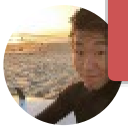


twitter.com/jonathanzwhite
Apr 19, 2016 · 6 min read

Server-side architecture when bots invade

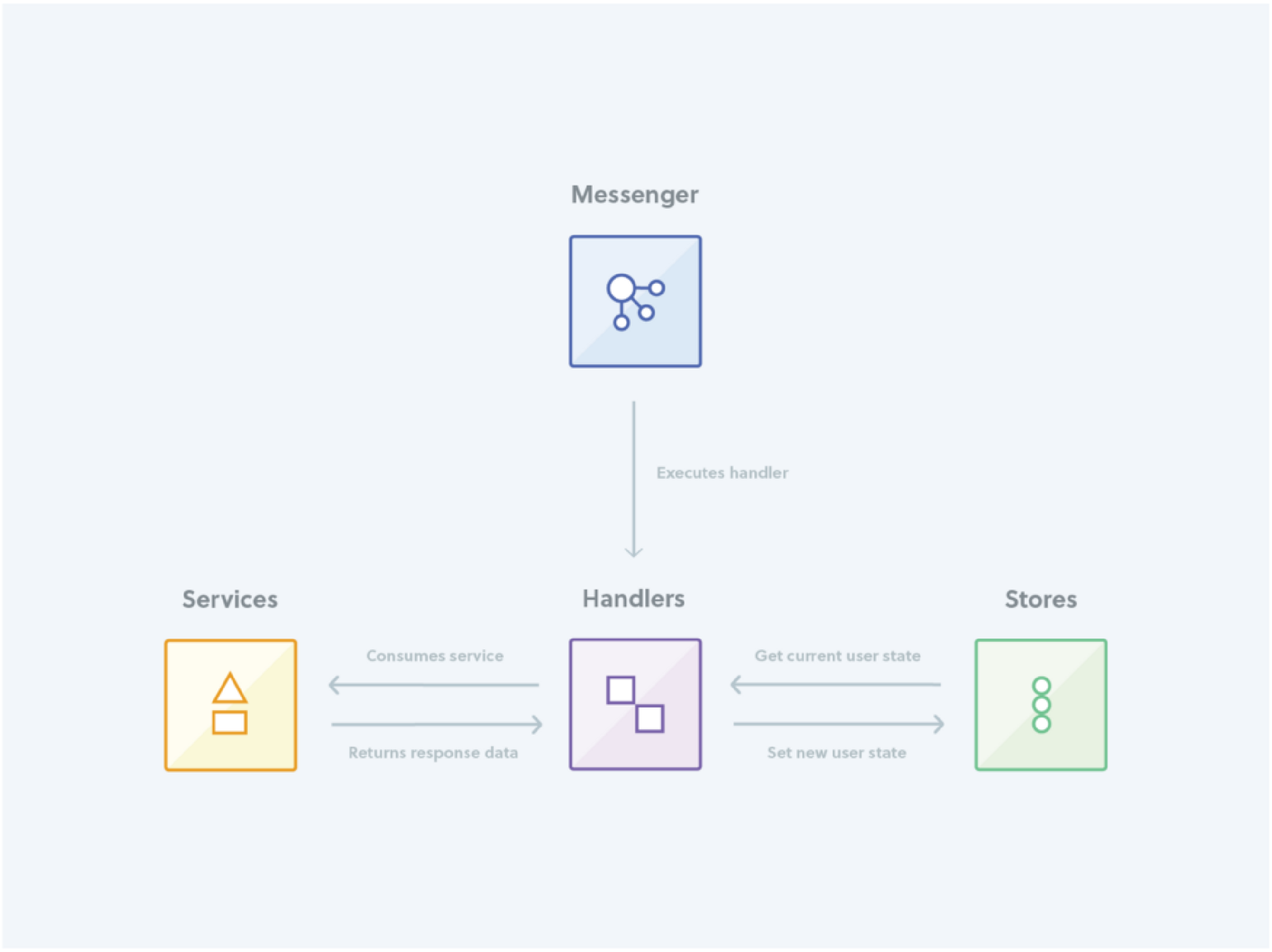
Since more developers are building bots, I decided to share a simple server-side structure for supporting conversational interactions. I've been using this approach for the bots I've been building in my spare time.

The motivation behind this system is to take traditional Node API architecture and adapt it for bots. I had a few goals in mind when designing this system. The system had to…

- Be easy to understand and extend
- Manage stateful chat exchanges
- Maintain a clear separation of concerns between application layers
- Be flexible enough to adapt to different bot APIs

Overview

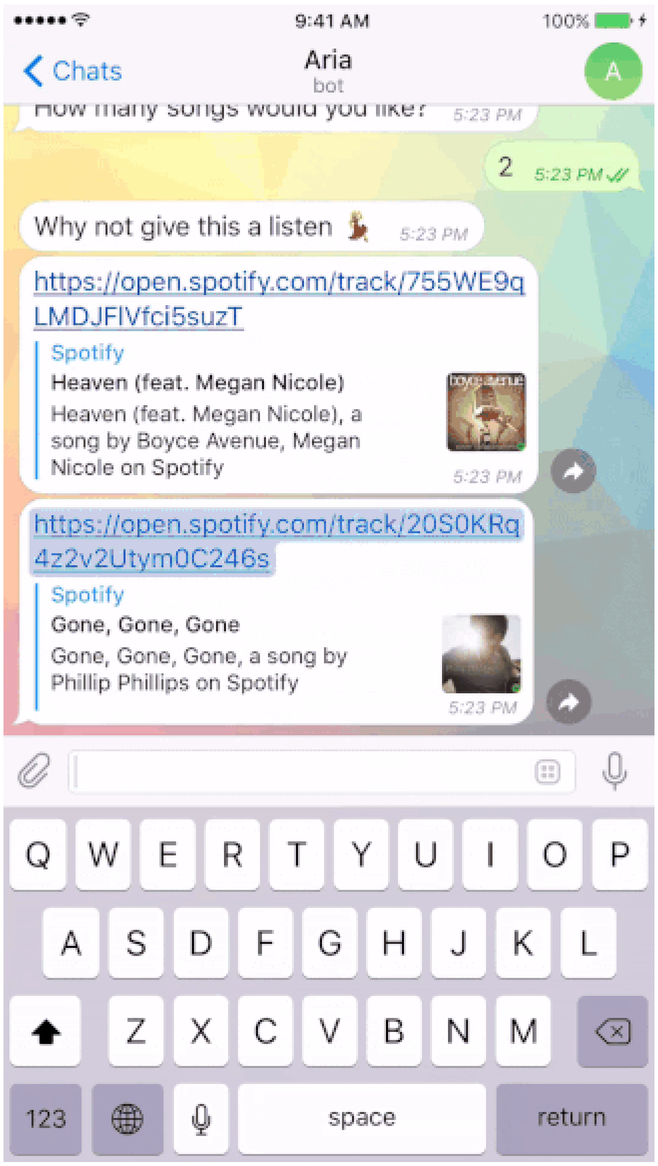
There are four main components in this system. The **Messenger** wraps bot APIs and provides a universal interface for receiving and handling messages. The Messenger figures out what the user wants and routes it to the right **handler**. You can think of handlers as the controllers of our system. The handlers delegate tasks to services**. Services** are where all the business logic for our system goes. When you need to store state—that is, information from chat exchanges—we interact with **stores**.



. . .

Break it down

Let’s break the system down into its four primary components. I’ll be using a Spotify music recommendation bot I built as an example. First, here is the final result.



How many songs would you like? 5:23 PM

2 5:23 PM

Why not give this a listen 5:23 PM

https://open.spotify.com/track/755WE9qLMDJFlVfci5suzT
Spotify
Heaven (feat. Megan Nicole)
Heaven (feat. Megan Nicole), a song by Boyce Avenue, Megan Nicole on Spotify
5:23 PM

https://open.spotify.com/track/20S0KRq4z2v2Utym0C246s
Spotify
Gone, Gone, Gone
Gone, Gone, Gone, a song by Phillip Phillips on Spotify
5:23 PM

Messenger

Messenger is where you receive messages and route them to an appropriate handler. The point of Messenger is provide a layer of abstraction on top of bot APIs. This way, if you decide to support more messaging services or switch away from one, the only code you have to modify is inside Messenger.

```
class Messenger {
  constructor() {
    this.bot = new TelegramBot('Your token here', { po
  }

  listen() {
    this.bot.on('text', this.handleText.bind(this))
  }
```

In the constructor of Messenger, we instantiate an instance of a pre-written bot API library. In this case, I'm using Telegram's API but you can easily replace it. Next we have the function *listen,* which is where we receive incoming chat hooks and pass messages to the *handleText* function.

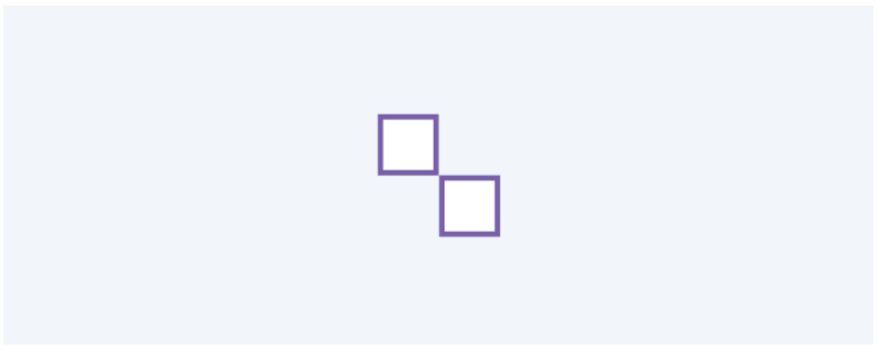
```
class Messenger {
  ...

  handleText(msg) {
    const message = new Message(Message.mapMessage(msg
    const text = message.text

    if (inputParser.isAskingForGenreList(text))
      return handlers.music.getGenreList(message, this

    if (inputParser.isAskingForNumberOfRec(text, store
      return handlers.music.getNumOfRec(message, this.

    if (inputParser.isAskingForRecommendation(text, st
```

handleText plays a similar role to that of a router in a traditional Node API. Inside *handleText* we pass the message and the latest state in our store to *inputParser* which figures out what the user wants and helps Messenger route to the right handler.

```
class InputParser {
  isAskingForGenreList(text) {
    const pattern = /music|recommendation/i

    return text.match(pattern)
  }

  isAskingForNumberOfRec(text, prevCommand) {
    return prevCommand === commands.GET_GENRE_LIST
  }
```

As you can see, *inputParser* uses either simple regex or looks at the state to provide an answer to Messenger. *inputParser* is also where you can add integrations with natural language processing platforms like Facebook's Wit.ai or API.ai.

Handlers



The role of the handler is to handle the message passed from Messenger, delegate tasks to services, update the store's state if necessary, and send messages back to the user.

```
class Music {
  getGenreList(message, bot) {
    // clears store for new command tree
    store.clearState(message.from)
    store.update(message.from, { command: commands.GET

    bot.sendMessage(message.from, 'Choose a genre you
      reply_markup: {
        keyboard: genresJSON.genres,
        one_time_keyboard: true
      }
```

This is a snippet from a music handler which is in charge of all actions and commands related to getting the user Spotify recommendations. In this example, *getGenreList* takes in a message and a bot passed in from Messenger. Message is an object that contains the ID of the user and the text they sent. Bot is how we send messages to the user.

Handlers typically update the store. In this example, we notify the store of where we are in the conversation tree. After updating the store, *getGenreList* sends the user a list of genres on Spotify to choose from.

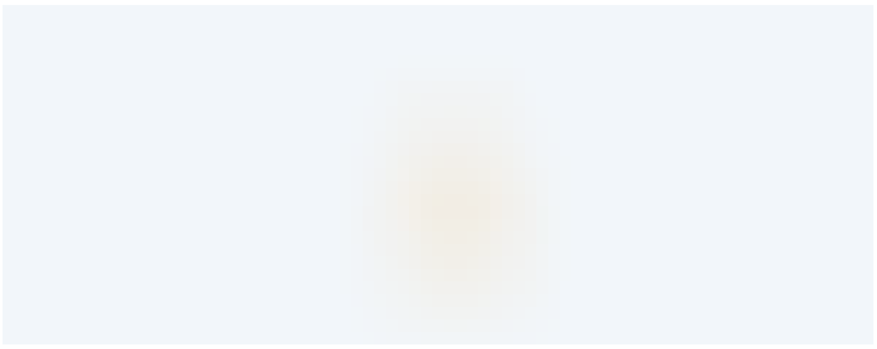
There are times when we need more complex business logic in order to figure out what to send back to the user. In these situations, we delegate the processing to services.

```
class Music {
  ...

  getRecommendation(message, bot) {
    store.update(message.from, {
      command: commands.GET_RECOMMENDATION,
      spotify: {
        numberOfRecs: message.text
      }
    })

    const state = store.getState(message.from)
    const selectedGenre = state.spotify.genre
    const numberOfRecs = state.spotify.numberOfRecs

    return MusicService.getRecommendation(selectedGenr
      .then(sendRecommendation)

    function sendRecommendation(resp) {
      if (!resp.length) return bot.sendMessage(message

      bot.sendMessage(message.from, 'Why not give this
```

Inside *getRecommendations,* just like in *getGenreList*, we first update the store. In this case we also update another part of the stored state so that our system remembers how many music recommendations the user wants.

Since we need to make an API call to get Spotify recommendations, we call the function *getRecommendation* from the music service. It returns a list of song recommendations that we can send to the user.

Services



The music service mentioned earlier is an example of what services do. Instead of polluting our handlers with logic, we let services do the heavy lifting. Services are in charge of making API requests, parsing data, and formatting responses for our handlers. Drawing the line between the responsibilities of handlers and services makes the code easier to test.

```
(function(module) {

  function getRecommendation(selectedGenre, limit) {
    ...

    return Promise.resolve()
      .then(getSpotifyAccessToken)
      .then(getRecommendation)

    function getRecommendation() {
      return spotify.getRecommendations({
        seed_genres: selectedGenre,
        limit: limit
      })
      .then((resp) => {
```

In this code snippet, the music service makes a request to the Spotify API, formats the response, and passes it back to the handler.

Stores

Stores are the last part of the system and help manage state from conversational interactions. In chat exchanges, your server might need to recall things like what genre of music your user is asking for and the number of recommendations they want. That's where stores come in. Stores hold conversation data and keep track of what state the system is in.

Above is a state diagram that models all the states of our recommendation system. Each state has data associated with it that we collect from the user. For example, the *asking for number of recommendations* state has to know what genre the user chose. Likewise, the *asking for recommendations* state has to not only inherit the previous state's data but also additional data about the number of recommendations a user wants.

```
let store = {
  _users: {},

  // creates state tree
  _initializeUserState: function(hash) {
    this._users[hash] = {
      command: '',
      spotify: {
        genre: '',
        numberOfRecs: 0
      }
```

Inside our store, we have a *users* property. This attribute stores a hashtable of users where the key is the user ID we get from a message and the value is the user's current state.

We also have an *initializeUserState* function that is called upon the user's first interaction with the system. This is where the state data and initial state is defined.

```
let store = {
  ...

  update: function(hash, data) {
    if (!this._users[hash]) this._initializeUserState(

    console.log('  => PREVIOUS STATE')
    console.log(this.getState(hash))

    this._users[hash] = _.merge(this._users[hash], dat
```

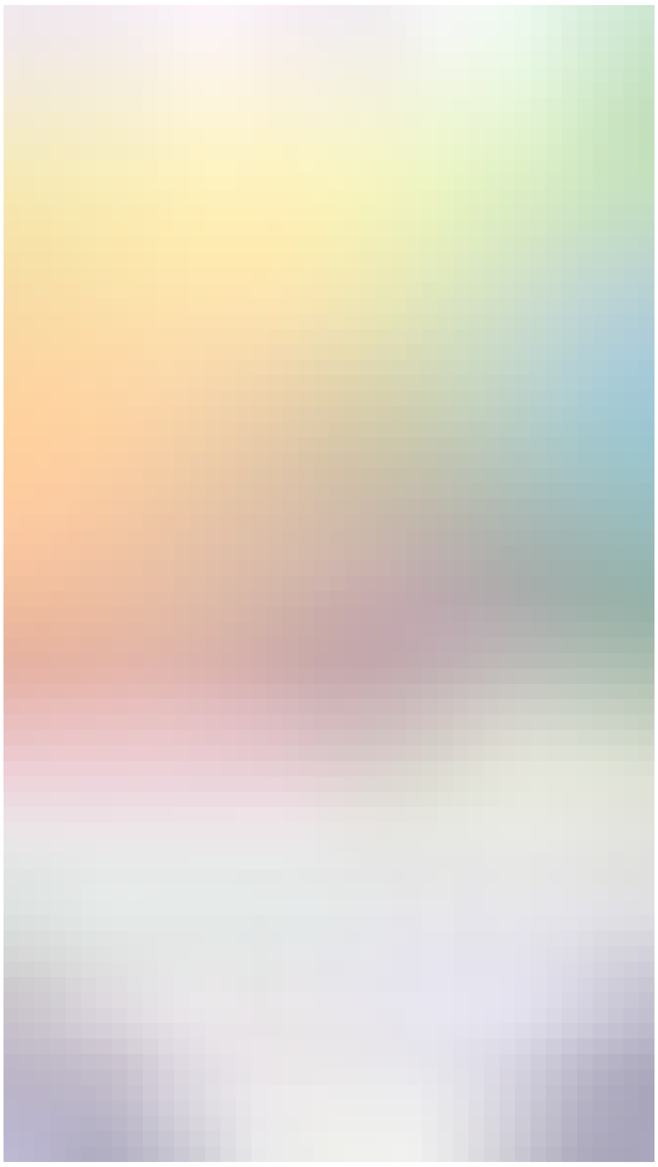
When the state needs to be updated, we call the *update* function. The *update* function takes the old state and deep merges it with the new state.

```
let store = {
  ...

  getState: function(hash) {
    if (!this._users[hash]) return this._initializeUse

    return this._users[hash]
  },

```

The last two functions inside of the store are *getState* and *clearState*. We use *getState* to get the user's current state and *clearState* for resetting the conversation state.

Note: Because our store is just a Javascript object, the state is not persistent. It is meant to hold only temporary state. If you want persistent data storage for storing things like user information or purchase history then you would use a database instead.

. . .



Conclusion

Although it’s far from perfect, hopefully this approach gives you inspiration for building your own systems. Feel free to leave a note or tweet to me to let me know how you are currently structuring your server-side code.

If you want to see the full code, you can find it here on Github.

P.S. If you liked this article, it would mean a lot if you hit the recommend button or share with friends.

. . .

If you want more, you can follow me on Twitter where I post non-sensical ramblings about design, front-end development, bots, and machine learning.