      * Defined in *Bots/EchoBot.cs*;
      * *EchoBot* derives from *ActivityHandler,* which derives from the *IBot* interface;
      * *ActivityHandler* defines different activities, such as:
        + *OnTurnAsync:*

When any activity type is received, calls one of the other handlers, based on the type of activity received;

* + - * + *OnMessageActivityAsync:*

When any message activity is received, overrides this to handle a *message* activity;

* + - * + *OnConversationUptadeActivityAsync:*

When any activity conversation update activity is received, on a *conversationUptade* activity, calls a handler if members other than the bot joined or left the conversation;

* + - * + *OnMembersAddedAsync:*

When non-bot members join the conversation, override this to handle members joining a conversation;

* + - * + *OnMembersRemovedAsync:*

When non-bot members leave the conversation, override this to handle members joining a conversation;

* + - * + *OnEventActivityAsync:*

When event activity is received, on an *event* activity, call a handler specific to the event type;

* + - * + *OnTokenResponseAsync:*

When token-response event activity received, override this to handle token response events;

* + - * + *OnEventAsync:*

When non-token-response event activity received, override this to handle other type events;

* + - * + *OnMessageReactionActivityAsync:*

When message reaction activity received, on a *messageactivity* activity, call handler if one or more were added or removed from a message;

* + - * + *OnReactionsAddedAsync:*

When message reactions added to a message, override this to handle reactions to a message;

* + - * + *OnReactionsRemovedAsync:*

When message reactions removed from message, override this to handle reactions removed from a message;

* + - * + *OnUnrecongnizedActivityTypeAsync:*

When other activity type is received, override this to handle any activity type otherwise unhandled;

* + - * These handlers have *turnContext* that provides info about incoming activity from HTTP request;
      * These methods handle error checking and then call each of the handlers depending on activity type;
      * **Example:**
      * *public class MyBot : ActivityHandler*

*{*

*protected override async Task OnMessageActivityAsync(ITurnContext<IMessageActivity> turnContext, CancellationToken cancellationToken)*

*{*

*await turnContext.SendActivityAsync(MessageFactory.Text($"Echo: {turnContext.Activity.Text}"), cancellationToken);*

*}*

*protected override async Task OnMembersAddedAsync(IList<ChannelAccount> membersAdded, ITurnContext<IConversationUpdateActivity> turnContext, CancellationToken cancellationToken)*

*{*

*foreach (var member in membersAdded)*

*{*

*await turnContext.SendActivityAsync(MessageFactory.Text($"welcome {member.Name}"), cancellationToken);*

*}*

*}*

*}*

* + - * **Javascript:**
  + **Access the bot from your app:**
    - **Set up services:**
      * The *ConfigureServices* method is in Startup.cs;
      * It loads the connected services, their keys from appsettings.json or Azure key vault, connects state and so on;
      * **Example:**
      * *// This method gets called by the runtime. Use this method to add services to the container.*

*public void ConfigureServices(IServiceCollection services)*

*{*

*services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);*

*// Create the credential provider to be used with the Bot Framework Adapter.*

*services.AddSingleton<ICredentialProvider, ConfigurationCredentialProvider>();*

*// Create the Bot Framework Adapter.*

*services.AddSingleton<IBotFrameworkHttpAdapter, BotFrameworkHttpAdapter>();*

*// Create the bot as a transient. In this case the ASP Controller is expecting an IBot.*

*services.AddTransient<IBot, EchoBot>();*

*}*

* + **Bot controller:**
    - It lets determine the routing of messages and HTTP POST requests;
    - The controller implements *ControllerBase,* holds the adapter that we set in Startup.cs, and passes info to bot when receiving info through HTTP POST;
    - **Ex:**

*// This ASP Controller is created to handle a request. Dependency Injection will provide the Adapter and IBot*

*// implementation at runtime. Multiple different IBot implementations running at different endpoints can be*

*// achieved by specifying a more specific type for the bot constructor argument.*

*[Route("api/messages")]*

*[ApiController]*

*public class BotController : ControllerBase*

*{*

*private readonly IBotFrameworkHttpAdapter Adapter;*

*private readonly IBot Bot;*

*public BotController(IBotFrameworkHttpAdapter adapter, IBot bot)*

*{*

*Adapter = adapter;*

*Bot = bot;*

*}*

*[HttpPost]*

*public async Task PostAsync()*

*{*

*// Delegate the processing of the HTTP POST to the adapter.*

*// The adapter will invoke the bot.*

*await Adapter.ProcessAsync(Request, Response, Bot);*

*}*

*}*

Managing state:

* **Managing state:**
  + A bot is inherently stateless;
  + For some bots, they can either operate without additional info, or the info required is guaranteed to be within the incoming message;
  + For others, state is necessary to have a useful conversation;
  + **Why do I need state?**
    - Allows bots to have more meaningful conversations by having more user data;
    - State allows to keep data longer than the current turn (allows multi-turn conversations);
  + **Storage layer:**
    - **Memory storage:**
      * In-memory storage for testing purposes;
      * For local testing;
      * Cleared each time the bot is started;
    - **Azure Blob Storage:**
      * Connects to AZ Blob Storage object database;
    - **Azure Cosmos DB Storage:**
      * Connects to a Cosmos DB NoSQL Database;
  + **State management:**
    - Automates the reading and writing of your bot’s state to the storage;
    - State is stored as *state properties*, which are key-value pairs that the bot can read and write through the state management object;
    - State properties define how that info is stored;
    - State properties are lumped into scoped **“buckets”, such as**:
      * User state;
      * Conversational state;
      * Private conversation state;
    - All these buckets are subclasses of *bot state* class;
    - **Buckets scope:**
      * **User state:**
        + Available in any turn the bot is conversing with that user on that channel, regardless of the conversation;
      * **Conversation state:**
        + Available in any turn in a specific conversation, regardless of user;
      * **Private conversation state:**
        + Scoped to both the specific conversation and to that specific user;
    - When setting the value of your state property, the key is defined for you internally with information contained on the turn context to ensure that each user or conversation gets placed in the correct bucket and property;
    - See documentation to know how to define keys (code);
    - **When to use each type of state:**
      * **Conversation (context of the conservation):**
        + Whether the bot asked a question, and which question that was;
        + What the current topic of conversation is, or what the last one was;
      * **User state (information about the user):**
        + Non-critical user information, such as name and preferences, an alarm setting, or an alert preference;
        + Info about the last conversation;
      * **Private conversation (track both user and conversation specific info):**
        + **Example, if you had a classroom clicker bot:**

Could aggregate and display student responses for a given question;

Could aggregate each student’s performance and privately relay that back to them at the end of the session;

* + - **Connecting to multiple databases:**
      * Create a storage layer to each database;
      * For each layer, create the state management objects you need;
    - **State property accessor:**
      * Used to read/write state properties;
      * Used to provide *get, set* and *delete* methods for accessing your state properties within a turn;
      * These allow the SDK to get state from underlying storage, and update the bot’s state cache;
      * The state cache is a local cache that stores the data object for you, allowing read and write operations without accessing the underlying storage;
      * Once retrieved, the state property can be controlled like a local variable;
      * You can make your state cache persist with *set* method, and even save it to underlying storage if need be;
      * **How the state property accessor methods work:**
        + **Used to interact with state, how to they work?**
        + ***Get:***

Accessor requests property from state cache;

If property in cache, returns it;

If not, get it from the state management object;

* + - * + ***Set:***

Update the state cache with new property value;

* + - * + ***Save changes:***

Check the changes to the property in state cache;

Write that property in cache;

* + - * **Saving state:**
        + Must call the state management object’s *save changes* method, available in the bot class;
        + Saves all properties in the state cache that you have set up to that point for that bucket, but not for any other buckets;

Dialogs library:

* **Dialogs library:**
  + Dialogs are structures that act like functions;
  + The dialog library provides built-in features, such as *prompts* and *waterfalls* dialogs to make your bot’s conversation easier to manage;
* **Dialogs and their pieces:**
  + Library contains different types of dialog;
  + It contains the idea of a *dialog set, dialog context* and *dialog result;*
  + **Dialog set:**
    - Collection of dialogs;
    - Can be of prompts, waterfall dialogs, or component dialogs;
    - When added, get a string ID;
  + **Dialog context:**
    - Information pertaining to dialogs;
    - Used to interact with a dialog set from your bot’s turn handler;
    - Includes the current turn context, parent dialog and dialog state;
    - Has a string ID;
  + **Dialog result:**
    - Can be returned when a dialog ends;
    - Used to save info to some persisted location if wanted;
* **Dialog state:**
  + Bot holds a dialog set collection as a member variable, with a handler;
  + Within the bot’s on turn handler, it initializes the dialog subsystem by calling *create context,* which returns a *dialog context*;
  + It’s necessary to have state, which is accessed with the accessor provided when creating the dialog set;
* **Dialog types:**
  + **Prompts:**
    - Provide a way to ask the user for info and evaluate their response;
    - Prompts first ask for input, and then return the valid value or start from the top with a reprompt;
    - There are prompt options given when the prompt is called, where you specify the text to prompt, retry it if it fails and set choices to answer;
    - Can choose to add custom validation for your prompt;
    - When a prompt completes, it returns the resulting value that was asked for;
    - **Prompt types:**
      * **Attachment prompt (asks for):**
        + One or more attachments, such as document or image, returns a collection of attachment objects;
      * **Choice prompt:**
        + A choice from a set of options, returns a *found choice* object;
      * **Confirm prompt:**
        + Confirmation, returns a Boolean value;
      * **Date-time prompt:**
        + A date-time, returns a collection of date-time objects;
      * **Number prompt:**
        + Number, a numeric value;
      * **Text prompt:**
        + General text input, returns a string;
    - **Prompt locale:** 
      * **It’s used to de determine language-specific behavior of the:**
        + Choice;
        + Confirm;
        + Date-time;
        + Number;
  + **Waterfall dialogs:**
    - Used to collect info about the user or guide him through tasks;
    - Each step of the conversation is implemented as an asynchronous function that takes a waterfall step context (*step)* parameter;
    - The bot prompts the user at each step, waits for response, and then passes the result to the next step;
    - The context of the waterfall dialog is stored in its waterfall step context;
    - Can use the waterfall step context object to interact with a dialog set;
    - **Waterfall step context properties:**
      * **Options:**
        + Contains input info for the dialog;
      * **Values:**
        + Contains info to add to context, and is carried forward int subsequent steps;
      * **Result:**
        + Contains the result from the previous step;
  + **Prompt options:**
    - The second parameter of the step context’s prompt method takes a prompt options object:
    - **Properties:**
      * **Prompt:**
        + To ask for user input;
      * **Retry prompt:**
        + To send the user if their first input didn’t validate;
      * **Choices:**
        + A list to choose from, for use with a choice prompt;
      * **Validation:**
        + Additional parameters to use with a custom validation;
      * **Style:**
        + Defines how the choices are presented;
    - Specify the initial prompt to the user, as well as retry prompts;
* **Prompt validation:** 
  + Can validate a prompt before returning it to the next step of the waterfall;
  + Validation object take a prompt validator context and return a Boolean;
  + **Properties:**
    - **Context:**
      * The current turn context for the bot;
    - **Recognized:**
      * A *prompt recognizer result* that contains info about user input;
    - **Options:**
      * *Prompt options* that were provided in the call to start the prompt;
  + **Prompt recognizer properties:**
    - **Succeeded:**
      * Indicates whether the recognizer was able to parse the input;
    - **Value:**
      * The return value from the recognizer;
      * The validation code can modify this value;
* **Component dialog:**
  + Used for creating independent dialogs to handle specific scenarios;
  + Each has its own dialog set;
* **Using dialogs:**
  + Use dialog context dialog to begin, continue, replace or end a dialog;
  + Can cancel all dialogs on its stack;
  + The turn handler directs the dialogs and servs as its fallback;
  + The topmost item in a stack is the active dialog;
  + The dialog context directs all inputs to the active dialog;
  + Can remove active dialog when it ends, or with the replace dialog method;
  + **Create the dialog context:**
    - Call the create context method of your dialog set;
    - It takes the set’s *dialog state* property and creates the dialog context;
    - This context is used to start, continue, control the dialogs in the set;
    - The dialog set requires a *state property accessor*

What I can do to potentiate my project:

* **What I will build:**

**Initial environment setup:**

* 1. Design bot structure and establish objective;
  2. Use Visual Studio Bot Framework to start build;
  3. (Optional) Connect to remote Git repository;
  4. (Optional) Connect to Azure DevOps;
  5. Deploy bot to Azure;
  6. Use Cortana Chanel for speech enabling;
  7. Connect Azure Bot Service to remote Git repository for continuous deployment;
  8. Create a LUIS App;
  9. Create a QnA Maker App;
  10. Create/use a database for saving user data and conversation data;
  11. Create/use a database to create the questions and answers for the bot;
  12. For the databases, can use a SQL Server, or SharePoint lists, or both combined;

**Build the bot connections (some may be optional):**

1. Populate LUIS with the wanted entities and intents (can be done using DB/SharePoint);
2. Populate QnA Maker with the questions and answers (can be done using DB/SharePoint);
3. Connect bot to database to save data from conversation (if needed/allowed);
4. Connect LUIS and QnA Maker to Azure;
5. Connect Bot to LUIS;
6. Connect Bot to QnA Maker;

**Create the bot:**

1. (Optional) Create cards for visual prompts;
2. Use LUIS intent to get the top scored message in the DB to match an answer to the question asked;
3. Use QnA Maker and our DB to get the response that matches with the top intent message from LUIS;
4. (Optional) Save answers to SQL / Azure,
5. (Optional) Export answers to Excel / any spreadsheet/table;
6. Debug and test using Bot Emulator and Azure Web Bot service;
7. Fully publish bot to Azure and Cortana;