Alex Muramoto Follow
Jan 5 · 11 min read

Tutorial: Adding Live Chat via the Page Inbox with the Handover Protocol

One of the most powerful uses of the Messenger Platform's Handover Protocol is its ability to pass control of a conversation to the Page Inbox. Bots are awesome and can automatically do a lot of things for us, but there are many types of interactions that either require or are better handled by a human. This is where the handover protocol and Page Inbox come in.

Scope

In this tutorial, we will set up a simple webhook using Node. js and express that can do the following:

- Handle incoming messages, messaging_postback, standby, and messaging_handovers webhook events
- Send a welcome message with quick replies that allows the user to pass thread control to the Page Inbox
- Pass control to the Page Inbox
- Take thread control from the Page inbox
- Handle thread control being passed from the page inbox back to the bot manually when the live agent clicks the 'Done' button in the Page Inbox.

What you will need

Messenger Features

- **Webhook:** the Messenger Platform sends events to your webhook to notify your bot when a variety of interactions or events happen, including when a person sends a message.
- **Handover protocol:** the Messenger Platform's handover protocol enables two or more Facebook apps to participate in a

conversation by passing control of the conversation between them.

- **Quick replies:** quick replies provide a way to present a set of up to 11 buttons in-conversation that contain a title and optional image, and appear prominently above the composer.

Prerequisites

- **Facebook Page**: A Facebook Page will be used as the identity of your bot. When people chat with your app, they will see the Page name and the Page profile picture. Here is where you can create a page. Create a page here.
- **Facebook Developer Account**: Your developer account is required to create new apps, which are the core of any Facebook integration. To create a new developer account visit Facebook for Developers and click the 'Get Started' button.
- **Facebook App**: The Facebook app contains settings for your Messenger bot, including access tokens. To create a new app, visit your app dashboard.
- A server that has Node.js installed
- For reference, here are all set up instructions in case you need them.

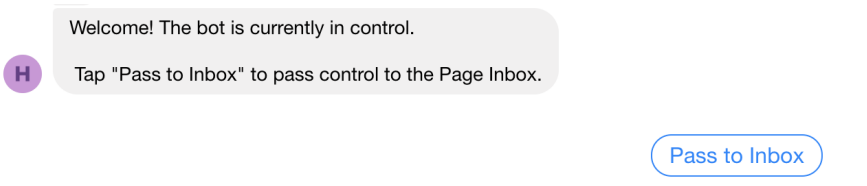
Repository with working code

All the code snippets used below can also be found here.

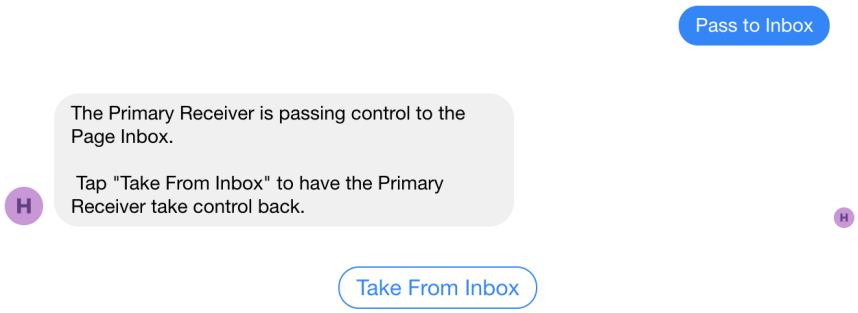
User Experience Overview

This is what the user experience will look like once you complete the tutorial:

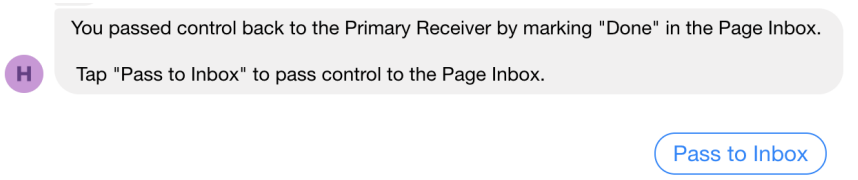
Welcome message



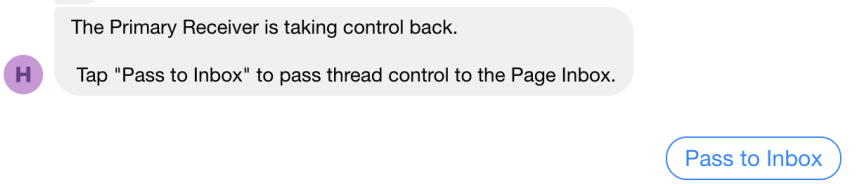
Users passes to inbox



User clicks done in page inbox



User clicks take from inbox quick reply



Step-by-step

STEP 1: Download the starter project

To make things a bit easier, we've created a starter project that has the basic project structure set up for you. You can download it from the tutorial-starters branch of our Github repo.

If you want to skip ahead and see the completed project, you can get that in the master branch.

STEP 2: Install dependencies

To begin, we'll need to install some Node modules. For this tutorial, we'll be using Express to set up a basic web server for our webhook, the body-parser module to parse the JSON body of the incoming POST requests, and the Request module to send HTTP requests to the Messenger Platform.

```
npm install
```

STEP 3. Setup your webhook

This is a basic webhook that listens on port 1337 by default, accepts GET requests for the webhook verification and accepts POST requests for webhook events. For a detailed walkthrough of building your own webhook from scratch, you can also follow our webhook set up guide in the Messenger Platform documentation.

```
// index.js //

'use strict';

// import dependencies
const bodyParser = require('body-parser'),
      express = require('express'),
      app = express();



// import helper libs that we will build next
const sendQuickReply = require('./utils/quick-reply'),
      HandoverProtocol = require('./utils/handover-
protocol'),
      env = require('./env');

// webhook setup
app.listen(process.env.PORT || env.PORT || 1337, () =>
console.log('webhook is listening'));
app.use(bodyParser.urlencoded({extended: false}));
app.use(bodyParser.json());


// handles webhook verification
app.get('/webhook', (req, res) => {
  if (req.query['hub.verify_token'] === env.VERIFY_TOKEN) {
    res.send(req.query['hub.challenge']);
  }
});
```

```
// receives webhook events from Messenger Platform
app.post('/webhook', (req, res) => {

  // parse messaging array
  const webhook_events = req.body.entry[0];



  // respond to all webhook events with 200 OK
  res.sendStatus(200);
});
```

STEP 4. Add code to listen when primary receiver is in control and when it is not

When the handover protocol is enabled, our bot will have the app role of Primary Receiver, which means it will receive all webhook events for our Page by default. Later, when we enable passing control of the conversation to the Page Inbox, we will also need it to listen on the standby channel, which will allow our bot to continue to monitor the conversation while it does not have control.

To do this, we update our webhook code to parse normal messaging events and standby events by detecting whether the incoming request has the standby or messaging property.

```
// receives webhook events from Messenger Platform
app.post('/webhook', (req, res) => {

  // parse messaging array
  const webhook_events = req.body.entry[0];

  // initialize quick reply properties
  let text, title, payload;

  // Secondary Receiver is in control - listen on standby
channel
  if (webhook_events.standby) {

    // iterate webhook events from standby channel
    webhook_events.standby.forEach(event => {

      const psid = event.sender.id;
      const message = event.message;


    });
  }
```

```
  // Bot is in control - listen for messages
  if (webhook_events.messaging) {

    // iterate webhook events
    webhook_events.messaging.forEach(event => {
      // parse sender PSID and message
      const psid = event.sender.id;
      const message = event.message;



    });
  }

  // respond to all webhook events with 200 OK
  res.sendStatus(200);

});
```

STEP 5. Create separate file to send messages

When building an app, it's often a good practice to create generic, reusable helper functions. In the code below, we use the request module to make send HTTP requests to the Messenger Platform, such as handover protocol actions, quick replies, and text messages.

```
// /utils/api.js //


'use strict';
const env = require('../env'),
      request = require('request');


// accepts the request info and makes the http request
function call (path, payload, callback) {
  const access_token = process.env.PAGE_ACCESS_TOKEN ||
env.PAGE_ACCESS_TOKEN;
  const graph_url = 'https://graph.facebook.com/me';


  if (!path) {
    console.error('No endpoint specified on Messenger
send!');
    return;
  } else if (!access_token || !graph_url) {
    console.error('No Page access token or graph API url
configured!');
    return;
  }


  request({
    uri: graph_url + path,
```

```
    qs: {'access_token': access_token},
    method: 'POST',
    json: payload,
  }, (error, response, body) => {
    console.log(body)
    if (!error && response.statusCode === 200) {
      console.log('Message sent succesfully');
    } else {
      console.error('Error: ' + error);
    }
    callback(body);
  });
};


// export the call function so it can be used elsewhere
module.exports = {
  call
};
```

STEP 6: Create separate file to send a quick reply

As part of this tutorial, we also want to be able to send quick replies that the user can tap to initiate pass_thread_control and take_thread_control handover protocol actions. This helper function simply formats our quick replies and then uses the call() function from the previous step to send.

```
// /utils/quick-reply.js //
'use strict';


//import API helper
const api = require('./api');


// Send a quick reply message
function sendQuickReply(psid, text, title, postback_payload)
{

  console.log('SENDING QUICK REPLY');

  let payload = {};

  payload.recipient = {
    id: psid
  }


  payload.message = {
    text: text,
    quick_replies: [{
        content_type: 'text',
        title: title,
        payload: postback_payload
```

```
      }]
    }


    api.call('/messages', payload, () => {});
}


// export the sendQuickReply function so it can be used
elsewhere
module.exports = sendQuickReply;
```

STEP 7: Add code to pass thread control

Next, we create helper functions for initiating handover protocol actions. The following code makes handover protocol pass_thread_control requests. The target_app_id will be the app ID we want to pass control of the conversation to.

```
// /utils/handover-protocol.js //


'use strict';


//import API helper
const api = require('./api');


function passThreadControl (userPsid, targetAppId) {
  console.log('PASSING THREAD CONTROL')
  let payload = {
    recipient: {
      id: userPsid
    },
    target_app_id: targetAppId
  };


  api.call('/pass_thread_control', payload, () => {});
}
```

STEP 8: Add code to take thread control

This is a helper function to make handover protocol take_thread_control requests. Note that unlike the pass thread control action, we don't specify an app ID to take thread control from. We only the PSID of the user because only the Primary Receiver can take thread control, so it only needs to specify which conversation it is taking control of.

```
function takeThreadControl (userPsid) {
  console.log('TAKING THREAD CONTROL')
  let payload = {
    recipient: {
      id: userPsid
    }
  };


  api.call('/take_thread_control', payload, () => {});
}


// export functions
module.exports = {
  passThreadControl,
  takeThreadControl
}
```

STEP 9: Add code to webhook to display welcome message with quick reply, and pass thread control when quick reply is clicked

From the screenshots above, you can see that the first thing our bot needs to do is send a welcome message, then detect when the user clicks the 'Pass to Inbox' quick reply. To enable this, we update the code in index.js that handles incoming message events with the following.

Notice that when control is passed to the Page Inbox, we are also sending a 'Take From Inbox' quick reply that will allow the user to return control of the conversation to the bot.

```
// Bot is in control - listen for messages
  if (webhook_events.messaging) {

    // iterate webhook events
    webhook_events.messaging.forEach(event => {
      // parse sender PSID and message
      const psid = event.sender.id;
      const message = event.message;
      if (message && message.quick_reply &&
message.quick_reply.payload == 'pass_to_inbox') {

        // quick reply to pass to Page inbox was clicked
        let page_inbox_app_id = 263902037430900;
        text = 'The Primary Receiver is passing control to
the Page Inbox. \n\n Tap "Take From Inbox" to have the
Primary Receiver take control back.';
        title = 'Take From Inbox';
        payload = 'take_from_inbox';
```

```
        sendQuickReply(psid, text, title, payload);
        HandoverProtocol.passThreadControl(psid,
page_inbox_app_id);

      } else {

        // default
        text = 'Welcome! The bot is currently in control.
\n\n Tap "Pass to Inbox" to pass control to the Page
Inbox.';
        title = 'Pass to Inbox';
        payload = 'pass_to_inbox';


        sendQuickReply(psid, text, title, payload);
      }
    });
  };
```

STEP 10: Add code to detect when thread control is passed manually back from page inbox to bot

Once the live agent has finished chatting with the user, they can click the 'Done' button in the Page Inbox UI to pass control of the conversation back to the bot. When this happens, the bot will receive a messaging_handovers webhook event that we can detect by checking for the pass_thread_control property. Here, we've updated the conditional in the last step with one more conditional.

```
if (message && message.quick_reply &&
message.quick_reply.payload == 'pass_to_inbox') {

        // quick reply to pass to Page inbox was clicked
        let page_inbox_app_id = 263902037430900;
        text = 'The Primary Receiver is passing control to
the Page Inbox. \n\n Tap "Take From Inbox" to have the
Primary Receiver take control back.';
        title = 'Take From Inbox';
        payload = 'take_from_inbox';

        sendQuickReply(psid, text, title, payload);
        HandoverProtocol.passThreadControl(psid,
page_inbox_app_id);

      } else if (event.pass_thread_control) {

        // thread control was passed back to bot manually in
Page inbox
        text = 'You passed control back to the Primary
Receiver by marking "Done" in the Page Inbox. \n\n Tap "Pass
to Inbox" to pass control to the Page Inbox.';
        title = 'Pass to Inbox';
```

```
        payload = 'pass_to_inbox';

        sendQuickReply(psid, text, title, payload);


      } else {

        // default
        text = 'Welcome! The bot is currently in control.
\n\n Tap "Pass to Inbox" to pass control to the Page
Inbox.';
        title = 'Pass to Inbox';
        payload = 'pass_to_inbox';


        sendQuickReply(psid, text, title, payload);
      }
```

STEP 11: Add code to listen on standby channel if the user clicks take thread control quick reply

Ok, last thing! We also want the user of this app to be able to return control to the bot by tapping the 'Take from Inbox' quick reply, which will trigger a take_thread_control handover protocol action. But we have a problem. When the quick reply is clicked, the Page Inbox will be receiving all messages sent to the Page.

To remedy this, we update the standby channel code in our webhook to listen for the postback payload from the quick reply.

```
// Secondary Receiver is in control - listen on standby
channel
  if (webhook_events.standby) {

    // iterate webhook events from standby channel
    webhook_events.standby.forEach(event => {

      const psid = event.sender.id;
      const message = event.message;


      if (message && message.quick_reply &&
message.quick_reply.payload == 'take_from_inbox') {
        // quick reply to take from Page inbox was clicked
        text = 'The Primary Receiver is taking control back.
\n\n Tap "Pass to Inbox" to pass thread control to the Page
Inbox.';
        title = 'Pass to Inbox';
        payload = 'pass_to_inbox';

        sendQuickReply(psid, text, title, payload);
        HandoverProtocol.takeThreadControl(psid);
      }
```

```
    });
  }
```

STEP 12: Deploy the code

You'll need to deploy this Node app to your favorite server. Any cloud-based option or your own hardware will do just fine, as long as it has Node.js installed and is configured to accept HTTPS requests.

STEP 13: Subscribe the app to receive webhook events for your Page

a. In the 'Token Generation' section of the Messenger settings console, click the 'Select a Page' dropdown and select the Facebook Page you want to subscribe this app to. This is the Page that you want your webhook to receive events for when people on Messenger chat with it.

b. Copy the token that appears in the 'Page Access Token' field. You will use this token later to make API requests.

c. In the 'Webhook' section of the Messenger settings console, click the 'Select a Page' dropdown and select the same Facebook Page you generated a Page access token for. This will subscribe your app to receive webhook events for the Page.

d. Click the 'Subscribe' button next to the dropdown.

The Messenger Platform will now be able to send the webhook events you subscribed to for the selected Page to your webhook.

STEP 14: Create env file with PAGE_ACCESS_TOKEN and VERIFY_TOKEN

Just to make things a bit easier, you may have noticed we wrote out webhook code to pull our page access token and verify token from a file called env. Relying on a file like this or environment variables is a good way to avoid accidentally uploading credentials to public repos on places like GitHub.

REMEMBER TO ADD THIS TO YOUR .GITIGNORE FILE OR AVOID

UPLOADING IT. Wouldn't want your page token to get leaked, would we?

```
module.exports = {
  "PAGE_ACCESS_TOKEN": "your page access token",
  "VERIFY_TOKEN": "your verify token"
}
```

STEP 15: Run it

```
node index.js
```

STEP 16: Send a message to your page to kick off the flow

Conclusion

You now have a working bot that can handle passing and taking thread control from the Page Inbox. But wait, there's more! You can use the handover protocol to pass and take thread control from any other bot that you control, which can be useful for many purposes, such as modularizing your bot's capabilities or creating a single Messenger experience for your Page that is composed on multiple bots on the backend, all working in coordination.