Chatbot design supported by specialized platforms and frameworks

Arnaud Weyts

Aknowledgements

I would like to thank my 2 academic supervisors, Kristien and Davy for their guidance, feedback and help during both the writing of this bachelor thesis and the connected Blended Mobility project.

Contents

Glossary

CosmosDB One of Microsoft's database services, integrated into the Azure platform. 23

ECMAScript scripting-language specification standardized by Ecma International. 14

ES5 ECMAScript 5, the 5th edition of the language specification. 14

SDK Software Development Kit, set of software tools. 14

Table Storage One of Microsoft's database services, integrated into the Azure platform, NoSQL based. 23

Webhook A webhook is an HTTP callback, a POST request that calls for an event. 25

Research question

The aim of this bachelor thesis is to perform research into chatbots and investigate how they are built. What actually goes into building a chatbot that feels intuitive but is also functional?

1.1 Approach

This thesis will start with some use cases for chatbots, to find out when it would be a good idea to build a chatbot as a solution for a concrete problem.

Already existing chatbots will be presented and compared to paint a picture of what professional chatbots currently look like. Some general research into chatbot best practices will also be conducted based on real life cases

Finally, two popular platforms to create a fully-fledged chatbot will be compared by trying to build a food ordering chatbot.

1.2 Metrics

The metrics for the comparison will be based on a number of different factors. Their general business model will be researched, followed by their approach to building a chatbot. Finally their learning curve and quality of documentation will also be compared.

General chatbot design

2.1 Choosing a chatbot solution

2.1.1 Introduction

Why choose a chatbot as a solution to a problem? To find the answer to that question there should be a concrete description of the problem and research should be conducted into how that problem is currently handled.

2.1.2 Use case 1: Food delivery

Food delivery chatbots are some of the most common chatbots out there. That's because a basic chatbot for ordering food is easy to make and maintain. Most of the time they don't require complex language recognition and they will go through the same steps every time someone wants to order something.

A great example of this is Domino's[1]. They own one of the most popular facebook messenger chatbots even though the bot itself is really simple.

Let's take a look at the perspective of a new imaginary pizza place in town. This pizza place has a set menu with set formulas. They want to innovate and make it possible to receive and process online orders but don't want to lose the familiarity of their brand.

This is a perfect case for a chatbot. If they already have a facebook page, they can easily integrate a chatbot into it and promote it to their current customers. All they need on top of that is an admin panel for the company to maintain their menu and process orders.

Building a chatbot this way also opens up several possibilities for expansion in the future. They can easily start tracking customer's habits and improve their suggestions for specific customers.

However there are some downsides to this solution as well. Creating a chatbot from scratch for this purpose would cost a lot of money. That's why there are already solutions like Chatobook [2] popping up. This is an all-in-one solution that provides a restaurant with a messenger bot and an admin panel to manage promotion, reservations and the menu. You can also find templates for these kinds of bots that integrate with Google Sheets/Excel. [3]

The rise of general delivery services like Ubereats [4] and Deliveroo [5] also complicates this matter. If those companies manage to integrate full ordering services into their platforms using chatbots -which they will most likely succeed in at one point- that would simplify everything. But that also comes with a cost. The imaginary pizza restaurant would lose its unique identity as the chatbot's identity would be replaced with Uber's or Deliveroo's identity.

2.1.3 Use case 2: Banking industry

Next case relates to the banking industry. Lately there has been an influx of integrating AI-powered chatbots into this sector. This is because people like to engage with their bank and get answers that feel human, instantly. It establishes a kind of trust. This type of communication is also very attractive to the banks themselves, because it allows them to quickly provide answers to repetitive questions.

There are lots of specific chatbot banking use cases out there. One of them is basic banking services. Checking account balance, transferring funds, analyzing expenditures. Chatbots can also hugely improve a bank's customer support, which should be available 24/7. They can help out people instantly, without long wait times which a lot of customer support services are currently struggling with. Further more, there's also the opportunity for intranet-based chatbots [6]. These chatbots communicate with the employees to improve their productivity and give them access to the right information in mere seconds. As seen in figure 2.1.5 customer service within a company is one of the business functions that can benefit the most from a chatbot.

Another use case is providing customer-focused marketing. Using a chatbot, it's possible to promote personalized banking services as if the customer is talking to a real life employee.

Overall, integrating a chatbot into a banking environment has a lot of benefits. That's why more and more banks are starting to invest in them.

2.1.4 Use case 3: E-commerce

Lastly there's the quickly growing industry of E-commerce. This ties into both of the use cases above as the main focus is customer interaction and engagement, support and personalized marketing.

Online web-stores are one of the fastest growing industries, they are popping up everywhere. This is because more and more people are starting to buy online. It's convenient, fast and reliable (in most cases). However there is one key benefit missing compared to in-store shopping: interaction.

Here's where the shopping assistant chatbot comes in. It can do everything a normal store employee can and even more. Like instantly retrieving your past orders to help you out quicker and more efficiently. It has access to every single product the web-shop offers and can instantly recommend one or more products that fit your needs. It can seamlessly provide post-purchase customer support. All of this whilst not losing the brand identity of the web-store itself.

As seen in the first figure 2.1.5, E-commerce is projected to be the industry that will benefit the most from chatbots. Anything involving the sale of goods or services online can be considered E-commerce. If a business corresponds to that description, experimenting with a chatbot could be interesting.

2.1.5 Conclusion

Chatbots are an obvious answer to lots of companies that are willing to innovate and are ready to experiment. It allows them to establish their brand in a familiar and human way, whilst also increasing their in-company productivity and overall efficiency.

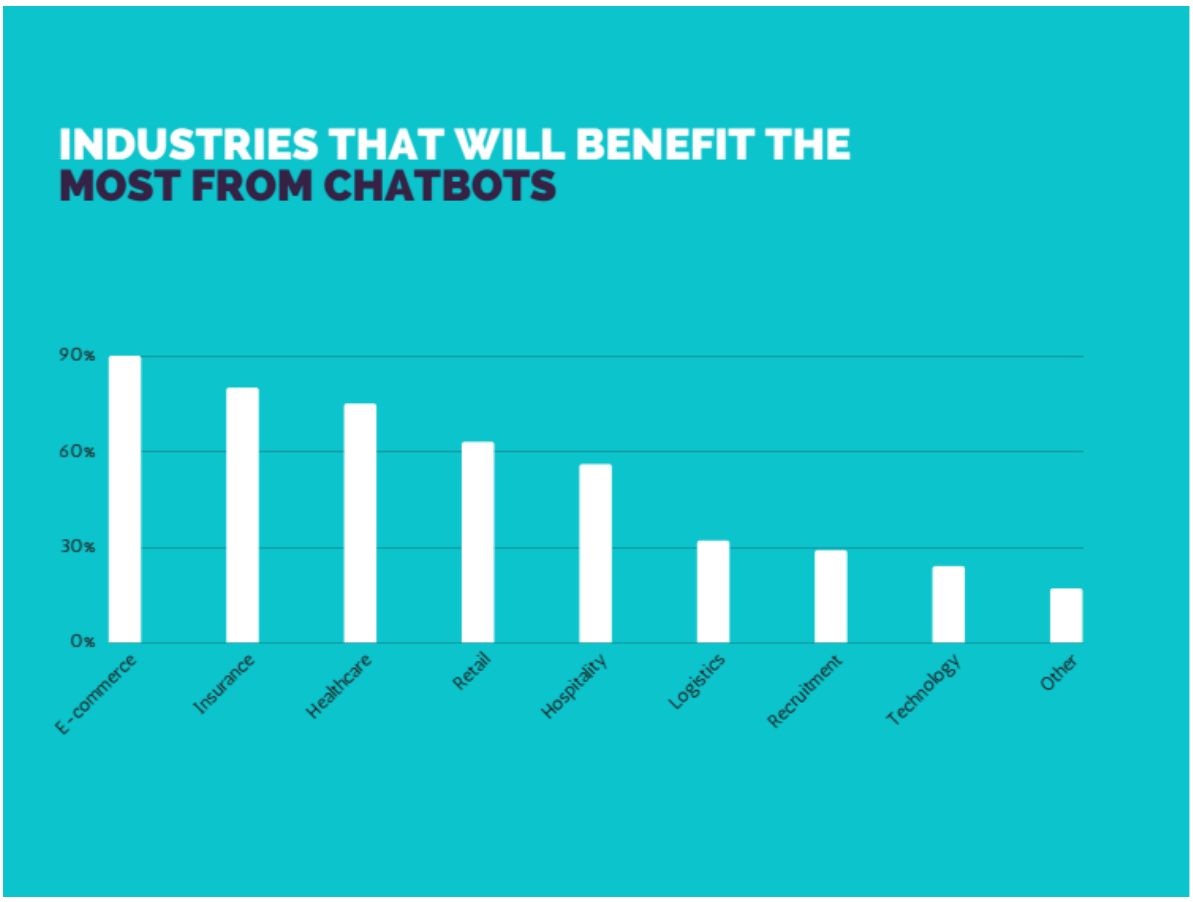

Figure 2.1: Industries projected to benefit from chatbots [7]

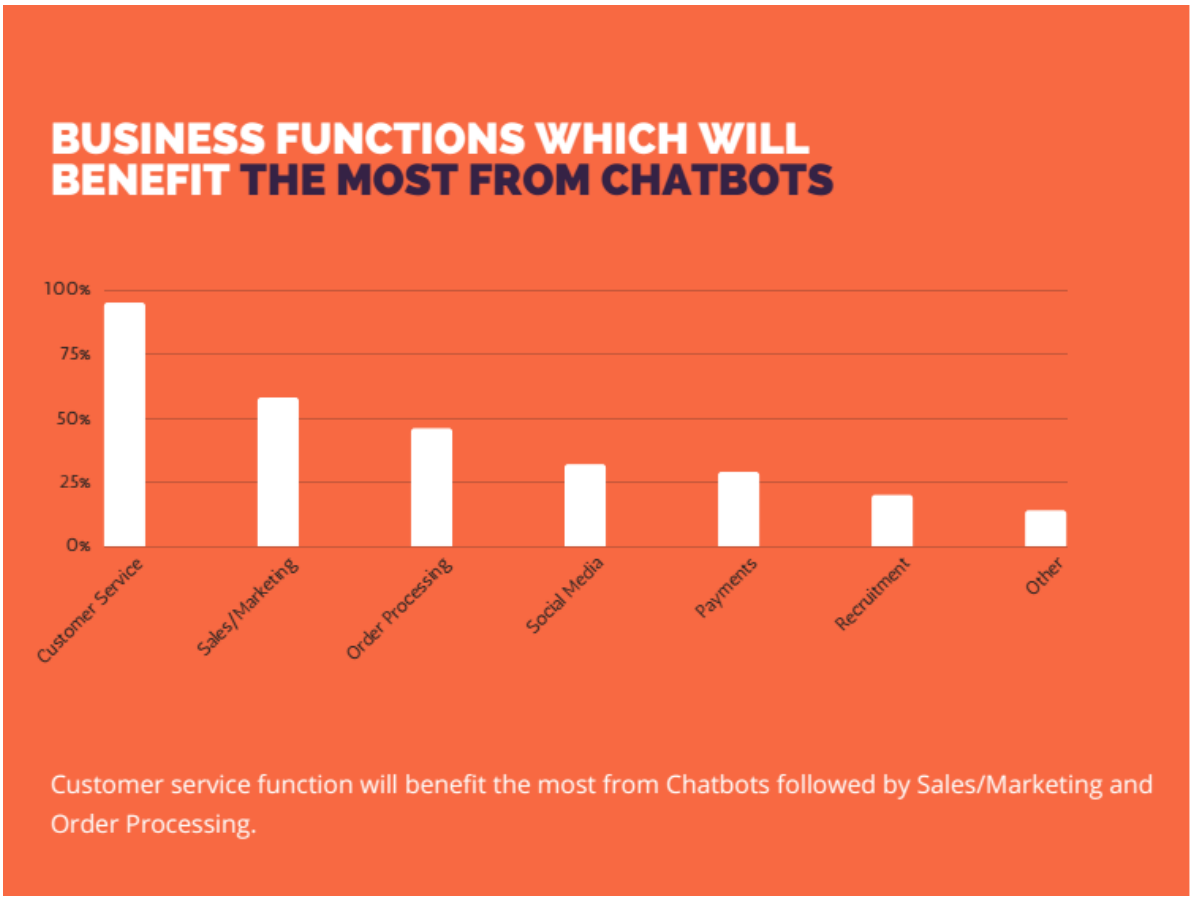

Figure 2.2: Business functions inside a company projected to benefit from chatbots [7]

2.2 Comparison of existing chatbots

Today there are plenty of well-implemented solutions already. There's food-ordering services like Domino's that offer an ordering service. But also stores, ranging from specific ones (H&M) to big, fully fledged web-stores like Bol.com, a Dutch-Belgian web-store.

2.2.1 Dom, Domino's ordering assistant bot

This is a facebook messenger bot for ordering food from Domino's, a fast food chain. Something that can immediately be noticed is the use of tappable quick reply answers. The bot starts with some simple questions in order to know the user's intent and asks some information about the user, such as the address and what store he or she wants to order from.

Once this has been established, the user can choose what he wants to order and checkout.

In conclusion, the bot isn't very smart. It simply takes orders and passes them through to the store. The user needs to follow the steps clearly and in the right order. It doesn't mimic a real life conversation. Some improvements would be building a profile of the user using past orders and providing recommendations. It could also keep you updated on new items being released. This would make the user feel more at home and establish a sort of familiarity with the bot, like he would with a normal employee.

2.2.2 Bol.com, Customer support bot

This bot's main focus lies in customer service. It replaces the long list of frequently asked questions that most customers really don't want to go through to find their answer. The bot also keeps your past and current orders in mind and integrates them into the conversation.

The bot keeps a very friendly and familiar tone towards the user throughout the entire conversation.

The language recognition seems to be focused on single word recognition to link the question to an answer. It also keeps spelling errors into account.

2.3 Best practices

2.3.1 Creating a personality

One of the most important factors in creating a chatbot is to develop a personality for it. This makes the bot a lot more engaging to the users and makes it harder for the user to get frustrated.

If there is already a company personality in place, the chatbot should get its traits there. If that's not the case, adding a name or a face, which doesn't necessarily have to be human, to it can already be a huge help.

2.3.2 Making the conversation flow

Conversations with a chatbot aren't as straight forward as they seem to be. Users can easily get distracted, overwhelmed or annoyed by a chatbot.

It is clear that a lot of research and testing should be performed into the conversational flow. A bad example of this is not having any wait time between messages. This overwhelms the

user as he doesn't have time to read the previous message. Adding typing bubbles between messages not only gives the user time to read the previous message, but it also humanizes the bot.

2.3.3 Predicting failure and user behavior

Predicting failure and user behavior is hard in any user interface. But there are some common mistakes in chatbots that can be avoided even before testing.

There are several ways to handle unexpected user input. One of them is revisiting the previous state of the conversation, another one is restarting the conversation, or asking the user politely what he's trying to accomplish.

Conversations that end up in this path should also be logged for evaluation afterwards to avoid future dead ends.

When using buttons in a bot, for example to visit a website, or start the conversation, they should be functional at all times.

Facebook's messenger platform supports 2 main buttons in their platform. General buttons, and quick replies. Regular buttons will always be clickable multiple times. Quick replies are bound to the message they are meant to follow up on, and disappear as soon as they are clicked.

Of course this best practice goes hand in hand with actual user testing and applies to every user interface, not just to chatbots.

Case study overview

To compare some existing frameworks, a simple chatbot will be designed and created employing the best practices from before. The chatbot will be a food ordering chatbot that offers the user a menu and takes in orders, similar to Domino's bot.

3.1 Conversational state diagram

To shape the bot, a conversational diagram needs to be created first. This diagram explains the main flow of the conversation, with the steps to reach the eventual purpose of the bot. In this case ordering food.

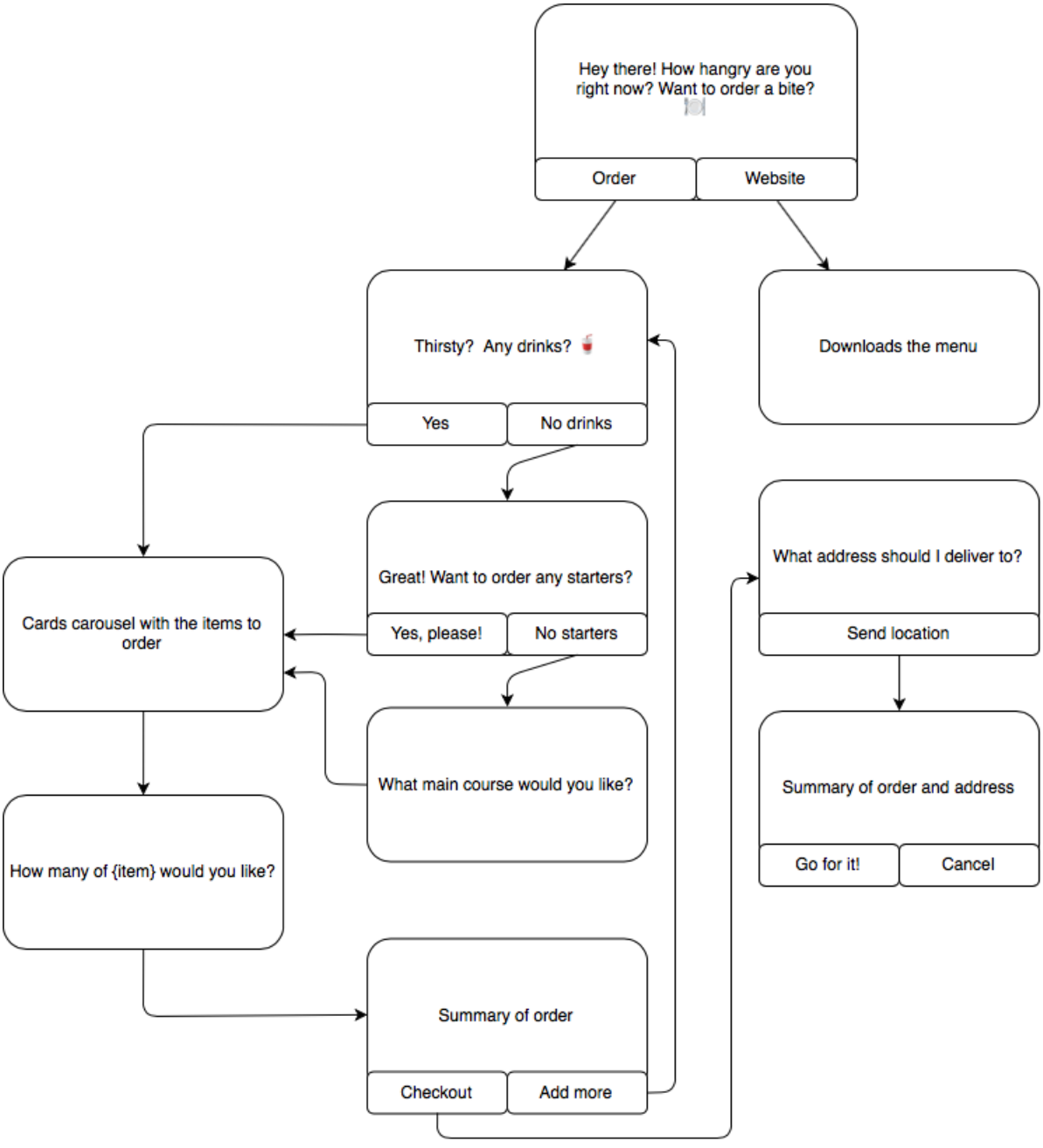

Figure 3.1: Conversational flow diagram the case study bot

Microsoft Bot Framework

The BotBuilder SDK is an open-source solution developed by Microsoft to write a chatbot once and make it possible to connect that same chatbot to multiple channels without rewriting the code. Examples are Skype, Slack, Facebook Messenger, etc.

4.1 Business Model

The Microsoft Bot Framework can be used in several ways. The base platform can be hosted on a server of choice. But to access the extra services, like spell checking, or recognizing user intents using Microsoft's LUIS (Language Understanding), an Azure subscription plan is required.

Azure is Microsoft's Cloud service that can be tailored to the customer's need. The customer can decide what modules or extra features he wants to add and calculate the price.

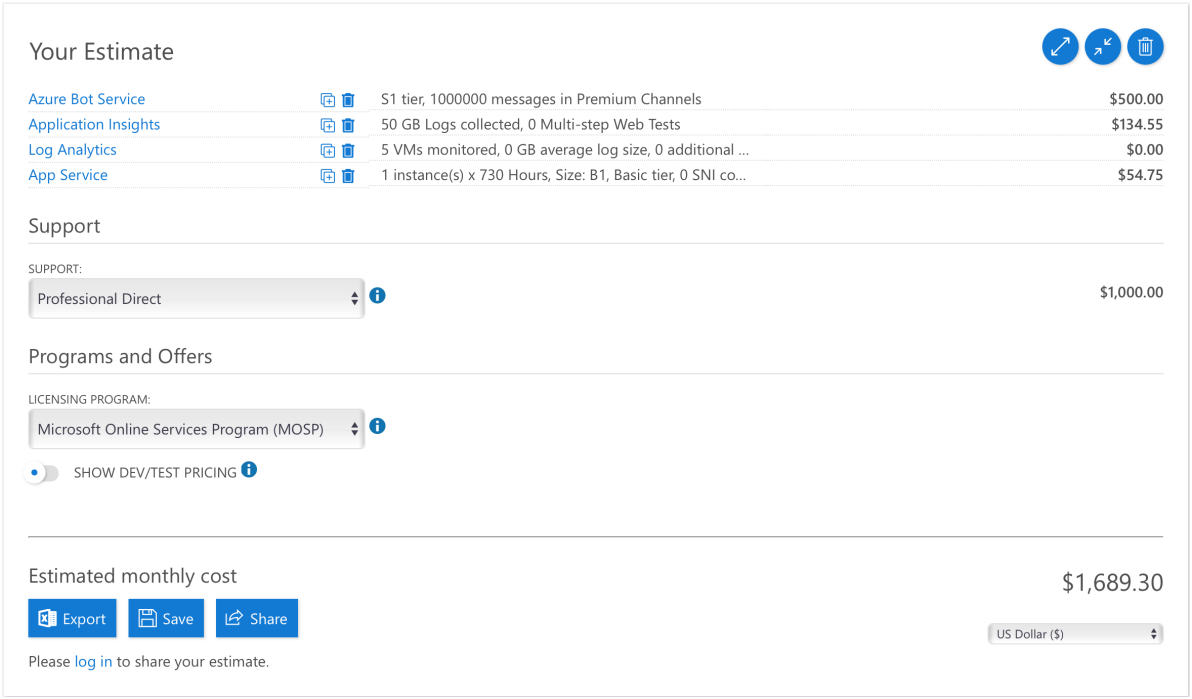

Figure 4.1: Microsoft's Azure Pricing Calculator [8]

4.2 Technical implementation

Microsoft provides the bot framework for 2 languages: Node.JS[9] and C#. If C# is used however, development can only be done inside of a Windows environment, or the limited online code editor.

One solution would be to use Node.JS in a code editor of choice. Preferably Visual Studio Code[10], Microsoft's cross-platform code editor. This way everything can be tailored and configured to the developer's needs.

4.2.1 Developer environment

Setting up a developer environment can be configured from scratch to be as complicated and complex as needed by the company or developer. Node.JS is an open-source, cross-platform JavaScript environment that can be run on a server. And in the end that's what a chatbot is, a server API that responds to requests (messages or events from the user).

It's important to note that JavaScript or also called ECMAScript is a language that is advancing very quickly and is rolling out yearly iterations. Currently the most recent iteration is called ES8. Node.JS is playing catch-up with these iterations and trying to support all of the additions.

The documentation involving the Bot Framework is written in ES5. But it's still possible to write everything in the most recent version of ECMAScript if you compile your code down. This process will turn your unsupported code into code that is supported by Node.JS, depending on the configuration.

Finding documentation on setting up an environment like this is not as easy as one would think. Microsoft does not provide any instructions or boilerplates on how to set up an efficient environment. This gives the developer more freedom but increases the learning curve for a someone who wants to start coding a bot using newer ECMAScript features or the option to hot-reload his code.

Project structure

Structuring the project is straight forward. Just like most production JavaScript projects there is a source folder containing all of the actual source code. Developers can decide to divide the test files into a separate folder but generally it's recommended to keep the test files close to the source file they refer to.

The node_modules folder is a dependency folder, it contains any dependencies the project uses. There are two types of dependencies: First there are developer-dependencies, these are dependencies needed to build the project and participate in development. An example is ESLint[11], a configurable linter to make sure the source code follows specific language rules. There are also regular dependencies, these are external libraries the project might use to do calculations, or wrappers for certain technologies. The Microsoft Bot Framework functions using their botbuilder SDK as a dependency.

Next up are the configuration files, these are the workhorses of the developer's environment. They are used by Webpack[12]. Webpack is a very powerful, well documented build tool and allows for easy customization. It is commonly used for front-end development but supports Node.JS development as well. The development config A starts a server that hot-reloads any changes made to the source files. This way it's easy for the developer to seamlessly check his changes instead of manually recompiling and restarting the server. The production config B

compiles the code down to its most efficient format, which makes it unreadable to a developer, but very efficient for the production server to execute.

Further more there is support for environment variables using the dotenv[13] dependency. Variables can be securely stored in the env file, this file is not versioned to git and will contain any API keys used in the project.

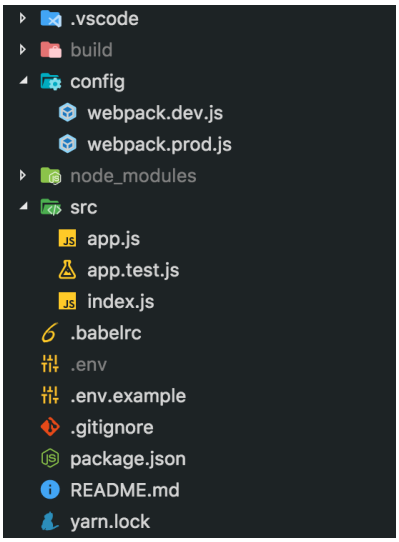

Figure 4.2: Project structure

4.2.2 Testing

The bot can be tested locally or remotely on any platform using Microsoft's own Bot Emulator 4.2.2, which is also open-source. This provides the developer with a live testing interface and detailed information about the bot it's connected to.

It's possible to connect to a locally running bot, or a remotely hosted bot. Speech Recognition is supported for speech enabled bots. Sending system activities is another useful feature allowing the developer to emulate certain user actions like for example a user joining the conversation. Lastly, payment processing is supported to emulate a transaction.

A new iteration of the emulator makes it possible to write transcripts of a conversation in a simple format and load them into the emulator.

4.3 Building a bot

4.3.1 Initialization

Initializing a bot is straightforward, the official documentation recommends restify[15] as the server. During initialization it's possible to define a default message the bot will always use as a reply if no other reply options match.

```
1 import restify from 'restify';
2 import * as builder from 'botbuilder';
3
4 // Setup Restify Server
5 const server = restify.createServer();
6 server.listen(process.env.port || process.env.PORT || 3978, () => {
7   console.log('%s listening to %s', server.name, server.url);
8 });
```

```
9
10 // Create chat connector for communicating with the Bot Framework Service
11 const connector = new builder.ChatConnector({
12     appId: process.env.MicrosoftAppId,
13     appPassword: process.env.MicrosoftAppPassword
14 });
15
16 // Listen for messages from users
17 server.post('/api/messages', connector.listen());
18
19 const bot = new builder.UniversalBot(connector, (session) => {
20   session.send(
21     "Sorry, I didn't get that. Type 'help' if you need assistance or try a
          different sentence.",
22     session.message.text,
23   );
24 }).set('storage', inMemoryStorage);
```

Listing 4.1: Initialization of a chatbot

To make the bot start the conversation certain events can be used. In this case a 'conversationUpdate' event will be used to trigger the root dialog.

```
1 bot.on('conversationUpdate', (message) => {
2   if (message.membersAdded) {
3     message.membersAdded.forEach((identity) => {
4       if (identity.id === message.address.bot.id) {
5         bot.beginDialog(message.address, '/welcome');
6       }
7     });
8   }
9 });
```

Listing 4.2: The root event of a bot

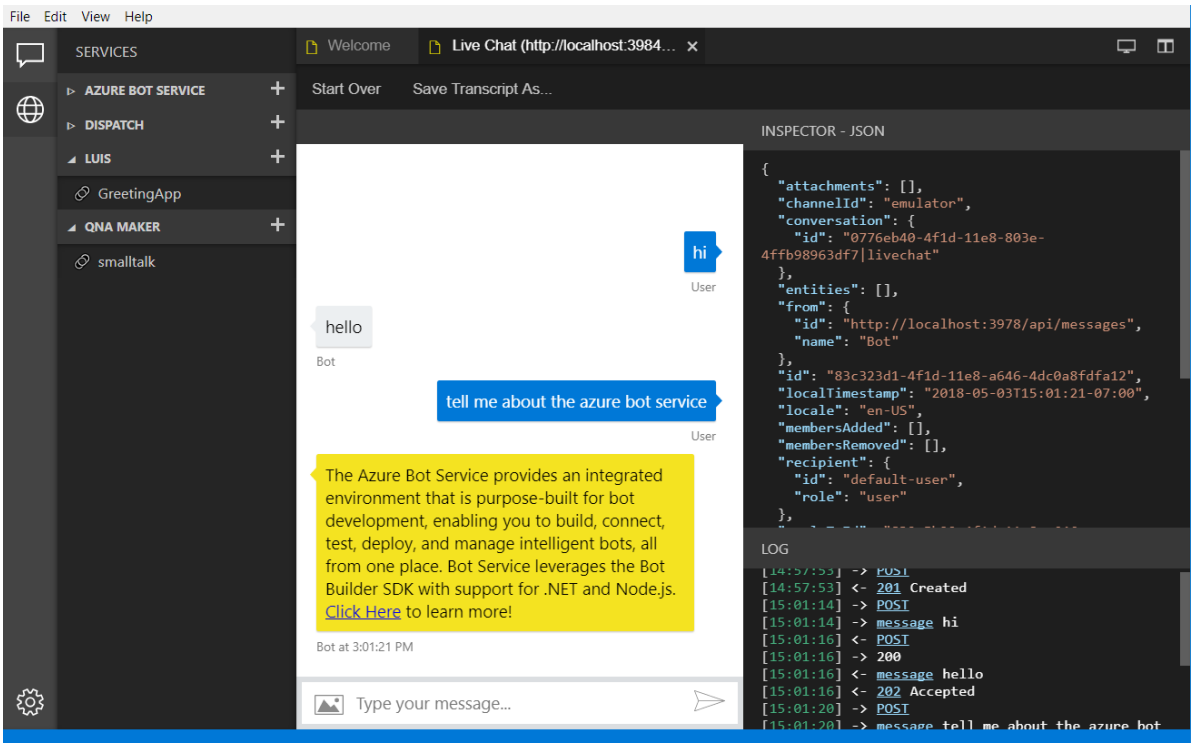

Figure 4.3: Microsoft's Bot Emulator [14]

4.3.2 Event types

- message
- conversationUpdate
- contactRelationUpdate
- typing
- ping
- deleteUserData
- endOfConversation
- event
- invoke
- messageReaction

During a regular conversation there are several different events that will be triggered. A message event simply represents any communication between bot and user. The conversationUpdate event is triggered when any members are added/removed from the conversation, including the bot. Or when conversation metadata has changed. The contactRelationUpdate event indicates that the bot was added to or removed from a user's contact list. These are the most common events, the other events are pretty self-explanatory.

Most of these events can be manually triggered using the bot emulator or defined in code, enabling unit testing.

```
1  const event = {
2    address: { bot: { id: '0' }, user: { id: 0 } },
3    agent: 'botbuilder',
4    source: 'facebook',
5    sourceEvent: '',
6    type: 'conversationUpdate',
7    membersAdded: [{ id: '0', isGroup: false, name: 'test' }],
8    user: { id: '0', isGroup: false, name: 'test' },
9  };
10
11 connector.processEvent(event);
```

Listing 4.3: Sending a mock event to the bot

4.3.3 Dialogs

One of the key concepts in the Bot Builder SKD are dialogs. Dialogs help the developer manage the conversational logic and is a fundamental part of developing a chatbot in the bot framework.

A dialog can be compared to a function. It can perform a specific task and be called at any point in time. Dialogs can also contain other dialogs to advance and branch off a conversation.

4.3.4 Waterfall

A waterfall is a type of dialog that guides the user through a set of steps or tasks. After each step, the bot will wait for the user to respond and pass the reply to the next step. A great example of this is ordering food. First the user will be asked for drinks, followed by the

amount, again followed by the question for what starter the user wants, and so on. There is a clear structure the bot goes through, hence the name waterfall.

A waterfall is defined by creating an array of consecutive functions or waterfall steps inside of a dialog.

Prompts

To help the bot wait for a reply Microsoft provides developers with so called 'Prompts'. These already contain some extra functionality to validate the user's reply. A prompt should be used whenever the bot expects a reply from the user.

Types

- text
- confirm
- number
- time
- choice
- attachment

When used inside of a waterfall, the reply of the user will be passed to the next step using the 'results' object. Another argument called 'next' can be used as a method to skip to the next step of the waterfall.

```
1  bot.dialog('orderButtonClick', [
2    (session) => {
3      builder.Prompts.choice(session, 'Thirsty? Want to order any drinks?', 'Yes
           |No drinks', {
4        listStyle: builder.ListStyle.button,
5      });
6    },
7    (session, results, next) => {
8      if (results.response.entity === 'Yes') {
9        session.beginDialog('orderDrink');
10     } else {
11       next();
12     }
13   },
14 ]);
```

Listing 4.4: 2-step waterfall using a prompt

4.3.5 User actions

There are multiple ways to handle user input other than the usage of prompts. One way of doing so is by using actions. Actions are bound to specific dialogs.

Triggeraction

The most common action is a 'triggerAction'. This can be connected to a dialog to invoke it whenever the user inputs a matched term. Whenever this is triggered, the dialog stack is cleared and the invoked dialog will be the new first dialog on the stack.

This behavior isn't always desired when the conversation needs to be temporarily redirected and resumed later on. That's where the 'onSelectAction' option comes in. This allows the developer to program different behavior to the action.

```
bot.dialog('help', function (session, args, next) {
  //Send a help message
  session.endDialog("Global help menu.");
})
// Once triggered, will start a new dialog as specified by
// the 'onSelectAction' option.
.triggerAction({
  matches: /^help$/i,
  onSelectAction: (session, args, next) => {
      // Add the help dialog to the top of the dialog stack
      // (override the default behavior of replacing the stack)
      session.beginDialog(args.action, args);
  }
});
```

Listing 4.5: triggerAction being bound to a dialog and its behavior overwritten by onSelectAction

BeginDialogAction

A specific dialog can also be attached to a dialog using the 'beginDialogAction' action. This can be useful to send the user to a contextual action. An example of this is when the user asks for specific help in a dialog context. When triggered, it will run the specified dialog and when that dialog finishes, resume at the step of the dialog it was bound to.

This is similar to calling the 'session.beginDialog('dialogid')' method from inside any dialog.

ReloadAction

The reloadAction is somewhat self-explanatory. Binding this to a dialog means it will restart the dialog whenever the action is invoked.

CancelAction

This action cancels the dialog it is bound to when triggered. The parent dialog will also receive an indication the child dialog was canceled.

EndConversationAction

Similar to the cancelAction action, the endConversation action will end the entire conversation when triggered. This clears the entire dialog stack and persisted state data.

CustomAction

Unlike all the other actions, a customAction does not have any default action defined. It's up to the developer to define what it should do. The main benefit of using these is providing quick answers to a user without manipulating the dialog stack at all.

Further more a customAction does not bind to a dialog, instead it binds to the bot itself.

Important to note

Important to note is the fact that some of these actions are quite interruptive to the conversation's flow. To make sure the user really wants to clear the entire dialog stack and start over, a confirmation prompt can also be added to these actions.

4.3.6 Messages

There are many different kinds of messages the bot can send to the user. This ranges from the default text message to a fully constructed card with images or a video. However not all channels support fully constructed cards or other rich attachments.

Channel Inspector

The channel inspector [16] is a tool developed by Microsoft to see what all of the message types look like on different platforms. A Skype card will different to a Messenger card for example. It also provides users with notes and limitations on a specific message type for a platform.

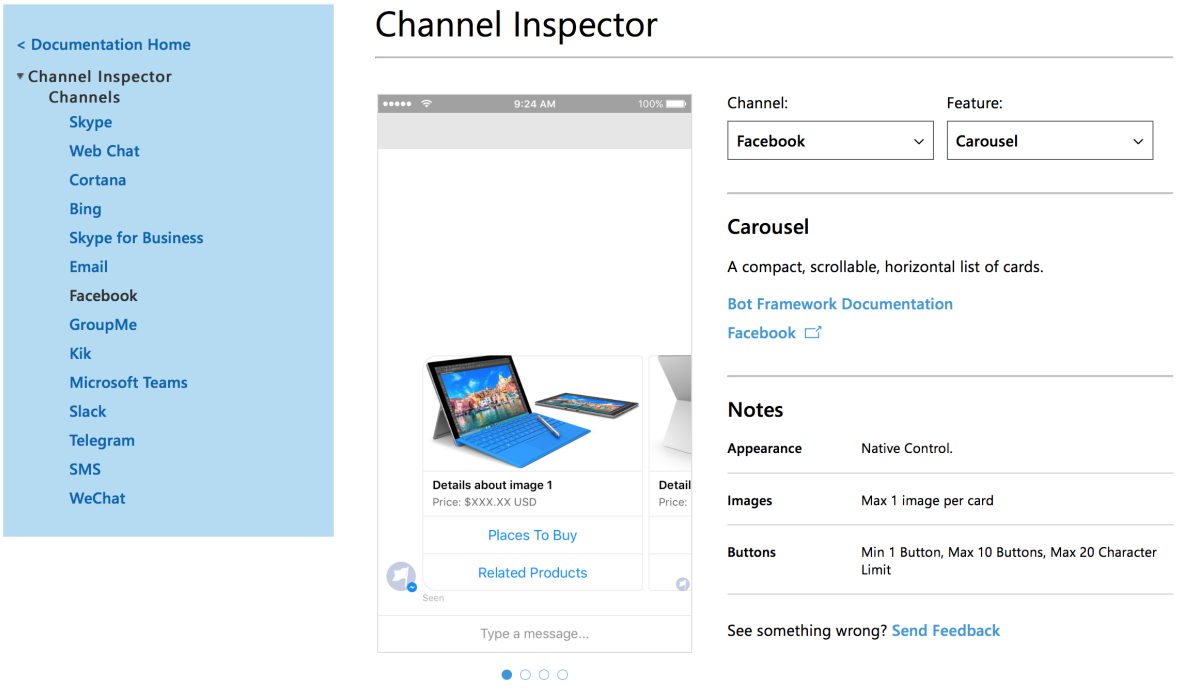

Figure 4.4: Channel Inspector

Default message handler

```
1 session.send("Good morning.");
2
3 const userMessage = session.message.text;
```

Listing 4.6: Sending a simple text message or reading the message from the user is very easy

To construct more complicated messages, a 'Message' object will have to be created. After initialization, different methods can be called on the object to set properties, like attachments, input hints or even the format of the message.

Attachments

There are two main types of attachments that can be added to a message. There are media and files, this involves files like images audio or videos. A second type is a card. As seen in the example of the figure of the Channel Inspector 4.3.6.

```
// Getting an attachment
const msg = session.message;
const attachment = msg.attachments[0];


// Sending a simple attachment
const attachment = {
  contentType: 'image/jpg',
  contentUrl: 'https://78.media.tumblr.com/tumblr_m1wmyyTckD1qejbiro1_500.jpg'
      ,
  name: 'This is definitely not a cat image',
});

session.send({
  text: 'Here is a cat image!',
  attachments,
});
```

Listing 4.7: Receiving an attachment from the user and sending an image

Cards

Many different types of cards are available. Multiple cards can be added to a single message and displayed using a carousel, as seen in listing 4.8.

Hero Card	The most common one, usually contains one big image, text and one or more buttons.
Thumbnail Card	Contains a single thumbnail, text and one or more buttons.
Animation Card	A card for gifs or short videos.
Audio Card	A card for an audio file.
Video Card	A card for a video file.
Receipt Card	A card that mimics a real receipt, allowing to input an item and cost, taxes, total cost, etc.
Signin Card	To request a user to sign in from a 3rd party
Adaptive Card	A fully customizable card that can contain images, text, input fields, buttons, etc.

Table 4.1: The different types of cards

```
const cards = [
  new builder.HeroCard(session)
    .title(Item 1)
    .subtitle($5)
    .buttons([builder.CardAction.imBack(session, 'Item 1', 'Add to order')])
    .images([builder.CardImage.create(session, 'https://item-1.jpg')])),
```

```
  new builder.HeroCard(session)
    .title(Item 2)
    .subtitle($7)
    .buttons([builder.CardAction.imBack(session, 'Item 2', 'Add to order')])
    .images([builder.CardImage.create(session, 'https://item-2.jpg')])),
]

const msg = new builder.Message(session)
  .attachmentLayout(builder.AttachmentLayout.carousel)
  .attachments(cards);
```

Listing 4.8: Example of how to construct some HeroCards and display them in a carousel

The chatbot created for this case study makes use of the HeroCard to represent products. Each product has a title, price, image and button 'Add to order'.

4.3.7 Storage

Managing state data can be approached in several ways. State data could represent a user and its preferences over time, or it could be data that is only temporarily stored during the conversation, like the current order for a user.

Microsoft recommends the usage of the Bot Builder Framework's own in-memory data storage for testing and prototype bots. But for production bots they recommend the developers to implement their own storage solution, or use on of the Azure extensions to store data in Table Storage[17], CosmosDB[18], or SQL.

If one of the Azure integrations is implemented, the methods used to set and persist data are the same as the In-memory data storage.

In-memory data storage, CosmosDB or Table Storage

Important to note is the fact that this storage is cleared every time the bot is restarted, that is why it's only recommended for testing purposes.

```
1 const inMemoryStorage = new builder.MemoryBotStorage();
2
3 const bot = new builder.UniversalBot(connector, [..waterfall steps..]).set('
    storage', inMemoryStorage);
```

Listing 4.9: Example on how to set up the in-memory data storage

As seen in listing 4.9, the storage is set when creating the bot instance. The setup when using one of the Azure extensions is very similar, the developer simply imports a package and sets some API keys before adding the storage to the bot.

Container	Scope	Description
userData	User	Data bound to user, persists across different conversations.
privateConversationData	Conversation	Data bound to user, does not persist across conversations.
conversationData	Conversation	Data bound to a conversation, but shared across all users inside of the conversation.
dialogData	Dialog	Data bound to a single dialog, when the dialog ends the data is cleared.

Table 4.2: Storage containers

As seen in table 4.2 there are several different containers which can be used to store data. The 'userData' container can be used to store user preferences and persist them across conversations. The 'conversationData' and 'privateConversationData' containers could be used to store the user's current order, as that is only relevant to the current conversation. The 'dialogData' container is useful for data relevant to the dialog.

Setting or getting this data is as easy as it gets. Once set up, the containers can be accessed anywhere using the 'session' object as seen in listing 4.10.

```
1 // set data
2 session.userData.preferences = ['spicy'];
```

```
3
4 // get data
5 const preference1 = session.userData.preferences[0];
6
7 // delete data
8 session.userData = {};
```

Listing 4.10: Example on how to use the storage containers

Google Dialogflow

Dialogflow[19] is a platform developed by Google to build natural and rich conversational experiences. Much like the Microsoft's Botframework, Dialogflow aims to make it possible for chatbot creators to create a single chatbot that connects to multiple channels like Skype, Slack and Messenger, etc.

Dialogflow seems to focus on the usage of their GUI and tools to create conversations, but it's also possible to write all of the logic using code.

5.1 Business Model

Google offers 2 main pricing options for their platform. Firstly there is the free edition, it includes free usage with unlimited text query requests. These requests are limited to 3 queries per second. It's also possible to easily integrate voice interaction into Dialogflow, the free edition limits these kinds of interactions to 1000 requests per day with a maximum of 15000 per month. Support is also limited to community and email support. Google themselves recommend this plan for small to medium businesses or developers trying to experiment with Dialogflow. Next to the free edition, there is an enterprise edition as well. This edition has a pay as you go model, which means it's easily scalable as the business will be paying per request. Choosing this option also opens up the possibility to integrate Google Cloud Support and receive specialized instant support.

5.2 Technical implementation

Because Dialogflow is very focused on their GUI, small to medium businesses that don't need extensive logic in their chatbot can make use of the existing tools to create a chatbot. This is recommended to get started and get familiar with the platform. When more complicated logic needs to be implemented like API calls and custom logic that wouldn't be supported by the platform, custom Webhooks can be created. These allow developers to receive a message that was recognized by Dialogflow and implement their custom logic to handle the message.

It's also possible to take full advantage of Dialogflow using their client libraries. These libraries are available for C#, GO, Java, Node.JS, PHP, Python and Ruby. There is some documentation available to get started but diving deeper into the development will take great effort. To take advantage of this functionality, the enterprise edition of Dialogflow needs to be used.

5.2.1 Developer environment

There are several ways to choose a development environment and set up a workflow. Using Google Dialogflow isn't just focused on coding, developers mostly need to work with the user interface provided by google.

To get started, simply making use of the website GUI will suffice. There is also an inline editor (5.2.1) provided to handle requests in code. Another way to handle requests is by enabling a custom webhook to receive information from Dialogflow, like an intent that was recognized, and handle it. This webhook would be written by the developer, following Dialogflow's specification to send back messages. Setting up this webhook is completely up to the developer, it can be done in any server language and does not make use of any library linked to Dialogflow, all the webhook will receive is a simple POST request.

Figure 5.1: Dialogflow's inline editor for handling intents

5.2.2 Testing

Testing the bot is done using Dialogflow's own testing segment on the interface. When making use of a custom webhook testing becomes more complicated as the webhook will have to be deployed every time the developer wants to actually test the replies of the chatbot.

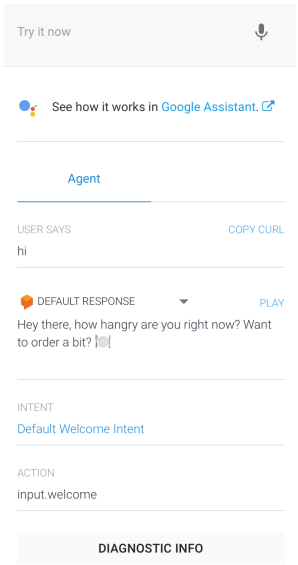

Figure 5.2: Testing a bot using the Dialogflow platform

5.3 Building a bot

5.3.1 Initialization

Because a Dialogflow bot cannot be locally tested using any emulator, initialization of the bot happens on the platform itself. The platform uses a different term for chatbot called 'agent'. The agent contains intents, entities and responses towards the user. To begin, a new agent should be created.

This can be done from scratch, or by using one of the prebuilt agents (figure 5.3.1) provided by the Dialogflow team as a base. There is a number of prebuilt agents available ranging from simple use cases like setting an alarm to more complicated ones like banking operations or online shopping.

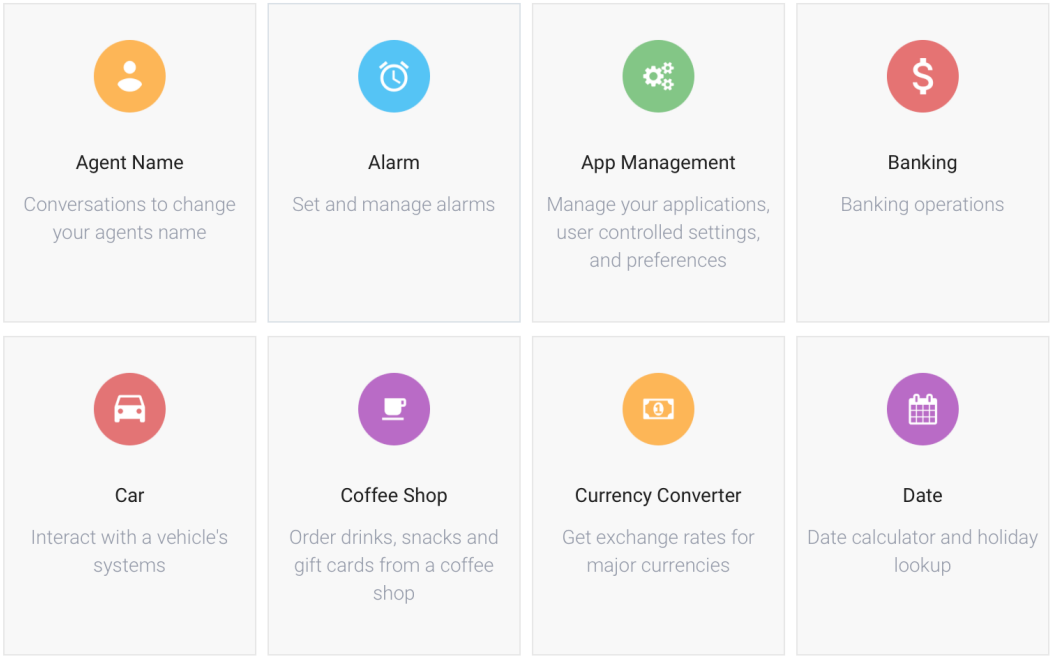

Figure 5.3: Some of the prebuilt agents made available by the Dialogflow team

Agents are fully customizable and easy to import but the prebuilt agents do not contain any responses. There is also a small talk agent that can be added on top of a main agent to push specific branding and provide users with a more human experience.

5.3.2 Intents

After initialization, intents should be defined. An intent links what the user says to a specific action the bot should take. It represents what a user means and how the agent should react.

Next to a regular intent, a follow-up intent can also be added. These intents follow up on existing intents and are used to for example go through the process of ordering food.

Fallback intents can be defined as well, these will be triggered if none of the user's input matched any of the intents. Typically these are used to define responses like 'I didn't get that'. It's where the agent ends up when it doesn't know what to do.

Another useful feature is the ability to add weight to different intents using something called 'Intent priority'. This comes in handy when multiple intents were matched but some should

be more important than the others.

An intent contains 4 main segments: training phrases, action, response and contexts.

Training phrases

A training phrase is an example phrase the user says. Inside of an intent, multiple training phrases can be defined. Dialogflow will recognize a user's response and use these training phrases to link it to a certain intent.

Two types of training phrases can be defined, examples or templates. Examples are the best way of constructing a training phrase. They are written in natural language and certain words are annotated for extraction. Annotation means linking a word to an entity, more on entities later on. Dialogflow can automatically annotate words in the training phrase and recognize certain values like for example a time definition. A template training phrase contains direct references to an entity instead of an annotation.

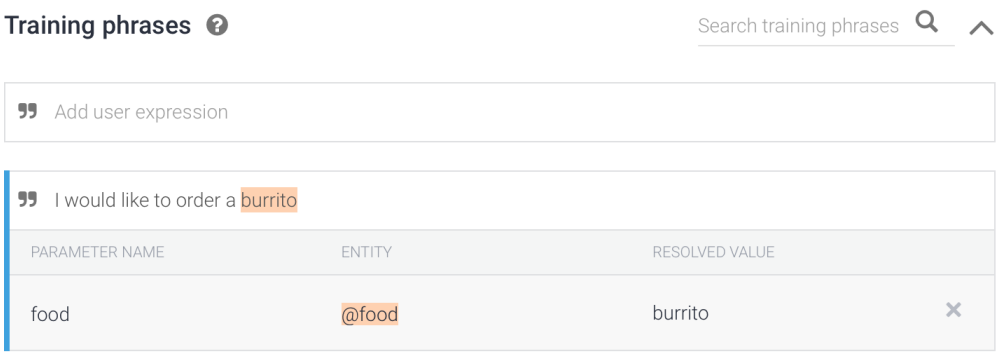

Figure 5.4: An example training phrase with an annotated parameter

Action and parameters

This section contains a single action and its parameters. An action is a simple trigger word for the agent to handle the intent using its recognized parameters. The parameter table is connected to the training phrases. If some words were automatically annotated and recognized as a parameter from inside the training phrases section, they will have been added to the table. There can be cases where parameters would have to be added manually, which can be done using this section.

In this section, it's also possible to set certain parameters as required. This means that they should always get recognized inside of the sentence. When a parameter is missing, prompts can be called to ask the user for that specific information. An example is when a user asks to order a hamburger but doesn't specify a date. If the date parameter is set as required with a defined prompt (figure 5.3.2), the agent will ask the user when he wants his hamburger to be delivered.

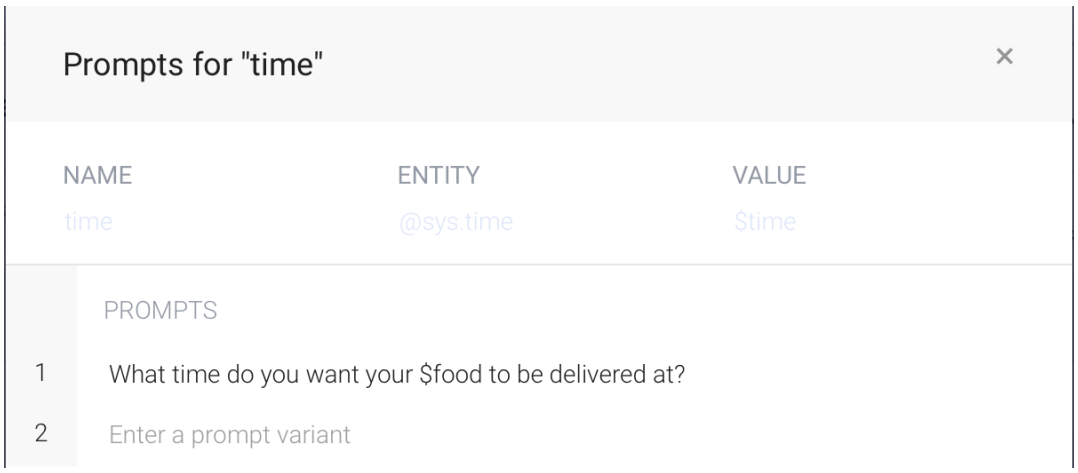

Figure 5.5: Setting custom prompts to ask for a missing parameter

Responses

This is the section where the bot responses will be defined. Multiple responses can be defined for an intent to allow for variation. Different response types can be added for specific platforms.

Contexts

Contexts are parameters that can be passed out of the intent. It's a way for the agent to remember the response of a certain intent and use that response again in a different intent. An example is when a user asks to turn on a specific light somewhere, and in his next message simply sends 'Turn it off'. Without context there is no way to determine what the user wants to turn off, but using context the agent can remember that the user was talking about that specific light. These contexts have a default lifespan that can be changed manually as well.

5.3.3 Entities

Entities are a second important part of Dialogflow's toolset. They are used to distinguish important values from a user's input. An example of an entity can be a date, an object or a location.

There are 3 types of entities in the Dialogflow platform: system, developer and user. These 3 types can each also be divided into a mapping, enum or composite.

System entities

System entities are prebuilt entities provided by the Dialogflow platform. They are general entities that can be used everywhere. Examples are the '@sys.date' entity, the '@sys.color' entity or the '@sys.unit-currency' entity. The date entity is an example of a mapping. They match different values like 'The first of January' or '01-01-2018' and they map to a specific date format. The color entity on the other hand is an example of an enum. The color gets matched but it doesn't link to a reference value and just returns the color as is. Lastly the unit-currency entity is an example of a composite. It's an entity containing another entity. For example 50 dollars will return an object consisting of 2 values linked to an attribute: the amount will be 50 and the currency will be dollar.

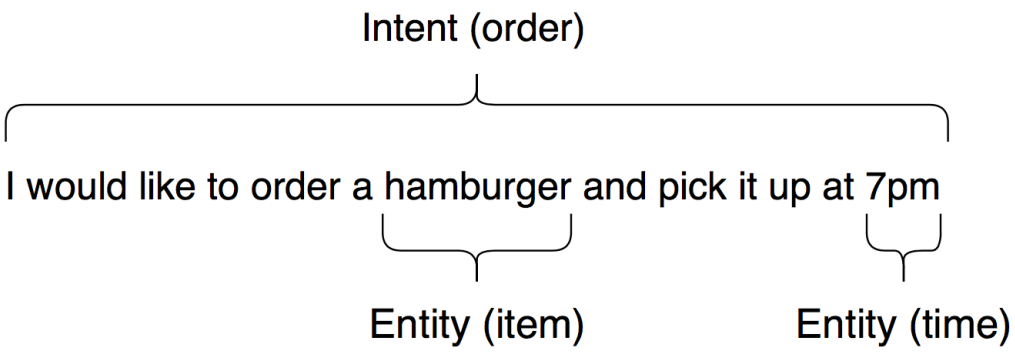

Figure 5.6: Representation of an intent and entity

Developer entities

It's possible to define custom entities as well. These should be used when the system entities don't cover all of the agents needs.

User entities

User entities are entities that are defined on a session level. They entities that can be specifically defined to a user. The use case for this is out of the scope of this thesis and documentation about it is scarce.

Using entities

Defining entities or using system entities is pretty straight forward (5.3.3). In the case of a food ordering chatbot you could have a custom entity called 'food-type'. This kind of entity could be an enum. A food type for the food ordering chatbot would be 'starter'.

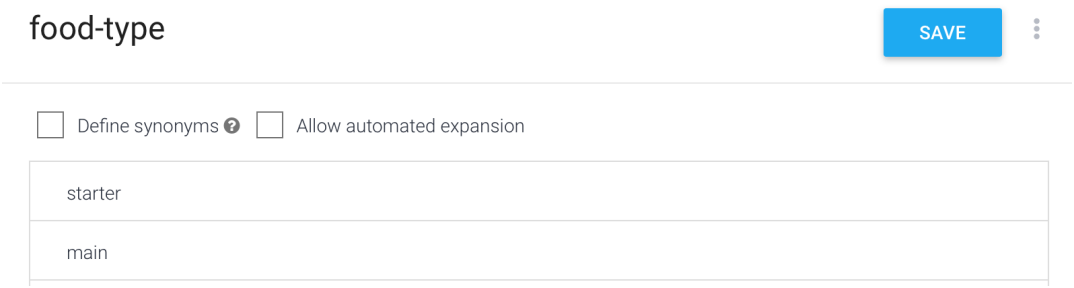

Figure 5.7: Creating an enum entity

This entity can then be used straight away in any intent by inputting training phrases. The entity will be recognized automatically and added to the parameter table as seen in figure 5.3.3.

5.3.4 Events

Triggering intents using events is supported as well. These events can be custom events defined by the developer, or default events defined by platforms, as seen in figure 5.3.4. An example

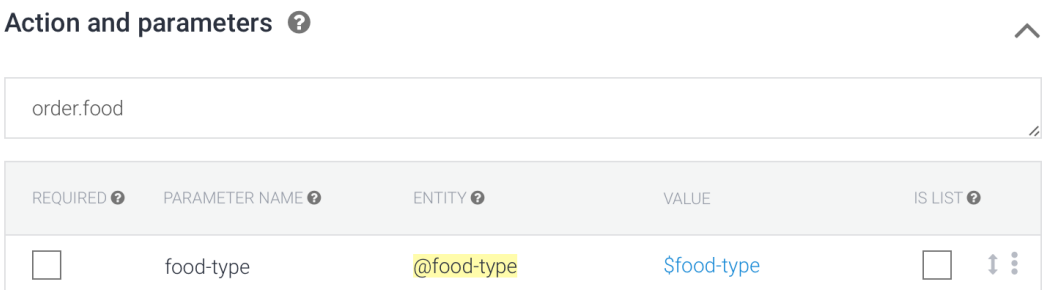

Figure 5.8: The parameter table of an intent

is the 'WELCOME' event, which will be triggered when a user starts the conversation.

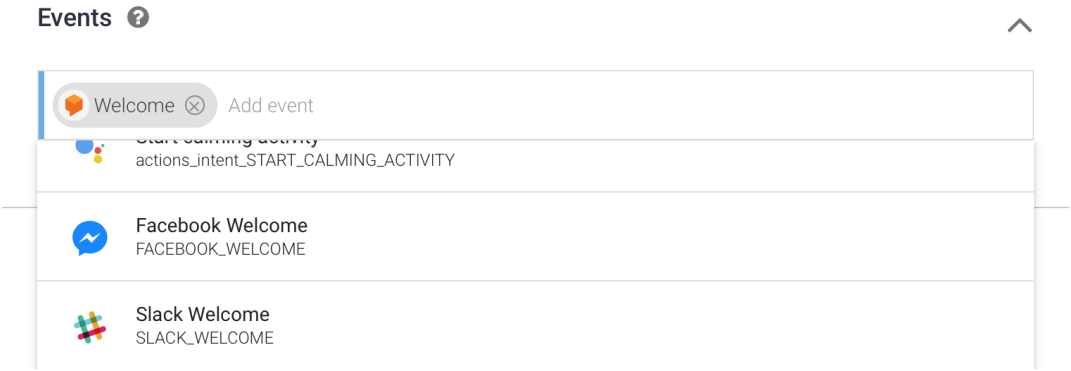

Figure 5.9: Selecting one of the default events to trigger an intent

The only way to call a custom event is by using the webhook functionality. Inside the matched webhook for an intent, a response needs to be sent to the Dialogflow platform.

```
{
  "followupEvent": {
      "name": "myCustomEvent",
      "data": {
        "ExampleParameter":"Test"
      }
  }
}
```

Listing 5.1: The response that needs to be sent to the Dialogflow API to trigger a custom event

5.3.5 Fulfillment

Fulfillment is the final section to construct a full conversational flow. In this section the developer can define a fully custom webhook URL that will receive POST requests from Dialogflow following their API specification. Inside the webhook the developers can handle all of their custom needs which aren't supported by the platform and send a request back to the platform to send a message.

Another option is using the inline editor (5.2.1) to set up some basic custom requests without having to deploy a full webhook outside of the platform.

Conclusion

List of Figures

References

[1] (2018). Domino's, [Online]. Available: https://www.facebook.com/Dominos/ (visited on 2018-04-27).

[2] (2017), [Online]. Available: https://chatobook.com/ (visited on 2018-03-20).

[3] (2018), [Online]. Available: https://botmakers.net/chatbot-templates/pizza-delivery-chatbot-for-facebook-messenger (visited on 2018-03-20).

[4] (2018), [Online]. Available: https://www.ubereats.com (visited on 2018-03-20).

[5] (2018), [Online]. Available: https://deliveroo.co.uk (visited on 2018-03-20).

[6] (2018), [Online]. Available: https://acuvate.com/solutions/meshbot/ (visited on 2018-03-26).

[7] (2017), [Online]. Available: https://reviewkiss.com/2017-year-chatbot-booming/ (visited on 2018-03-26).

[9] (2018). Node.js, [Online]. Available: https://nodejs.org/en/ (visited on 2018-04-27).

[10] (2018). Visual studio code, [Online]. Available: https://code.visualstudio.com (visited on 2018-04-27).

[11] (2018). Eslint, [Online]. Available: https://eslint.org (visited on 2018-04-27).

[12] (2018). Webpack, [Online]. Available: https://webpack.js.org (visited on 2018-04-27).

[13] (2018). Dotenv, [Online]. Available: https://github.com/motdotla/dotenv (visited on 2018-04-27).

[14] (2018). Microsoft bot emulator, [Online]. Available: https://docs.microsoft.com/en-us/azure/bot-service/bot-service-debug-emulator (visited on 2018-03-28).

[15] (2018). Restify, [Online]. Available: http://restify.com (visited on 2018-04-27).

[16] (2018). Channel inspector, [Online]. Available: https://docs.botframework.com/en-us/channel-inspector/channels/Skype/ (visited on 2018-04-28).

[17] (2018). Table storage, [Online]. Available: https://azure.microsoft.com/en-us/services/storage/tables/ (visited on 2018-04-28).

[18] (2018). Cosmosdb, [Online]. Available: https://azure.microsoft.com/en-us/try/cosmosdb/ (visited on 2018-04-28).

[19] (2018). Dialogflow, [Online]. Available: https://dialogflow.com (visited on 2018-04-30).

[30] (2018), [Online]. Available: https://www.just-eat.com (visited on 2018-03-20).

[33] (2018). Webhook explanation, [Online]. Available: https://webhooks.pbworks.com/w/page/13385124/FrontPage (visited on 2018-04-30).

Bibliography

[8] (2018). Azure pricing, [Online]. Available: https://azure.microsoft.com/en-us/pricing/#explore-cost (visited on 2018-03-28).

[20] J. Verplanken. (2018). Learning ux design? build a chatbot., [Online]. Available: https://www.invisionapp.com/blog/learn-ux-design-build-chatbot/ (visited on 2018-03-13).

[21] F. Teixeira. (2017). Why chatbots fail, [Online]. Available: https://chatbot.fail (visited on 2018-03-13).

[22] N.N. (2018). Chatbots & conversational ui, [Online]. Available: https://uxdesign.cc/chatbots-conversational-ui/home (visited on 2018-03-13).

[23] W. Fanguy. (2017). What every designer can learn from alexa, [Online]. Available: https://www.invisionapp.com/blog/every-designer-learn-alexa/ (visited on 2018-03-13).

[24] J. Toscano. (2016). The ultimate guide to chatbots, [Online]. Available: https://www.invisionapp.com/blog/guide-to-chatbots/ (visited on 2018-03-13).

[25] I. Jindal. (2017). Building chatbots: Why message length matters, [Online]. Available: https://www.invisionapp.com/blog/chatbots-message-length/ (visited on 2018-03-13).

[26] D. B. Sera Koo. (2016). Chatbots: Your ultimate prototyping tool, [Online]. Available: https://www.invisionapp.com/blog/chatbots-prototyping-tool/ (visited on 2018-03-13).

[27] D. Wright. (2018). Us messenger usage for top brands 2018 march, [Online]. Available: https://medium.com/@davidwright_68835/us-messenger-usage-for-top-brands-2018-march-d4f1ef786622 (visited on 2018-03-13).

[28] D. Faggella. (2017). 7 chatbot use cases that actually work, [Online]. Available: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754 (visited on 2018-03-19).

[29] V. Chettupalli. (2018). Important use cases of ai-powered chatbots in the banking industry, [Online]. Available: https://medium.com/swlh/important-use-cases-of-ai-powered-chatbots-in-the-banking-industry-19115411af54 (visited on 2018-03-19).

[31] E. ". Seo. (2017). 11 more best ux practices for building chatbots, [Online]. Available: https://chatbotsmagazine.com/11-more-best-ux-practices-for-building-chatbots-67362d1104d9 (visited on 2018-03-28).

[32] (2017). What are the best practices for chatbot development? [Online]. Available: https://chatbotsmagazine.com/which-are-the-best-practices-for-chatbot-development-91c1b05fdb07 (visited on 2018-03-28).

Appendices

Botframework developer build script

```
const webpack = require('webpack')
const path = require('path')
const nodeExternals = require('webpack-node-externals')
const StartServerPlugin = require('start-server-webpack-plugin')
const Dotenv = require('dotenv-webpack')

module.exports = {
  mode: 'development',
  entry: [
    'webpack/hot/poll?1000',
    './src/index'
  ],
  watch: true,
  target: 'node',
  externals: [nodeExternals({
    whitelist: ['webpack/hot/poll?1000']
  })],
  module: {
    rules: [{
      test: /\textbackslash.js?\$/,
      use: 'babel-loader',
      exclude: /node_modules/
    }]
  },
  plugins: [
    new StartServerPlugin('server.js'),
    new Dotenv(),
    new webpack.NamedModulesPlugin(),
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoEmitOnErrorsPlugin(),
  ],
  output: {
    path: path.resolve('./build'),
    filename: 'server.js'
  }
}
```

Botframework production build script

```
const webpack = require('webpack')
const path = require('path')
const nodeExternals = require('webpack-node-externals')

module.exports = {
  mode: 'production',
  entry: [
    './src/index'
  ],
  target: 'node',
  externals: [nodeExternals()],
  module: {
    rules: [{
      test: /\.js?$/,
      use: 'babel-loader',
      exclude: /node_modules/
    }]
  },
  plugins: [
    new webpack.NamedModulesPlugin(),
    new webpack.NoEmitOnErrorsPlugin(),
  ],
  output: {
    path: path.resolve('./build'),
    filename: 'server.js'
  }
}
```