highscalability.com

Architecture of Probot - My Slack and Messenger Bot for Answering Questions - High Scalability -

31-39 minutes

I programmed a thing. It's called Probot. Probot is a quick and easy way to get high quality answers to your accounting and tax questions. Probot will find a real live expert to answer your question and handle all the details. You can get your questions answered over Facebook Messenger, Slack, or the web. Answers start at $10. That's the pitch.

Seems like a natural in this new age of bots, doesn't it? I thought so anyway. Not so much (so far), but more on that later.

I think Probot is interesting enough to cover because it's a good example of how one programmer--me---can accomplish quite a lot using today's infrastructure.

All this newfangled cloud/serverless/services stuff does in fact work. I was able to program a system spanning Messenger, Slack, and the web, in a way that is relatively scalabile, available, and affordable, while requiring minimal devops.

Gone are the days of worrying about VPS limits, driving down to a colo site to check on a sick server, or even worrying about auto-scaling clusters of containers/VMs. At least for many use cases.

Many years of programming experience and writing this blog is no protection against making mistakes. I made a lot of stupid stupid mistakes along the way, but I'm happy with what I came up with in the end.

Here's how Probot works....

Platform

- Serverless: AWS Lambda

- Web host: static site on S3, single page app
- Language: Javascript/Node
- API: API Gateway
- DNS: Route53
- CDN: CloudFront
- User login: Cognito
- SSL Certificates: Let's Encrypt
- Server: EC2, t2.small, Ubuntu
- Repository: GitHub
- Source code control: Git
- Payment: PayPal
- Slackbot: Botkit
- Queue: SQS
- Backup: AWS Data Pipeline, saves production data to S3 every week
- Database: DynamoDB
- Node Process Manager: PM2
- SSL Termination: Nginx
- Web Frameworks: jQuery, Bootstrap
- Development Platform: MacBook Air

- Local Test Web Server: http-server
- Tunnel for Testing: Ngrok
- SQS Polling: sqs-consumer
- Logging: CloudWatch

Origin Story

My wife Linda Coleman is an Enrolled Agent, which means she's an awesome tax accountant and all around accounting guru. She had a site, BizTaxTalk, where she gave great answers to people's tax questions: for free. She got a lot of questions. People with real problems needed help and it's hard to find someone to answer those type of questions.

As you might imagine it became overwhelming. She eventually had to give up BizTaxTalk because it took up too much of her time.

I thought why not monetize it? Linda could answer people's questions and get paid for her time. A win-win. I know this type of QA site is not new, but bots are a new spin on the problem and should be a really good way of handling this type of information exchange. The question-answer flow can be very conveniently handled asynchronously using text.

So Probot sets up a two sided market. Users with questions are matched, like a dating service, to a subject matter expert, called a Pro, capable of answering their question. Probot handles the matching, the billing, and coordinating the flow of messages.

Though I'd obviously start with tax and accounting questions, eventually more subject matter experts could be brought on and the number of topics expanded. And that's how I've structured the code. It can easily be extended to handle new topics and question trees.

So that's what I set out to build.

Architecture

Go with what you know. That's advice given over and over on HS. That's what I did. I have experience making Slackbots using the most excellent Botkit on Node, so that's where I started.

I also have experience using AWS Lambda to develop Alexa skills and no desire to manage anything--years of playing sysadmin are enough--so I went with AWS. If Google Cloud Functions had been GA I might have gone with Google Cloud.

Here are all the topics we'll cover:

- ProbotService Lambda Function
- Slackbot
- Messengerbot
- Website
- PayPal Integration
- Testing
- Adoption
- Lessons Learned

ProbotService Lambda Function

This is the AWS Lambda function that implements the backend of the Probot Service. Mostly. And it's also the source of many of my stupid mistakes. Let me explain.

The first Probot bot I programmed was for Slack, and this getting a little ahead, but the Slackbot runs in an Ubuntu process on an EC2 instance.

A big mistake I made is not going API first. Much of my career has been building messaging protocols of one sort another. Did I do that? Of course not.

I programmed all the bot logic so it executed directly in the Slackbot process. This is possible because AWS has a nifty Node API for accessing DynamoDB, SQS, Lambda and other AWS services. So all AWS access was coded directly in javascript. It worked great. No problem at all.

The problem occurred when I needed to make a web client. Well, it wasn’t a problem in the sense the same code worked almost perfectly in the web client too. Which is cool. So I wasted a lot of time making a common library that Slack and web client could both share. But I became very uncomfortable with the idea that my database access code would be visible over the web. For an intranet service I wouldn’t worry about it, but the attack surface of showing your data structures is way too large for my liking.

Another big mistake I made, also related to not creating an API first, was falling in the love with DynamoDB’s ability to pass in new and old values to a Lambda function on a database changes. What I did was trigger logic like a state machine. If the old record said the state was PENDING, for example, and the new state was ASSIGNED the PENDING-to-ASSIGNED transition was triggered. It worked and I thought it was really clever. The problem was it was too brittle in the face of a lot of operations that didn’t move the state yet still triggered the Lambda function.

The solution to both problems was to build an API, which I did using Lambda. I chose not to use API Gateway because it’s dead easy to call Lambda functions directly from Node. API Gateway adds latency, complexity, and cost. It’s also hard to use. Transforming arguments from an HTTP request and mapping them to a Lambda invocation is a process very close to magic for anything complex. Same in the reverse direction for replies. If a public API is needed in the future this decision can be revisited, but just using Lambda directly is quite straight-forward. All Slack and web code had to be changed to use the new API. What a pain.

How Do You Structure a Serverless API?

The next decision is how do you structure the API? Does each API call map to a separate function? Or do you group functions together behind a single entry point? Or possibly multiple entry points?

I chose having a single entry point. The reason I went with this approach is because I didn't want to use a framework to manage Lambda. I want to know how everything is wired together and frameworks contain a lot magic. So I create a zip file and upload it to either Production or Test using the AWS console. In the future I'll automate it, but it works fine for now.

A question when using a single entry point is how do you multiplex function calls? I chose to mimic JSON-RPC, which I've used successfully in the past for server applications.

Requests are wrapped in JSON-RPC payloads. The method to invoke is specified using the "method" property and any parameters are passed using the "params" property. The ProbotService index.js file unpacks the JSON-RPC request and calls the correct method. It looks something like:

```
if (request.method === "giveAnswerToUser") {
  var op = new QuestionImpl(appcfg);
  op.giveAnswerToUser(request.params, function(error, data) {
    return sendResponse(error, data, request, callback);
  })
}
```

In standard RPC fashion I created a stub object called

ProbotService that makes calling Lambda functions much easier from client code. For example: *service.getQuestion(id, callback)* formats the correct JSON-RPC structure, invokes the proper Lambda function, and invokes the callback when a reply is returned.

Sharing Code

I chose not to invoke ProbotService Lambda in the Facebook Messenger bot code. Instead, the service library code is packaged with the Messenger code and is called directly. This is a code sharing approach. The same underlying code is shared across ProbotService and Facebook Messenger.

The reason is the Facebook Messenger webhook invocation goes through API Gateway which calls a Lambda function. Using ProbotService also calls a Lambda function. So a Lambda function would be invoked within another Lambda function. This works fine, but the worst case latency is higher than I wanted users to experience, so I chose to remove a hop by sharing the code. I'm still not sure this was the best decision, but it does work and the Messenger bot response time feels snappy.

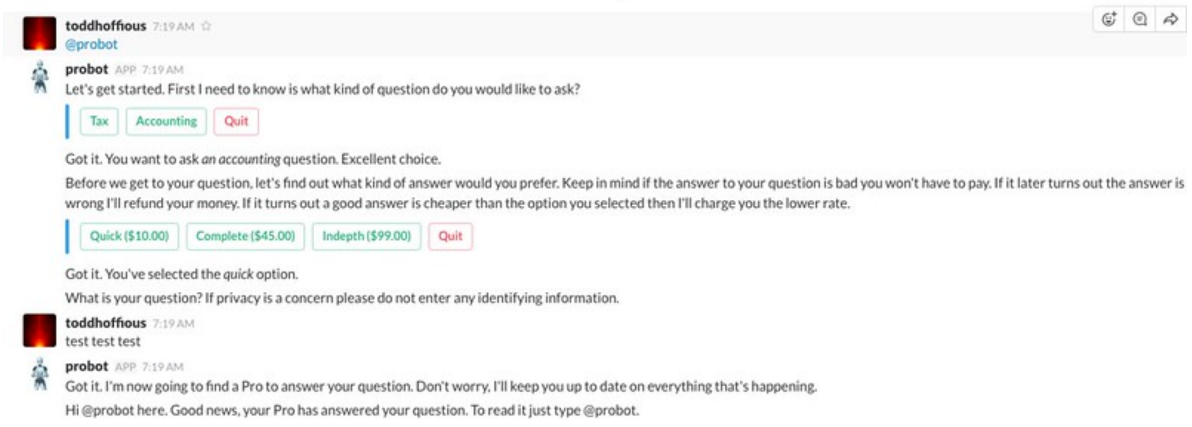
Since Slackbot runs in its own process there's no extra hop so using ProbotService is the right thing to do. ProbotService is also used in the web client code, so no internal details are leaked.

ProbotService API

Here are a few of the functions in the Probot API. I don't use anything like Swagger as documentation.

- updateFromPaypalWebhook
- getQuestion
- giveAnswerToUser
- giveAnswerToPro
- askQuestionOfUser
- cancelQuestion
- acceptProAnswer

Slackbot


toddhoffious 7:19 AM
@probot

probot APP 7:19 AM
Let's get started. First I need to know is what kind of question do you would like to ask?

Got it. You want to ask *an accounting* question. Excellent choice.

Before we get to your question, let's find out what kind of answer would you prefer. Keep in mind if the answer to your question is bad you won't have to pay. If it later turns out the answer is wrong I'll refund your money. If it turns out a good answer is cheaper than the option you selected then I'll charge you the lower rate.

Got it. You've selected the *quick* option.

What is your question? If privacy is a concern please do not enter any identifying information.

toddhoffious 7:19 AM
test test test

probot APP 7:19 AM
Got it. I'm now going to find a Pro to answer your question. Don't worry, I'll keep you up to date on everything that's happening.

Hi @probot here. Good news, your Pro has answered your question. To read it just type @probot.

Bot Style

One of the decisions you have to make when programming a bot is: do you want a conversational AI style bot or a structured command driven style bot.

I went with the structured command driven approach. A user selects a question topic and the bot guides them through a question tree to gather data the Pro needs to answer that type of question. For example, for a tax return you need to know if what year it's for, if it's personal or business, if it's business what entity type is it (partnership, c-corp, etc), and so on.

The AI style is sexy, but it's really hard to do well if that's not your thing. So I came up with a way of describing a dialogue with the user that could be customized for each question. Actually I came up with several different dialogue descriptor solutions and I wasn't happy with any of them. This is what one looks like for Slack:

```
probot.machine.add("ask-return-type", {
    action: probot.machine.askFromOptions,
    options: ["individual", "business"],
    display: ["Individual", "Business"],
    question: "When answering tax questions it's helpful
for your expert to know the kind of return.",
    sayOnAccept: "Got it. You've selected the
_{response}_ option.",
    answer: function (machine, cmd, answer, next) {
      console.log("ask-return-type:answer:" + answer);
      probot.dto.setReturnType(answer);
      if (answer === "individual")
        cmd.next = "ask-tax-year";
      else
        cmd.next = "ask-entity-type";
      next(null);
    },
  })
```

I went a completely different direction for Messenger. I hate that way too.

Slack Host

With Slack webhooks can be used to implement Slack commands, but the conversation style requires code to run in a process context. So I a set up a small EC2 instance without any redundancy. Since I don't expect users to constantly engage with Probot on Slack I figured a little down time wouldn’t hurt. All state would be in DynamoDB, so nothing would be lost. An Elastic IP is used to access the host.

Keep it simple and don’t overcomplicate things. Another long time lesson on HS. If anyone actually ends up using your system you can do all the fancy stuff later.

PM2, an excellent Node process manager, is used to restart the Slack process if it dies. PM2 also manages logs and does some other nifty things. Route53 pings the process and emails me if anything goes wrong. Good enough.

Botkit does a lot of the heavy lifting. It handles all the protocol work between your program and Slack while providing a lot of convenient abstractions for authentication, carrying out conversations, that sort of thing.

Interactive Messages

After I was completely done with my Slackbot implementation Slack introduced the ability for users to make selections using buttons instead of entering text. Since I hadn’t released anything yet that was the time to make the change. The buttons do make it easier for the user. The problem is the buttons require your process implement an HTTPS webhook so Slack can callback into your process with information on which button was pressed. This is a major change to the development flow. Slack development was nice and contained before. Now your bot process has to be externally accessible via a URL.

On the production machine in EC2 it wasn’t so bad. I use Nginx to terminate SSL and pass through the request to the Slackbot process. This requires the use of SSL certificates for which I use Let’s Encrypt. As they have to be regularly updated they’re a maintenance pain, but they're free, so pick your poison.

Letting Slack access a process running on my laptop through a Comcast router was not so easy. I wasted a lot of time with various solutions. A lot of time. I eventually gave in and paid Ngrok for their tunneling service. Worked like a charm. Spend the money. Slack should consider being able to work completely over Serverless. That would be a much cleaner solution.

Identity and Authentication

There's no attempt in Probot to create a user that is common across Slack, Messenger, and the web. Each uses the identity mechanism native to its environment. For Slack that means using the team ID and the user ID assigned by Slack. Probot asks a user for their name and email address, but otherwise doesn't bother with identity and authentication. Slack handles all that.

Database

Slack provides their own database abstraction for storing teams, users, and channels. As Probot uses DynamoDB I created a simple adapter that allows Slack to store its data in DynamoDB.

It's common practice to insert your own properties in Slack's data structures. So user data is stored in Slack's user object.

Sending Messages from the Pro to the User

When in a conversation with a user sending a message with Botkit is as easy as *bot.say("some string")*. Sending a message asynchronously to a user at some later time is not so easy. For example, when a Pro has an answer for the user, or wants to ask a question, the user must be given the message.

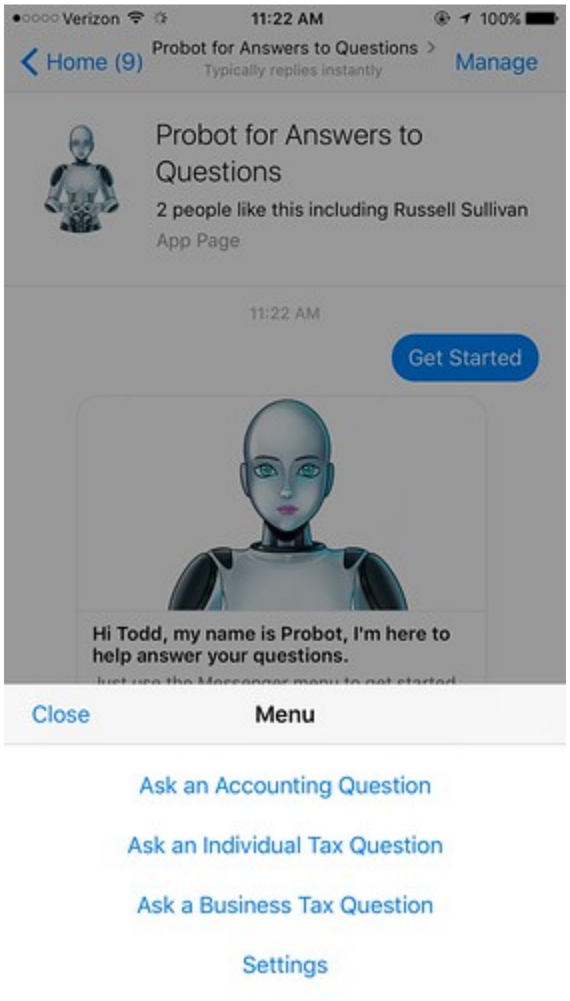
The only way I could figure how to make async messaging work:

- When a team connects to Probot store the Botkit bot object for the team in memory. A bot object is needed to send a message to a user.
- Each user is associated with a team so when a message comes in the proper bot object can be found.
- All messages from a Pro are queued to AWS SQS.
- The Slackbot process uses a nifty package called sqs-consumer to poll SQS for messages.
- Messages are sent to the user using the bot object.
- The user calls @probot to engage in a conversation to process the message. The message could be an answer, question, or status update. The user can rate the answer and leave a comment. They can also reject

the answer which means the user doesn't have to pay. I don't want people to feel like they have been ripped off. This is a risk the Pro takes. Though I don't do this yet the matching algorithm can take advantage of the rating data when making matches.

Sqs-consumer is an example of why Node is so productive for developers. Yah, javascript kind of sucks. Yah, callback programming kind of sucks. But it literally took me five minutes from the thought *I need to poll SQS* to finding a good npm installable package and having working code. That happens all the time with Node.

Messengerbot


Verizon
11:22 AM
100%
Home (9)
Probot for Answers to Questions
Typically replies instantly
Manage
Probot for Answers to Questions
2 people like this including Russell Sullivan
App Page
11:22 AM
Get Started
Close
Menu
Ask an Accounting Question
Ask an Individual Tax Question
Ask a Business Tax Question
Settings

Hi Todd, my name is Probot, I'm here to help answer your questions.

Messenger is a completely different beast than Slack. Though Botkit has support for Messenger I decided to code directly to the Messenger API because it's much simpler than Slack. That's because as a team collaboration tool Slack has a lot capabilities. Slack has all sorts of APIs for working with teams, channels, users, groups, permissions, capabilities, and so on.

With Facebook Messenger messages are received over a webhook and formatted messages are sent using an HTTP call. That's pretty much all you can do. As a consequence a Messenger bot can run completely on Lambda functions. If you don’t care about all the team stuff Messenger may be a better platform to start on.

In retrospect this is where going with what you know bit me. Probot is not a team tool. It interacts with users one-on-one, so all the overhead is of no benefit and took a long time to work out. On the other hand the people at Slack are an absolute delight to work with. If you have a question a real person with technical chops will give you a good answer. Can you imagine?

In fact, I’ve been thinking of pulling support for Slack because nobody is using it and that EC2 instance is costing me every month. Serverless wins big time for this type of workload. Using Lambda I pay only when people are actually using Probot.

Simplifying for Messenger

One thing that immediately became clear is that Probot had to be simplified to work well on Messenger. With Slack complicated user interactions felt natural. On Messenger complicated question trees don’t work at all.

One simplification I made was to reduce the answer types a user can request from a Pro. On Slack a user can ask for a Quick, Complete, or an In-depth answer to a question, each at different price points. Adding this extra layer of interaction bogged down the whole process so with Messenger the only possible answer type a Pro can give is the Quick type.

Further reinforcing the need for short and simple is the inability to send long strings of text to Messenger. Slack is great with text. You can spit out pages of text no problem. With Messenger there is a max size you can send. Long strings have to be broken into chunks, not a good look.

Another simplification is on Messenger a user can only have one question outstanding at a time. On the web and on Slack a user can have multiple questions open at one time. In retrospect this is probably how I should have built Slack as well. Managing multiple outstanding questions through a chat interface is complicated.

That’s another long time lesson on HS: Keep it Simple Stupid. A lot easier said than done.

Facebook Messenger Webhook

In the Messenger bot configuration a webhook is specified that Facebook calls with user related events.

The webhook is implemented using API Gateway tied to a Lambda function. The Lambda function is implemented using a shared code approach. It includes all the source code it needs rather than calling out to other services.

The startup code in the Lambda index.js file is responsible for parsing the message from Facebook and dispatching it correctly. Before each message is processed startSession is called with the sender ID, which is used as the user ID. startSession gets the user record if exists, if it doesn’t exist the profile for that user is requested and a user record created. If there’s an existing question ID in the user record that question is retrieved from the database and a state machine object for the question is created. The incoming message is an event on the state machine and drives how the message is processed.

Using startSession all data needed to process a user message is in place so none of the lower level code has to worry about it. Likewise when a message is done being processed dirty flags are checked and if an object has been changed it is automatically updated in the database. I found this style makes using Lambda functions much cleaner. Lower level code never has to worry about state management.

This is an area where Slack has an advantage over Facebook. Lambda functions can't store state so all state must be activated and passivated with each request. With Slack state can be stored in the Slackbot process, though if I had it to do over again I'd probably not keep state in Slack. It's very convenient to go to the database and edit records and have the changes be picked up on the next request. For those changes to be seen in Slackbot the process has to be restarted.

Messenger has a number of different types of messages: message, postback, authentication, delivery confirmation, message read, account link, and so on. The message type has several subtypes: quick reply, text, echo, receipt, read receipt, typing on, typing off, and so on.

Generally the only messages I pay attention to are interactions with the user. All sources of commands are mapped to a common request format and fed into the Question state machine. For example, a postback event happens when a user taps a postback button, Get Started button, Persistent menu or Structured Message. Any pure text input from a user is something I assume is in response to a question I asked previously. And quick reply is just another type of user tappable button.

Facebook Question State Machine

All questions are stored in DynamoDB. Each question has a property specifying the current state the question is in. After a message has been normalized into a common request format it is applied to a Question state machine, which looks something like:

```
var QuestionDto = require("./QuestionDto");
function FacebookQuestionSm(appcfg, startState) {
  console.log("FacebookQuestionSm:startState:" +
startState);
  this.startState = startState;
  var self = this;
  this.any = {
    help: {
      action: FacebookQuestionSm.sendHelp,
    },
    getstarted: {
      action: FacebookQuestionSm.sendGetStarted,
    },
    settings: {
      action: FacebookQuestionSm.sendSettings,
    },
    status: {
      action: FacebookQuestionSm.sendStatus,
    },
  ...
  this.states = {
    start : {
      on : {
        accounting: {
          forward: "ask_accounting_question"
        },
        ask_accounting_question: {
          action: FacebookQuestionSm.createQuestion,
          next: "ask_question",
          data: { type: "accounting", msg: "Great choice, I
see you want to ask an accounting question." }
        },
        ask_individual_tax_question: {
          action: FacebookQuestionSm.createQuestion,
          next: "ask_tax_year",
          data: { type: "tax", returnType: "individual",
                  msg: "Great choice, I see you want to ask a
tax question for an individual."
          }
        },
        ask_business_tax_question: {
          action: FacebookQuestionSm.createQuestion,
          next: "ask_entity_type",
          data: { type: "tax", returnType: "business",
            msg: "Great choice, I see you want to ask a
business related tax question."
          }
        }
      }
    },
    ask_tax_year : {
      action:
```

```
FacebookQuestionSm.sendTaxYearQuestion,
     on : {
       text: {
         verify: FacebookQuestionSm.verifyTaxYear,
         action: FacebookQuestionSm.saveTaxYear,
         next: "ask_question"
       }
     }
   },
   ...
}
 this.on = function(appcfg, userid, request, next) {
   console.log("on:userid:" + userid +  " request:" +
JSON.stringify(request));
}
```

The *on* method takes the request and applies it to the state definitions. This works OK, but handling all the ways a user can interact with the bot makes for some hacky looking code.

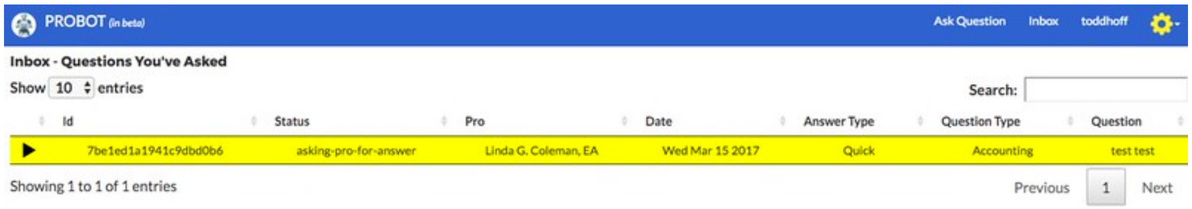
Identity and Authentication

Like for Slack, Facebook provides a user ID specifically for Messenger and I just use that ID. It’s not a real Facebook user ID, so you can’t use the graph API to get a lot of user information. That ID is stored with the question so replies can be texted directly to the user.

Database

There’s no equivalent of Slack’s database API for Facebook, so user data is stored in a Facebook specific user table in DynamoDB.

Website



Inbox - Questions You've Asked

Show 10 entries

Search:

	Id	Status	Pro	Date	Answer Type	Question Type	Question
▶	7be1ed1a1941c9dbd0b6	asking-pro-for-answer	Linda G. Coleman, EA	Wed Mar 15 2017	Quick	Accounting	test test

Showing 1 to 1 of 1 entries

Previous 1 Next

Probot.us is a single page app hosted on S3. As a single page app everything is done using the same ProbotService API used by Slack. The advantage of using S3 is I don’t have to manage anything. It’s a website much like any other, so I won’t go into much detail on it, but there are a few topics worth noting.

The Email Debacle

Have you ever done something you know wouldn't work in the end but you did it anyway? This was another great time waster of a mistake. I'm not very good at making websites, as you can no doubt tell. Which I hope explains why I really tried to avoid making one at all.

What I did was make all the Pro interactions work over email. Questions would be sent to Pros over email. Pros would reply with answers and questions over email. Keywords in the email made it so data could parsed and extracted. User's didn't notice a difference because they would still use Slack or Messenger.

Here's the flow: email was sent by Probot to a Pro. For example, when a question is assigned to a Pro they would get an email containing the question, information about the user, and some bookkeeping attributes needed to map the email back to the question. The Pro would reply with their response, making sure to insert text in the right places so it could be parsed correctly. The Pro's reply was received by AWS and saved into S3. That caused a Lambda function to be called. The function would parse the email, extract out all the data, and update the user accordingly.

This actually worked and was fun to code, though a pain to configure. Only my test Pros hated it. Just hated it. Of course they did, it sucked for them. So I ended up making the Pro Dashboard to manage all a Pro's questions that I always knew I was going to have to build. I also added an Inbox so users could ask and manage questions over the web. The same website also acts a landing page for Slack users so they can install the bot into their team.

Cognito

Though I had a complete user registration system already working in Golang, I wanted to keep everything in Node. Rather than reinvent the wheel I decided to give Amazon Cognito a try. I didn't want to be responsible for user password security and Cognito handles all that. It's tricky to use and there were some strange problems to figure out. The example code and documentation could be better. But I eventually got all the typical flows--registration, forgot password, change password, resend verification code, enter verification code, etc--working. It was not easy, but it seems to work reliably.

Sharing Code Between Node and Browser

This is another mistake I made: I didn't plan to share code between Node and the browser. By the time I realized I could share a lot of code it was too much work to adopt a different module philosophy. I do share code, but I have to be very careful about about including code that brings in dependencies that can't work in both contexts.

PayPal Integration

I went with PayPal because I found their API understandable, their dashboard workable, and their documentation useable. Support is slow, but generally helpful.

When a user accepts a Pro's answer to their question the payment state machine kicks in. In ProbotService the PayPal API is used to create an invoice which is sent to the user's email address by PayPal. PayPal lets you configure a webhook for consuming invoice related events. The webhook uses API Gateway tied to a Lambda function.

When the Lambda function receives invoice events it will notify both the user and the Pro as things happen.

- Slack: the user is notified via the SQS mechanism described earlier. The Pro is sent email that let's them know they should go visit their dashboard for details. No sensitive information is ever sent in email.

- Messenger: a text message is sent to the user. Note, to send messages to a user 24 hours after your last contact you must have special permission for your bot. This is not always easy to get. So if your bot requires this kind of functionality plan ahead.
- Web: email is sent to the user and the user is asked to login to Probot to get more information on their dashboard.

This took some time to get working. When the Lambda function associated with the webhook isn't called it's very difficult to understand what's the problem. Definitely turn on API Gateway logging. You'll need it.

The PayPal Lambda function uses the same code sharing approach described earlier. Instead of making a remote ProbotService call the code is called directly.

Testing

Nothing fancy here.

For API Gateway, Lambda, and DynamoDB there are test and production versions of everything. Which to use is selected at runtime based on configuration.

For the website there is a test and production versions. To test them I just use all the features. Files are copied from a development environment to S3 using the *aws s3* command line. For local development http-server is used as the webserver and all features can be tested through localhost.

PayPal has test and production configuration options. Each points to the test and production webhooks respectively. PayPal has logs that log when webhooks have been invoked, which is helpful when debugging. Unfortunately events are not sent in real-time so you have to wait for events to be delivered and for the logs to appear.

For Slack there is a separate bot configured for test and production. Each points to the test and production webhooks respectively. The same for Messenger. There’s no cleaner way that I know of other than treating test and production versions as completely different bots.

For local development a Slackbot runs on the development machine. Ngrok is used as a tunnel so Slack can send interactive messages to the process. A test bot is installed into a team using the test version of the website. It has the proper app token to identify the test version of the bot. After the test bot is installed you can just talk to the bot through Slack.

For local development testing Lambda code is made a lot easier by coding sharing. The only external services that will be called are other people's services, not your own. So to test it's not necessary to upload code into Lambda at all. The same code used in the Lambda function can be called from a script executable from the command line. For every function in your API make a corresponding script and it can be completely debugged locally.

Local development testing of a Messengerbot is trickier because when you send a message to a bot the message shows up in Messenger. And when you tap the button the reply goes back to your test webhook. So as far as I know there's no clean way to do this in a unit testable fashion. What I did was install the test version of the bot into Messenger and record the user ID. When making the test script for each scenario, which user ID to use, the test or production version, is selected at runtime. If a scenario involves asking the user to answer a question the question will be sent from the script. Then you go to your phone and use the test bot to make the reply. The reply will go to your test webhook, so that code should be functional. I'm not really happy with this approach, but it sort of worked, and was quicker than testing everything through Messenger.

Someday everything should be installed from GitHub, but today is not that day.

Adoption

Uptake has not been good. I'm hoping as tax season draws near activity will pick up. Part of the problem is as a programmer I pretty much suck at marketing. I'm trying Messenger Bot ads but those aren't working.

Using the funnel metaphor users who give the bot a try don't convert, they bail before they actually convert and ask a question. It's probable there's a trust issue. They don't know who they are asking or what kind of answer they will get. I try to address those concerns on the website and on the bot page. You don't have to pay if you don't like the answer, for example, so it's risk free.

One possibility is people aren't really using Messenger bots and it's a failed experiment for this sort of application.

Another possibility is my bot isn’t very good. Something that is entirely possible.

If you have any thoughts or questions please let me know.

Lessons Learned

- **Don’t do what you know**. Before blindly doing what you know take a look around and see if there's a better option.
- **API First**. Come up with an abstraction for your service and implement it in one place. You'll thank yourself when you have to implement another client on a different platform.
- **DTO First**. I made the mistake of spreading database access code through the code base. Just make a data transfer object and centralize the code in one place from the start. Then it can be easily packaged and reused.
- **One thing at a time**. The smart thing to do would have to been make one simple bot to test out the concept. I knew this at the time, but I really wanted to see what it would be like to develop a bot across all three

platforms. The person who made that decision was an idiot.

- **KISS**. So easy to say, so hard to do.
- **Plan ahead for Node and Browser module sharing**. It's hard to retrofit once you have a lot of code already written.
- **Code sharing**. Sharing nicely modularized library code between Lambda functions worked out great in practice. It made it easy to test from the command line.
- **Serverless works**. For all the reasons everyone gushes about. It lets one person get a lot done. You have devops, but you have much less to manage. For unpredictable workloads it's much cheaper and more scalable. I can't say the tooling suck because I chose not to use any, but we still have a ways to go to figure out how all this works in practice.
- **Events > API**. Using webhooks to connect systems together using Serverless is very powerful. By comparison standing up a box and running a process to use an API seems primitive.

Like most lessons most of these are depressingly head slappingly obvious in retrospect, but I guess that's why they're lessons.