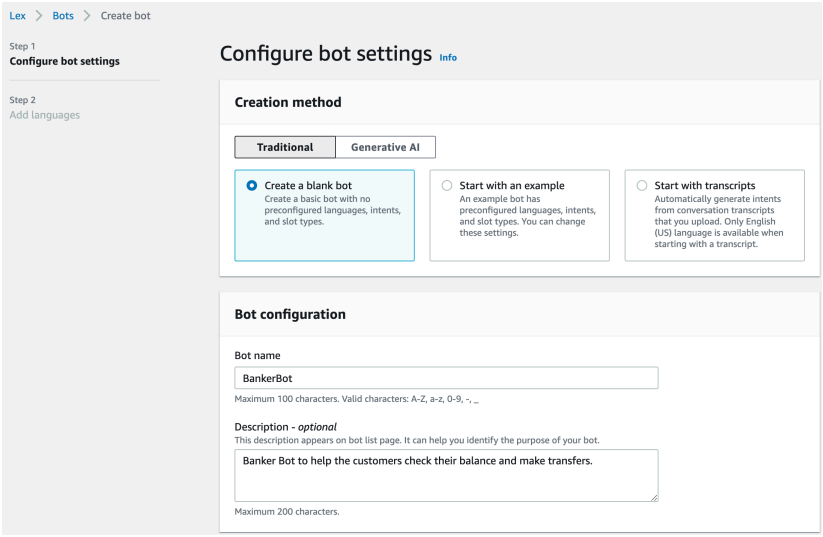
Build a Banking Chatbot with Amazon Lex and Lambda

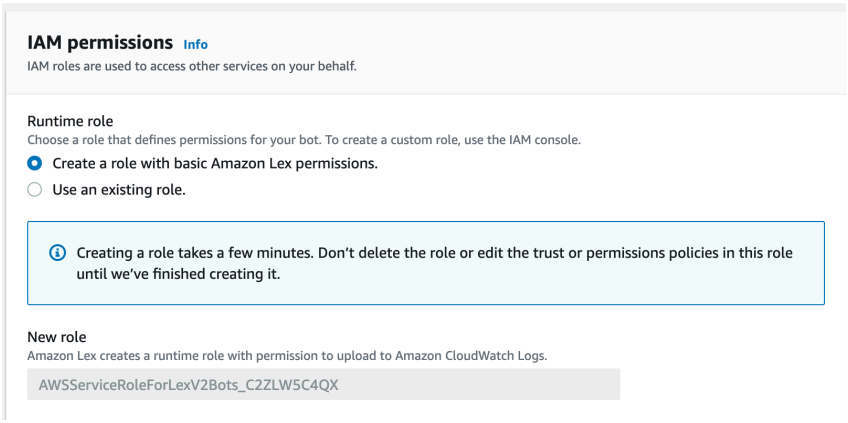
a practical chatbot, BankerBot, that can help bank's customers check their account balance and transfer money between accounts!

Amazon Lex is a service that uses deep learning to enable developers to build conversational interfaces into any application. It provides the same speech recognition, natural language understanding, and text-to-speech capabilities as those used in Amazon Alexa.

Created a Blank Bot in Lex with BankerBot as a name and give description for the bot.

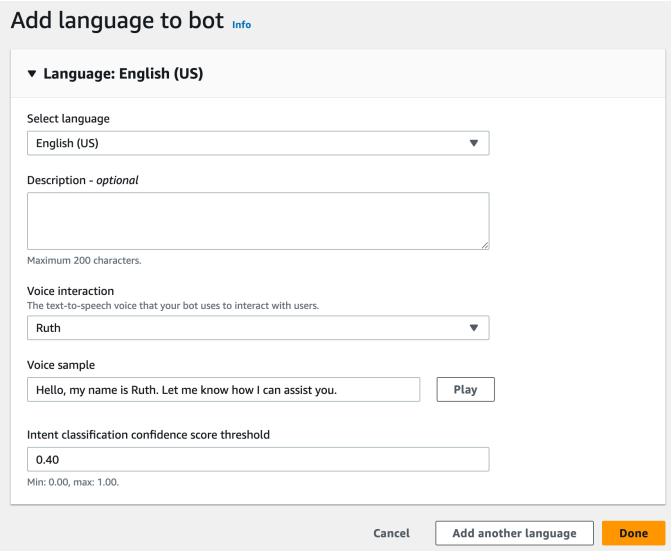


Under IAM permissions, selected the role with basin Amazon Lex permissions. Amazon Lex needs the permission to call other AWS services on our behalf.



- [ ] The Idle Timeout Session – Amazon Lex will only maintain a session for a set length of time. If the user is idle and doesn't add any input for X minutes, their session will end. For now, I've chosen 5 minutes which is default.

Next, given the language and Voice for the bot. For the confidence score keep it as default 0.40

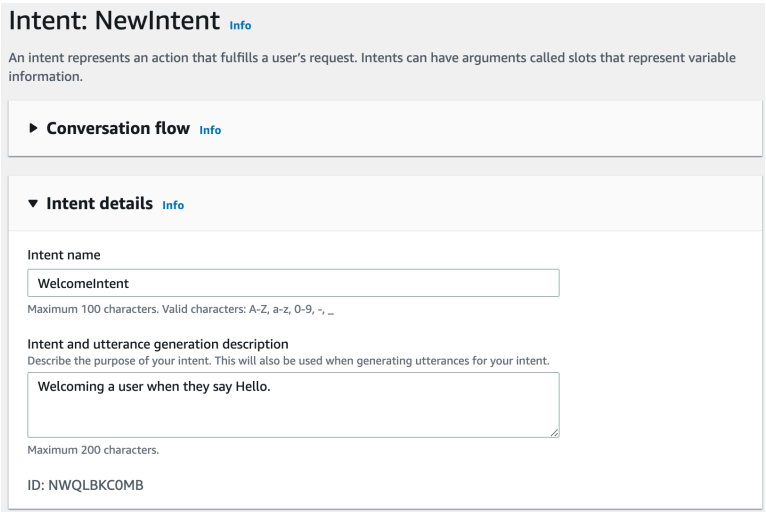


When using Amazon Lex to build a chatbot, this threshold is like a minimum score for the chatbot to confidently understand what the user is trying to say. Setting this to 0.4 means that your chatbot needs to be at least 40% confident that it understands what the user is asking to be able to give a response. So if a user's input is ambiguous and your chatbot's confidence score is below 0.4, it'll throw an error message.

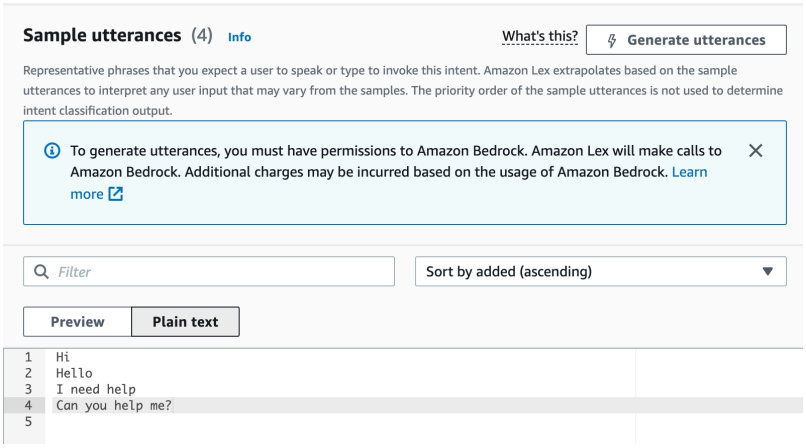
Creating intents

- An intent is what the user is trying to achieve in their conversation with the chatbot. For example, checking a bank account balance; booking a flight; ordering food.
  In Amazon Lex, building the chatbot by defining and categorizing different intents. If we set up different intents, one single chatbot can manage a bunch of requests that are usually related to each other.

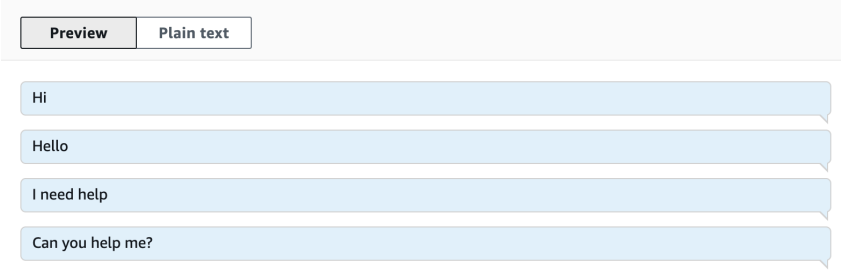
After creating the bot, Intent page opened. Entered the Name and Description.



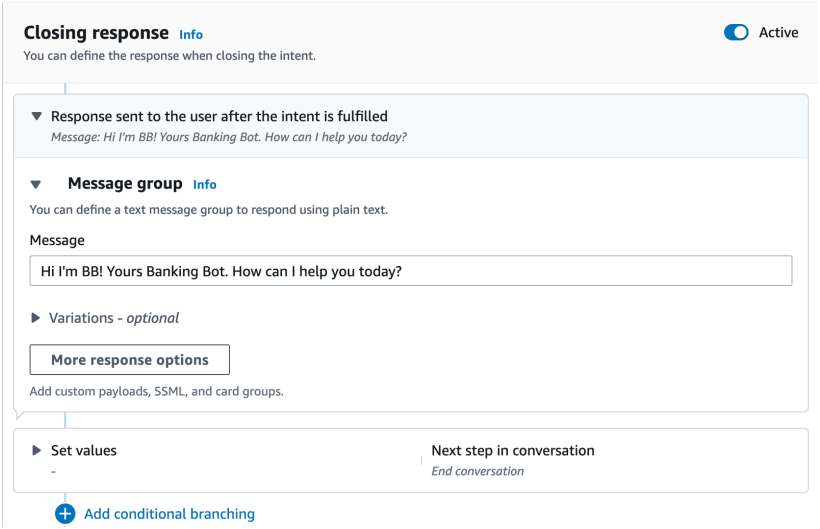
Under Sample Utterances, provided the sample text in Plain text, the people uses while starting the chat.



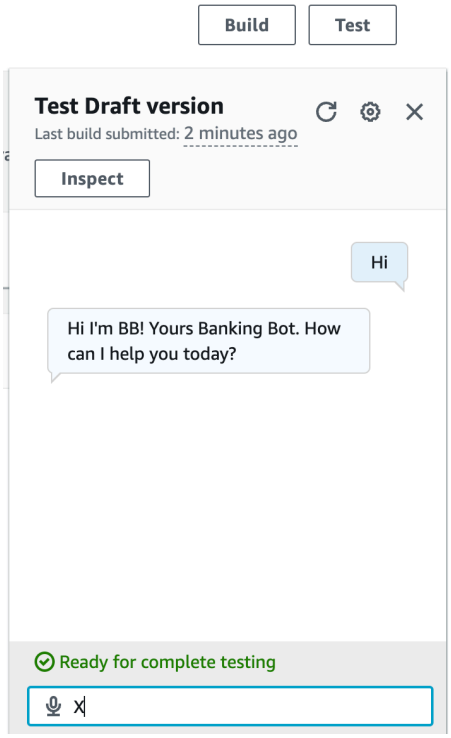
In the preview,



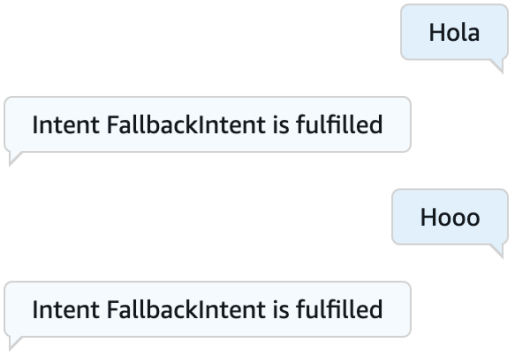
Under Closing Response, Entered the message of what the bot responds to the user.



Now Saved Intent and Built the English(US) and then Tested.



Try other than the mentioned utterances.

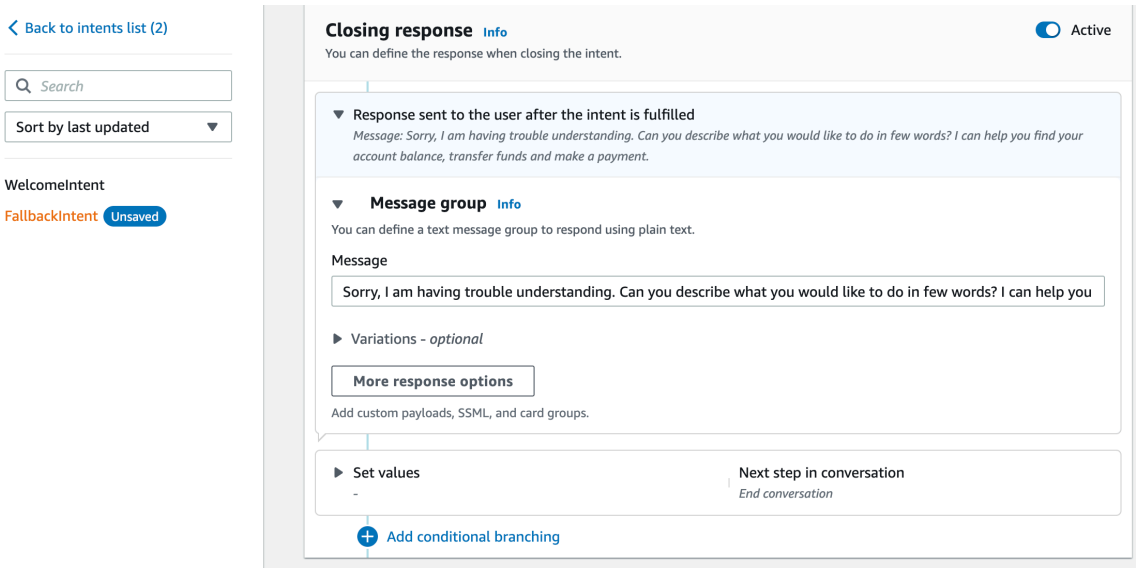


the last two failed, resulting in an **Intent FallbackIntent is fulfilled** response - meaning Amazon Lex doesn't quite recognize the utterance. Instead of text, also tried with voice.

Managing FallbackIntent

If the chatbot has a confidence score **below** 40% for all the intents defined (in our case, it's just the WelcomeIntent for now), the FallbackIntent is triggered. It is a custom error message that chatbot will use to tell the user it doesn't understand their input.

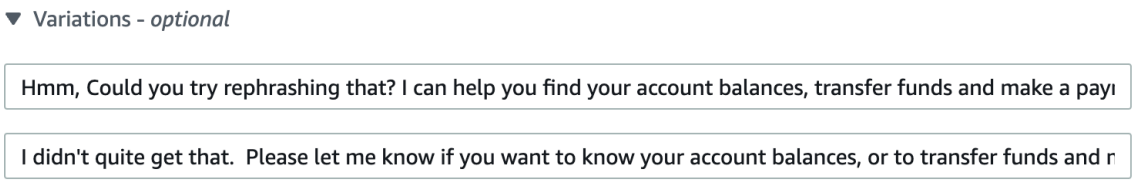
In the FallbackIntent Closing response changed the message **Intent FallbackIntent is fulfilled** to **Sorry, I am having trouble understanding. Can you describe what you would like to do in few words? I can help you find your account balance, transfer funds and make a payment.**



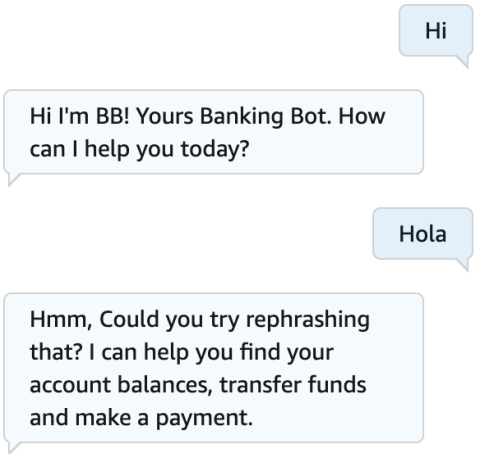
Under Variations - optional : Provided few Variations.

→ Hmm, Could you try rephrashing that? I can help you find your account balances, transfer funds and make a payment.

→ I didn't quite get that. Please let me know if you want to know your account balances, or to transfer funds and make a payment.



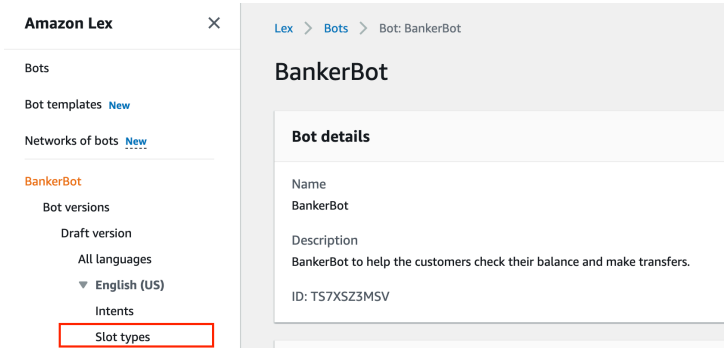
Saved the intent and Tested it.



Building Chatbot with Custom Slots

After successful creation of BankerBot, now it's time for letting the customers know their account balance with their date of birth as verification.

Slot Types



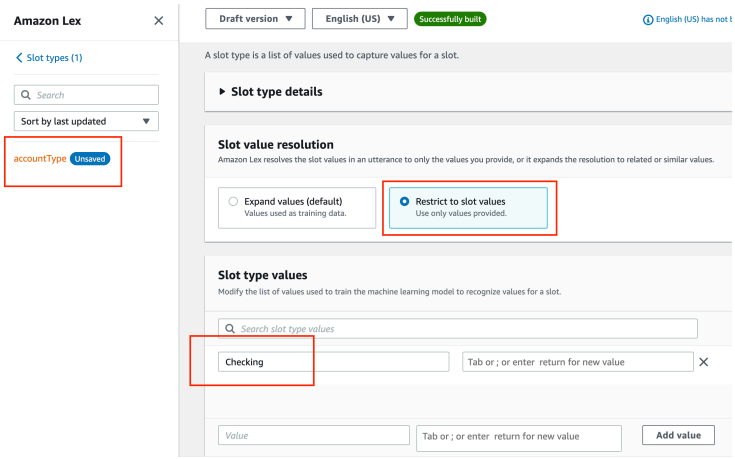
- **Slots** are pieces of information that a chatbot needs to complete a user's request.
  For example, if the intent is to book a table at a restaurant, the chatbot needs specific details like: restaurant name, date, time, number of people. Amazon Lex provides many ready-to-use slot types for common information, like dates and times, but we can also create our own custom slot types to fit our specific needs!

After Choosing **Add slot type,** From the dropdown, chosen **Add blank slot type**.

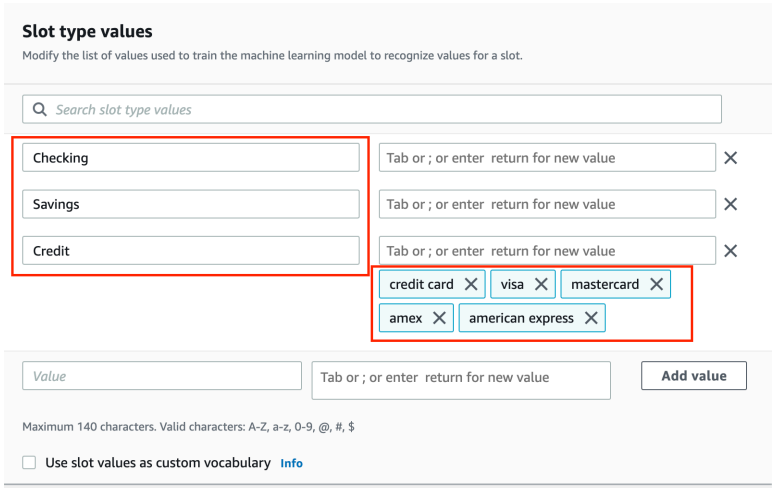
Then Entered accountType for the **Slot type name. Finally,** Chosen **Add** which will bring up a large **Slot types** editor panel. In the **Slot value resolution** panel, chosen **Restrict to slot values**.

- Selecting **Restrict to slot values** makes sure that only the values that you specify will count as a valid accountType! Otherwise, Amazon Lex will use machine learning to accept other values that it sees users constantly entering.
  Different use cases will require different settings, but our BankerBot will only offer customers 3 types of accounts - we don't want Amazon Lex to recognize any alternatives.

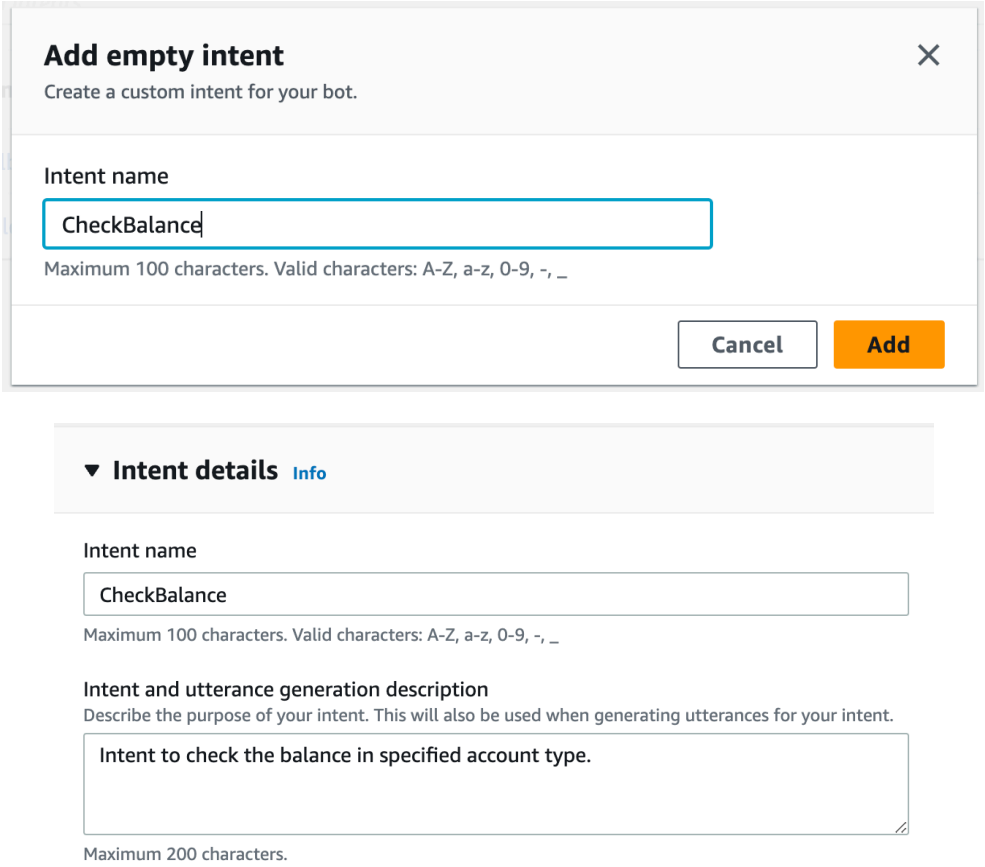
In accountType, **Slot value resolution** was **restricted to slot values**. and **Value** was given as **Checking.**



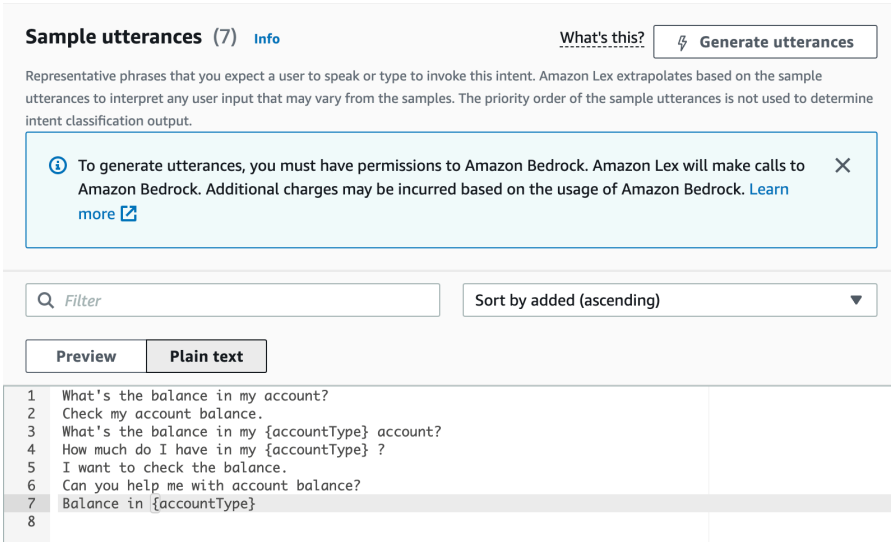
Added few more Slot Type Values and their synonyms. And then saved the slot type.



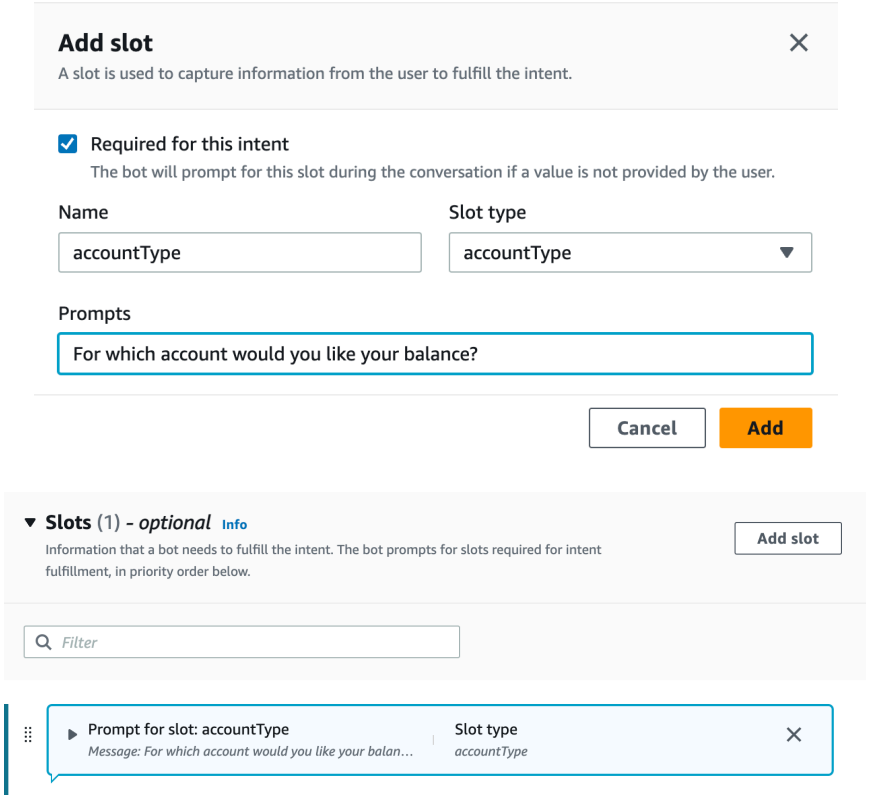
Created a new intent CheckBalance



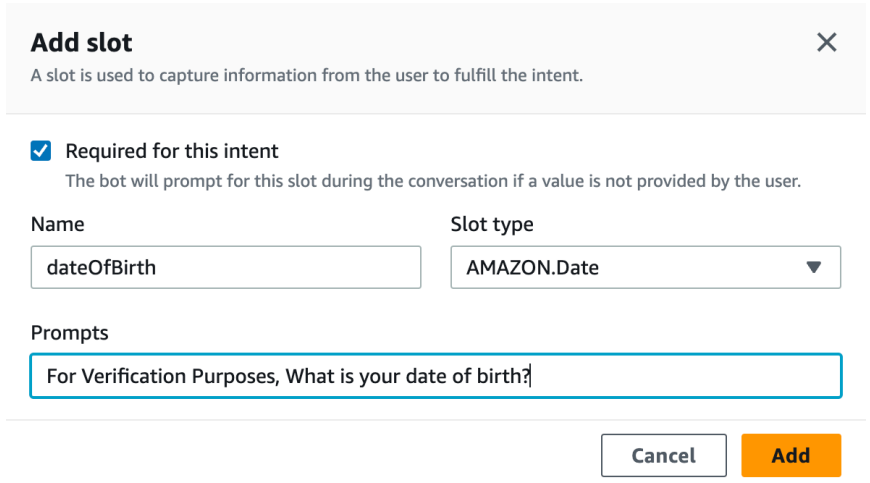
Provided few utterances, the customer may ask to check their balances.



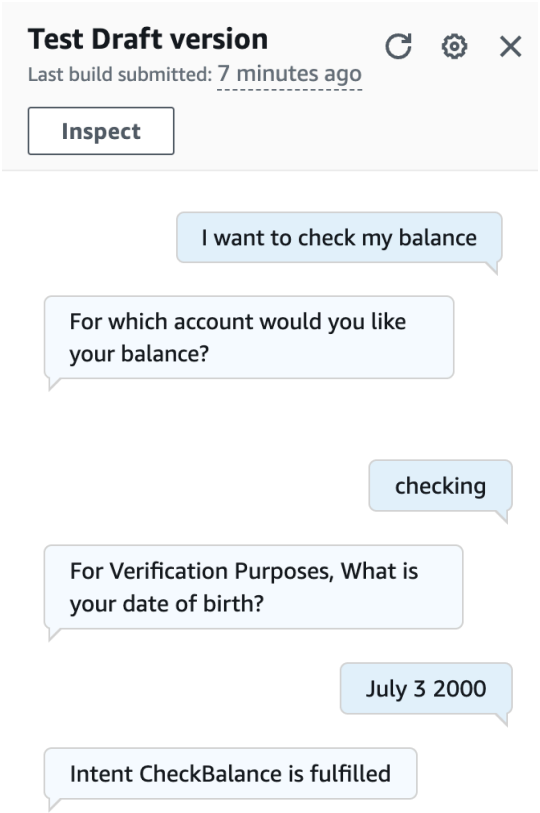
Created a slot, accountType for different account types like checking, savings, etc.



Created another slot, dateOfBirth for Verifying the customer.

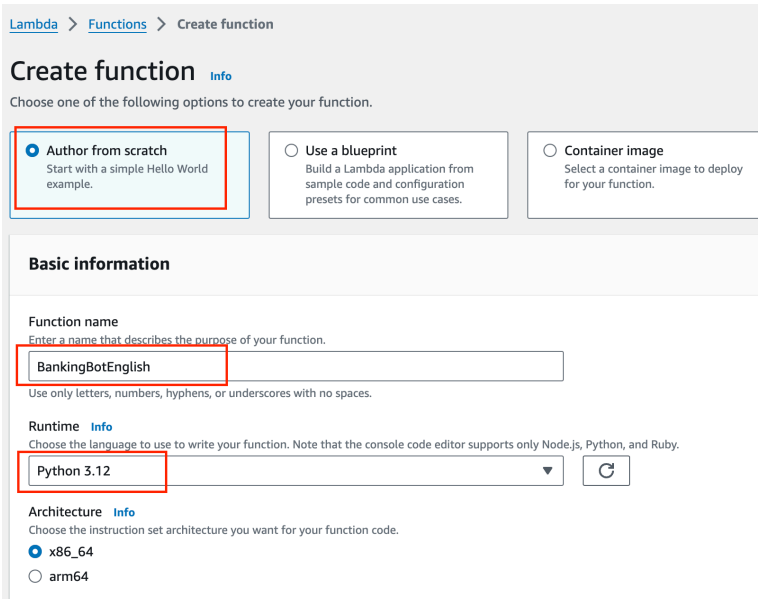


After building and tested the BankerBot with the Utterances and it worked.



Connecting Chatbot with Lambda

AWS Lambda is a service that lets you run code without provisioning or managing servers. Lambda runs the code only when needed and scales automatically, from a few requests per day to thousands per second - all we need to do is supply our code in one of the languages that Lambda supports.

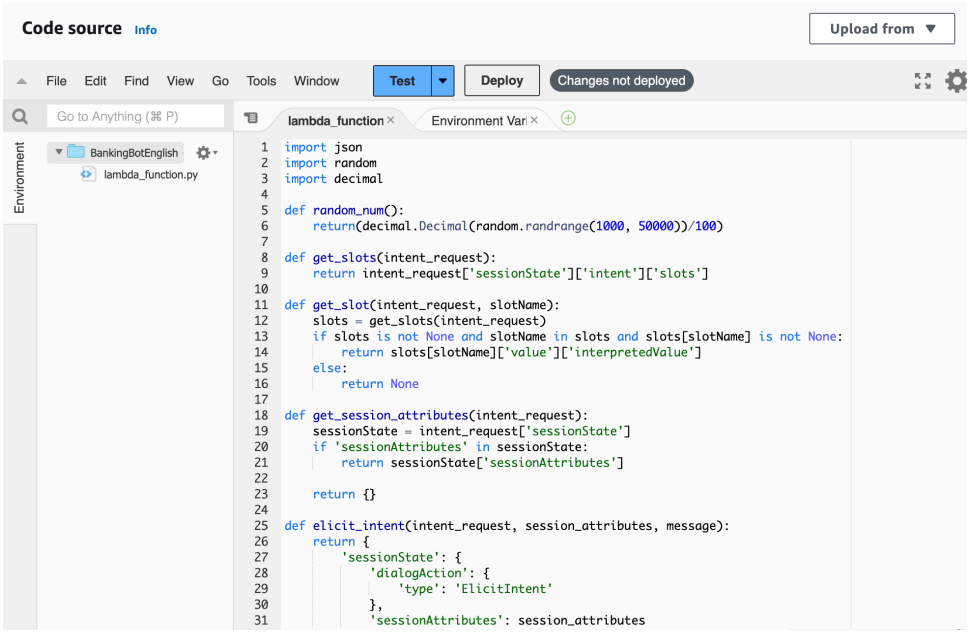


After creating the lambda function, under the code source, given the below code and deployed it to give answers.

```
import json
```

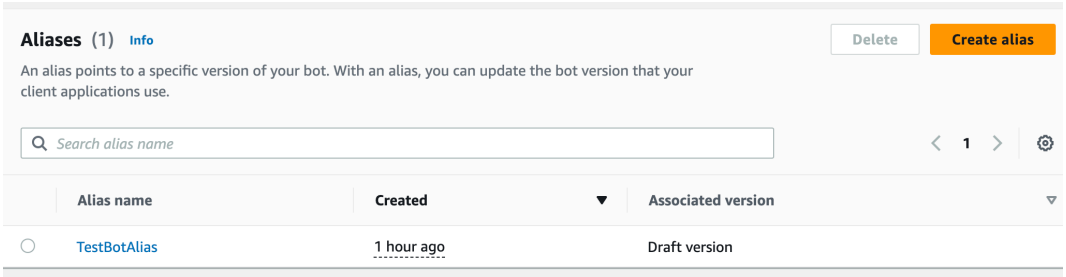
```
import random
import decimal
def random_num():
    return(decimal.Decimal(random.randrange(1000, 50000))/100)
def get_slots(intent_request):
    return intent_request['sessionState']['intent']['slots']
def get_slot(intent_request, slotName):
    slots = get_slots(intent_request)
    if slots is not None and slotName in slots and slots[slotName] is not None:
        return slots[slotName]['value']['interpretedValue']
    else:
        return None
def get_session_attributes(intent_request):
    sessionState = intent_request['sessionState']
    if 'sessionAttributes' in sessionState:
        return sessionState['sessionAttributes']
    return {}
def elicit_intent(intent_request, session_attributes, message):
    return {
        'sessionState': {
            'dialogAction': {
                'type': 'ElicitIntent'
            },
            'sessionAttributes': session_attributes
        },
        'messages': [ message ] if message != None else None,
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes' in intent_request else None
    }
def close(intent_request, session_attributes, fulfillment_state, message):
    intent_request['sessionState']['intent']['state'] = fulfillment_state
    return {
        'sessionState': {
            'sessionAttributes': session_attributes,
            'dialogAction': {
                'type': 'Close'
            },
            'intent': intent_request['sessionState']['intent']
        },
        'messages': [message],
        'sessionId': intent_request['sessionId'],
        'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes' in intent_request else None
    }
def CheckBalance(intent_request):
```

```
    session_attributes = get_session_attributes(intent_request)
    slots = get_slots(intent_request)
    account = get_slot(intent_request, 'accountType')
    #The account balance in this case is a random number
    #Here is where you could query a system to get this information
    balance = str(random_num())
    text = "Thank you. The balance on your "+account+" account is $"+balance+" dollars."
    message =  {
            'contentType': 'PlainText',
            'content': text
        }
    fulfillment_state = "Fulfilled"
    return close(intent_request, session_attributes, fulfillment_state, message)
def FollowupCheckBalance(intent_request):
    session_attributes = get_session_attributes(intent_request)
    slots = get_slots(intent_request)
    account = get_slot(intent_request, 'accountType')
    #The account balance in this case is a random number
    #Here is where you could query a system to get this information
    balance = str(random_num())
    text = "Thank you. The balance on your "+account+" account is $"+balance+" dollars."
    message =  {
            'contentType': 'PlainText',
            'content': text
        }
    fulfillment_state = "Fulfilled"
    return close(intent_request, session_attributes, fulfillment_state, message)
def dispatch(intent_request):
    intent_name = intent_request['sessionState']['intent']['name']
    response = None
    # Dispatch to your bot's intent handlers
    if intent_name == 'CheckBalance':
        return CheckBalance(intent_request)
    elif intent_name == 'FollowupCheckBalance':
        return FollowupCheckBalance(intent_request)
    raise Exception('Intent with name ' + intent_name + ' not supported')
def lambda_handler(event, context):
    response = dispatch(event)
    return response
```



Now **Connected AWS Lambda with Amazon Lex**

Under the deployment in Lex, Chosen Aliases



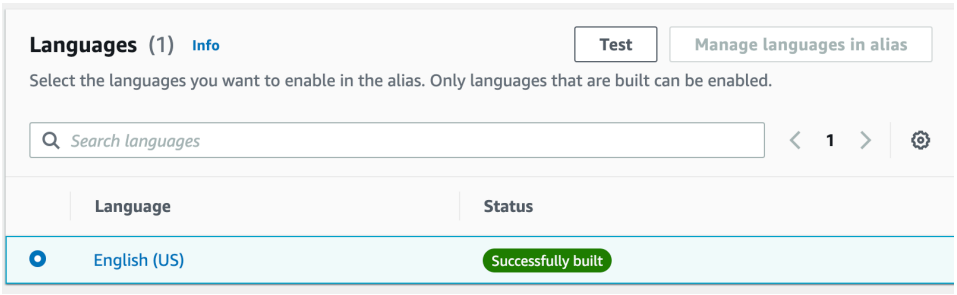
- [ ] An **alias in Amazon Lex** acts as a pointer for a specific version of the bot. So when connecting Lex with other AWS services or custom applications, those external resources will connect to an alias, which will point to the specific version of the bot that we want to use. Now, instead of always updating apps to connect to the newest version of the bot, we can just update the alias to point to that new version. All the apps will automatically start using the updated bot without needing any changes on our end - this saves developers a TON of time and reduces the risk of errors!

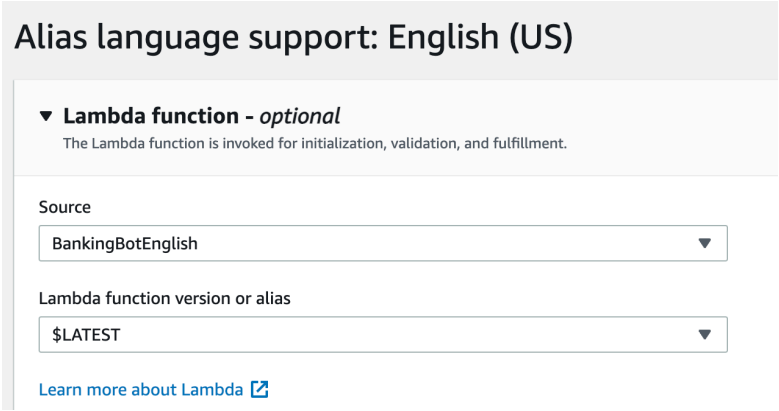
Now, Chosen TestBotAlias

- TestBotAlias is a default version of your bot that's made for testing or development.
  This is the playground version of your bot that you'll use to make sure everything works smoothly before rolling out changes!

Under Languages chosen English, and selected BankingBotEnglish as Lambda Function Source.

- Using the **$LATEST** version means we're directing our alias to always use the most up to date version of this Lambda function. This setup is a time saver that lets us immediately test any changes in our function.





The Lambda function is now ready to work on the BankingBot intents, but we still have to tell Amazon Lex *which* intent will actually use the Lambda function.

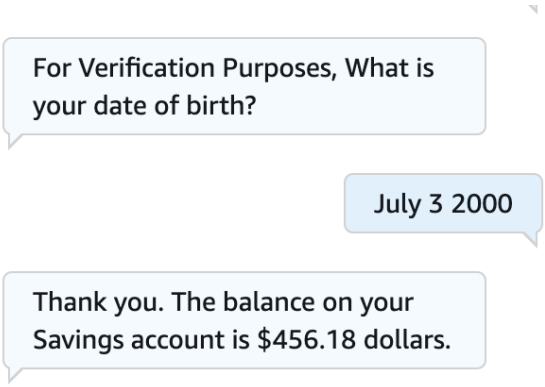
Connecting CheckBalance intent with Lambda function

Under the CheckBalance intent, in the Fulfilment Panel, on Expansion the **On successful fulfilment** bubble. → Chosen **Advanced options** And under the **Fulfilment Lambda code hook** panel, checked the checkbox next to **Use a Lambda function for fulfilment**.

- In Amazon Lex, **fulfilment** means completing the intent. With our BankingBot, after a user tells our bot: The account they want to check, and Their birthday for verification. The bot has all the information it needs and moves to fulfilment. This is where it will use the Lambda function to get the account balance and pass it back to the user.
- **Code hooks** help us connect our chatbot to custom Lambda functions for doing specific tasks during a conversation. They're used to handle more complex actions that the basic chatbot setup can't do on its own, like checking data from a database or making decisions based on past conversations.

Essentially, code hooks make our chatbot smarter and more useful by allowing it to perform these extra steps seamlessly during chats.

After saving the Function, built the bot and tested it and now it responds with the account balance.



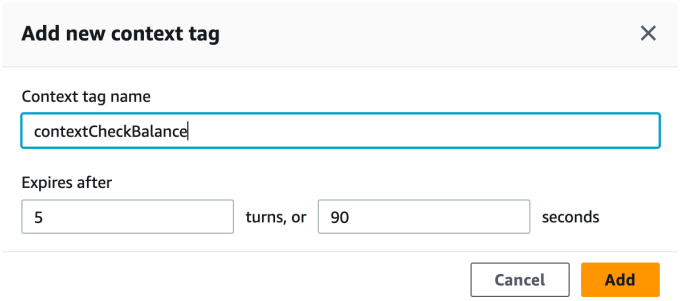
Now it's time save the user info.

Saving User Info with the Chatbot

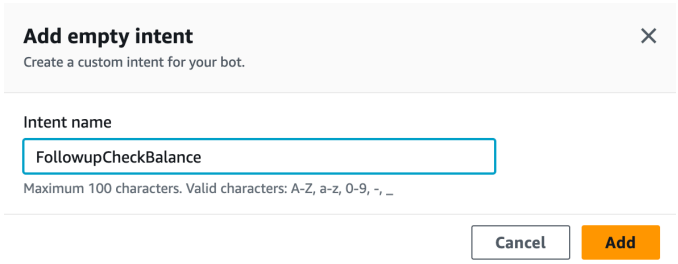
- In your **CheckBalance** intent page, scroll down to the **Contexts** panel.
- Under the **Output contexts** drop-down, choose **New context tag**.

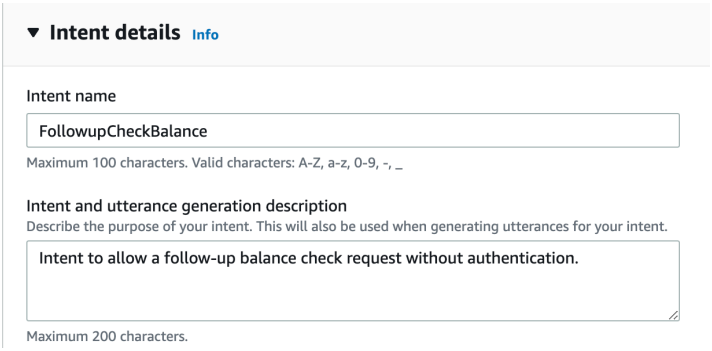
Context tags in Amazon Lex are used to store and check for specific information across different parts of a conversation. They help save the user from having to repeat certain information. There are two types of context tags in Amazon Lex:

Output context tag: This tells the chatbot to remember certain details after an intent is finished, so other parts of the conversation can use this stored information later. For example, the account type from BalanceCheck could be saved and reused.
Input context tag: This checks if specific details are already available before an intent activates. For example, FollowupCheckBalance will check if this conversation already has the user's date of birth saved somewhere, so it won't need to ask for that information again.

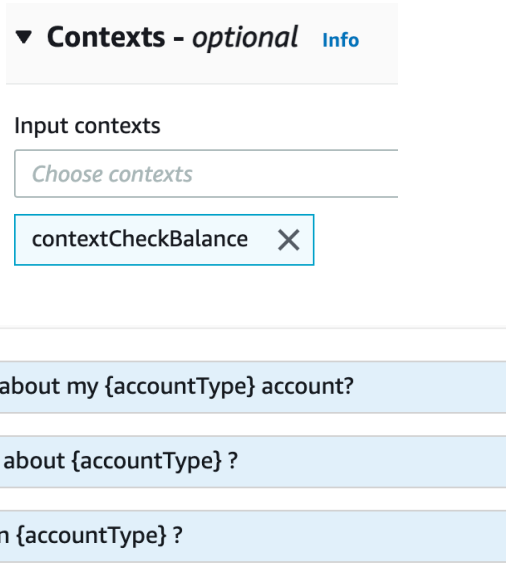


Creating the FollowupCheckBalance intent

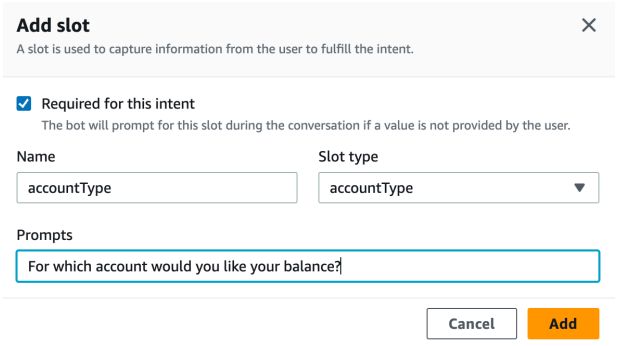


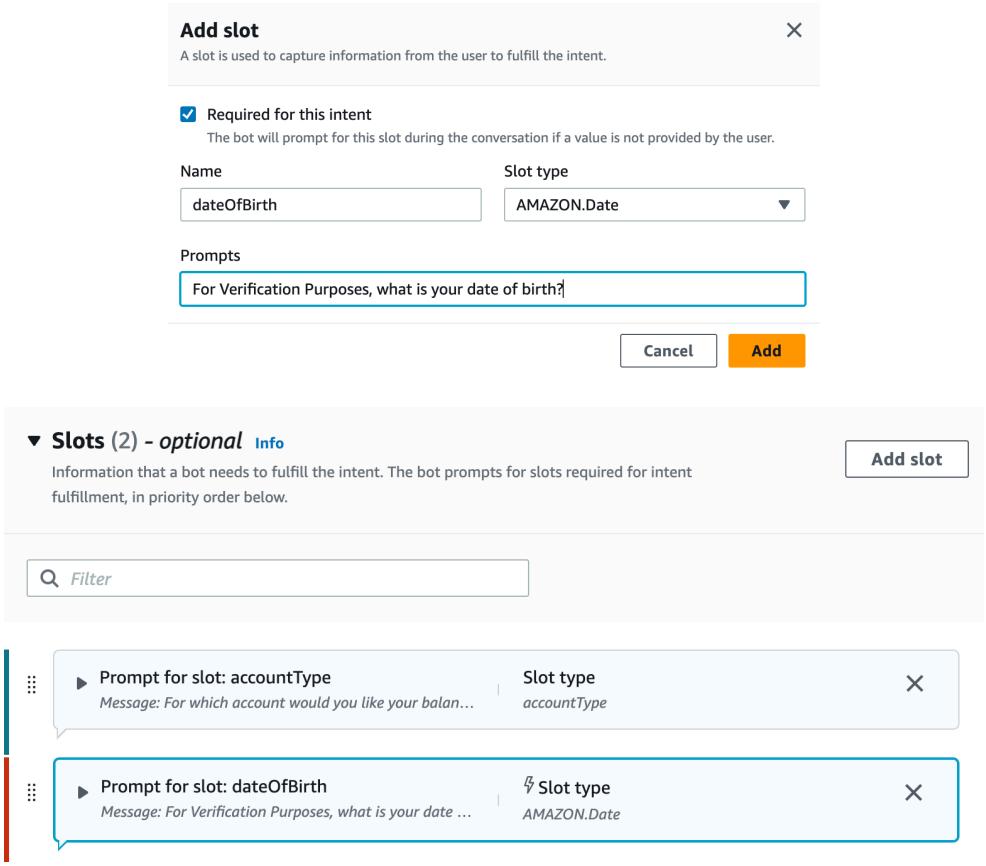


Added Input Context and Utterances for the Intent.



Added 2 slots - **accountType ; dateOfBirth ;**

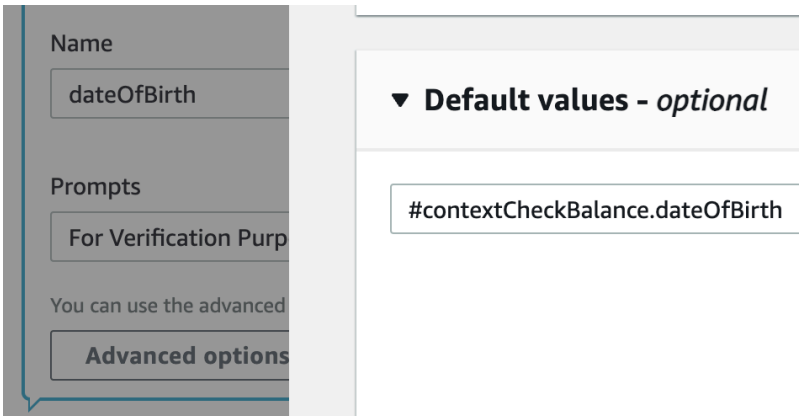




Now, need to tell the FollowupCheckBalance intent where to find the user's date of birth information from the CheckBalance intent!

- In **FollowupCheckBalance** intent page, expanded the **dateOfBirth** slot.
- In **Advanced options ->** Under **Default values** panel (This panel lets us create default values for the intent's slots) Entered *#contextCheckBalance.dateOfBirth* , added default value and updated the slot.

This tells Amazon Lex that the input context contextCheckBalance should have the value of dateOfBirth in CheckBalance.

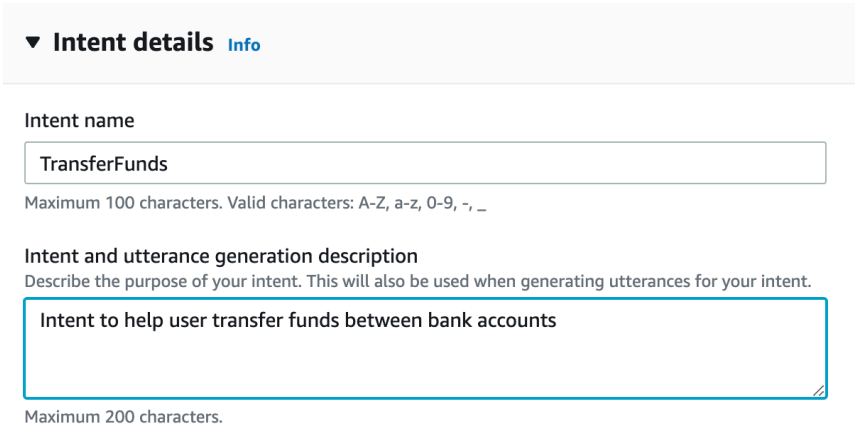


In the Fulfillment section, Advanced Options, Checked the checkbox of **Fulfillment Lambda code hook**

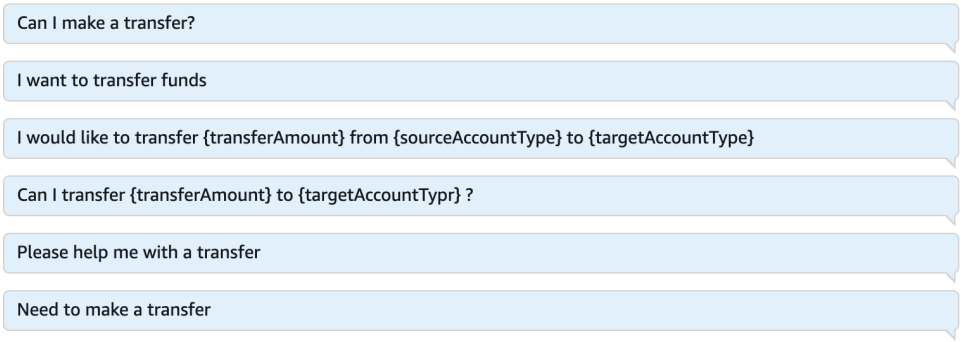
Now the BankerBot, can answer the account balance for multiple account types by verifying

only once.

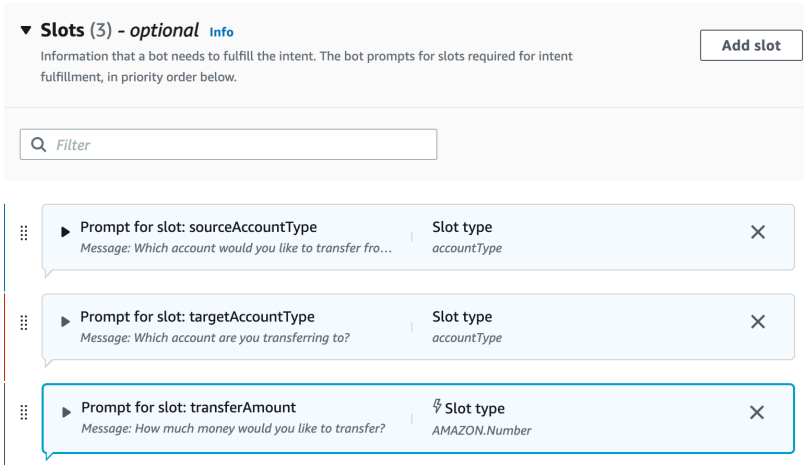
Creating the new TransferFunds intent



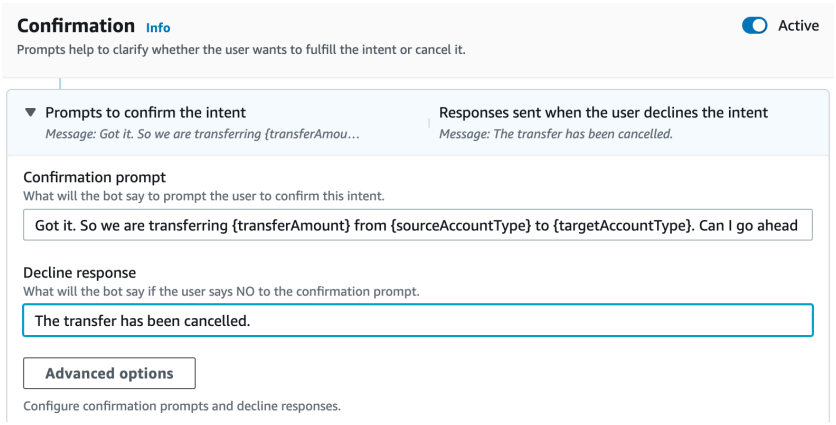
Sample Utterances



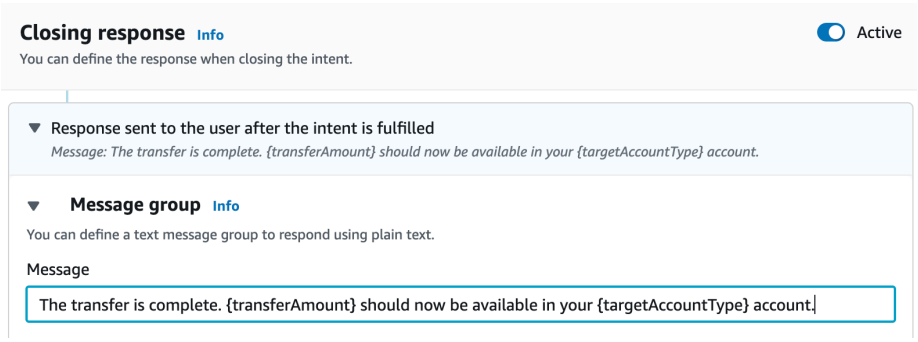
Added 3 Slots - **sourceAccoutType ; targetAccountType ; transferAmount**



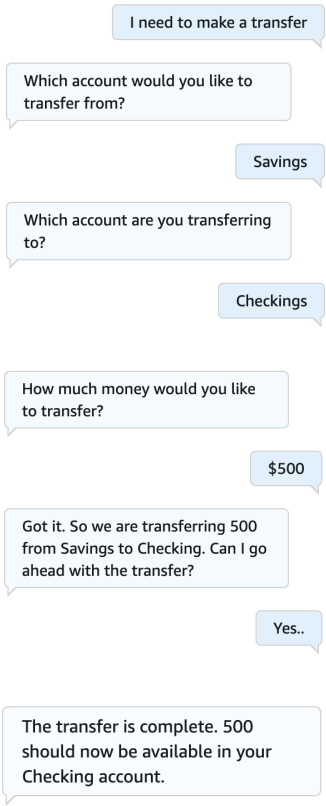
- [ ] Now setting up Confirmation Prompts - **Confirmation prompts** typically repeat back information for the user to confirm. e.g. "Are you sure you want to do x?" If the user confirms the intent, the bot fulfills the intent. If the user declines, then the bot responds with a decline response that you set up.



Added a closing response after the transfer been made.

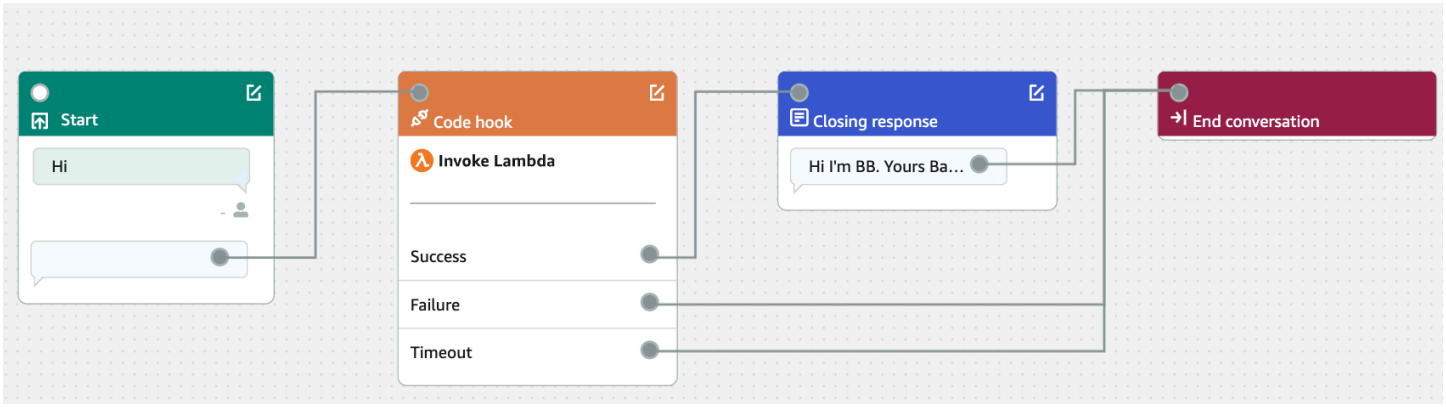


Saved Intent and Built the BankerBot and Tested it for Making Transfers.

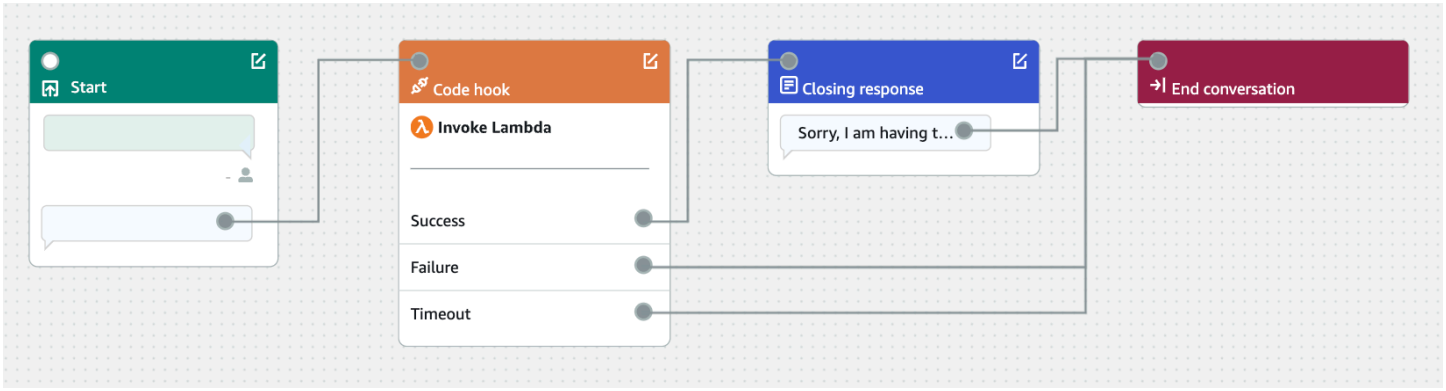


PROJECT

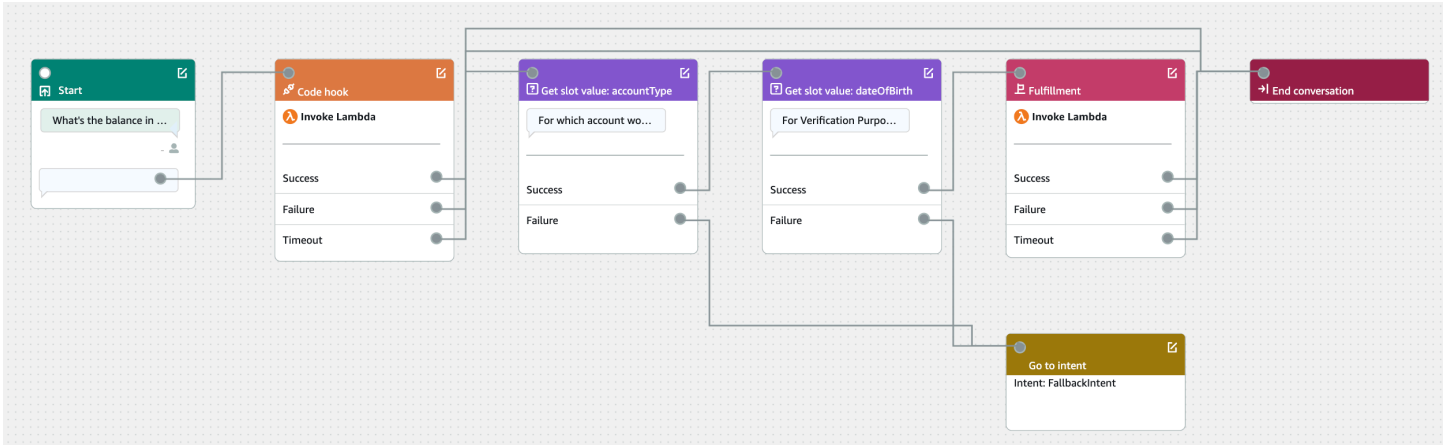
WELCOME INTENT



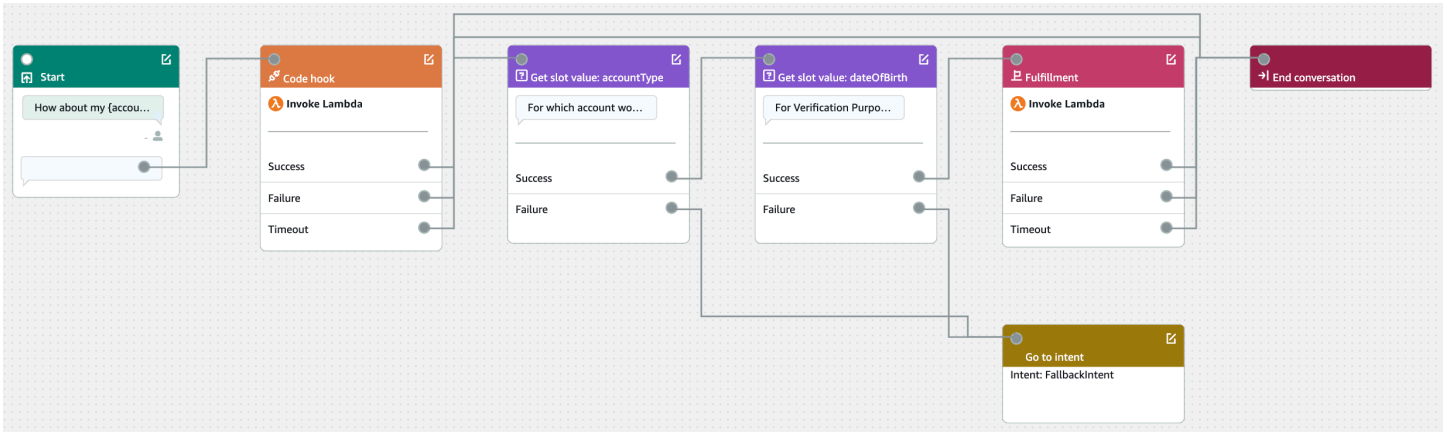
FALLBACK INTENT



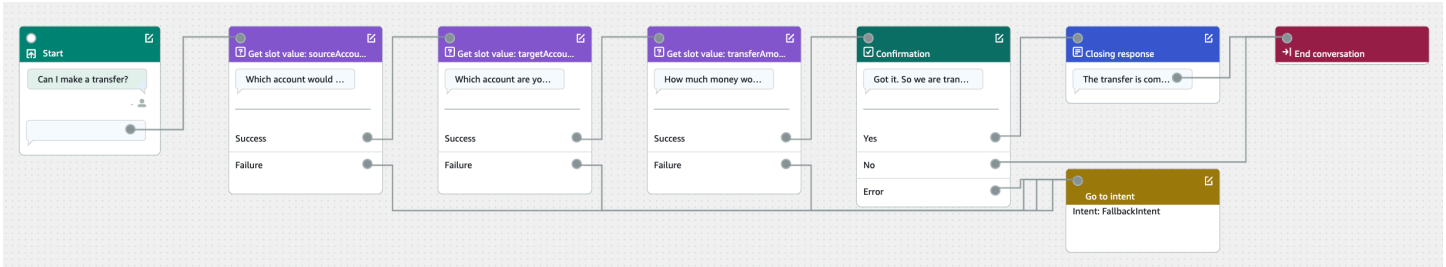
CHECK BALANCE INTENT



FOLLOWUP CHECK BALANCE INTENT



TRANSFER FUNDS INTENT



Final BankerBot Results