On This Page

Quick Start Tutorial

Get the Finished Code

This tutorial will guide you through everything you need to know to build your first Messenger bot. Before you begin, choose one of the options under Starter Projects to get the code you will need to start, then follow the steps under Getting Started to get set up.

Just want to run the finished code? Test Drive automates the bot creation process and gives you a complete code package that can be run locally on your computer.

Take Messenger Bots for a Test Drive

Contents

Starter Project

Before you begin this quick start, make sure you have completed one of the following to ensure you have the starter code you will need. The starter code provides a basic webhook that we will use as the foundation of our Messenger bot.

Option 1: Build It Yourself

Our webhook setup guide will walk you through building your first webhook that your can use with this quick start from start to finish.

Build Your Webhook

Option 2: Download It from GitHub

Download our webhook starter code from GitHub, and deploy it to a server of your choice.

Download the Code

Option 3: Remix It on Glitch

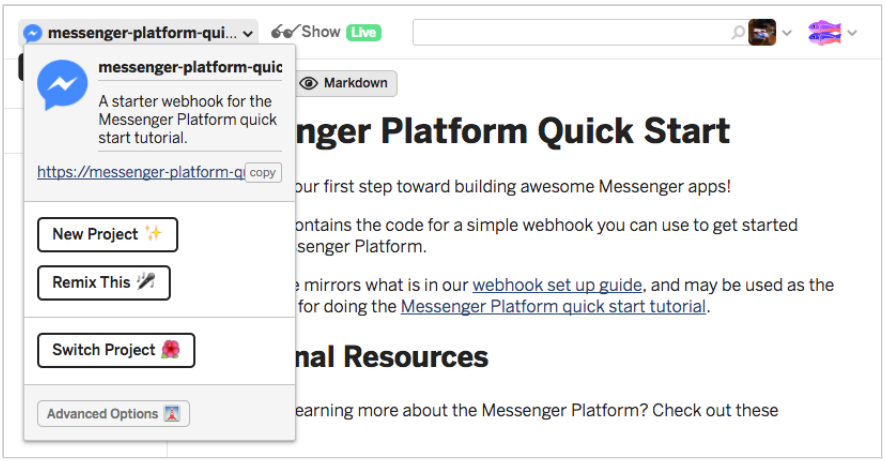
If you do not have a server to deploy your webhook to, you can remix our starter webhook project on Glitch, which will provide a public URL served over HTTPS for your webhook.

To create your own webhook on Glitch, do the following:

1. Open our starter webhook project on Glitch:

   Remix on Glitch

2. Click the 'Remix Your Own' button. A copy of the project will be created for you.
3. In the top-left corner, click the dropdown menu, then copy the public URL below the project description. This will be the base URL for your webhook.



4. Follow our app setup guide to subscribe your webhook to your Facebook app. Your webhook URL will be the Glitch URL appended with `/webhook`:

```
https://<GLITCH_URL>/webhook
```

Get Started

Before you build your first Messenger bot, start by setting up the credentials for your app.

1 Set Up Your Facebook App

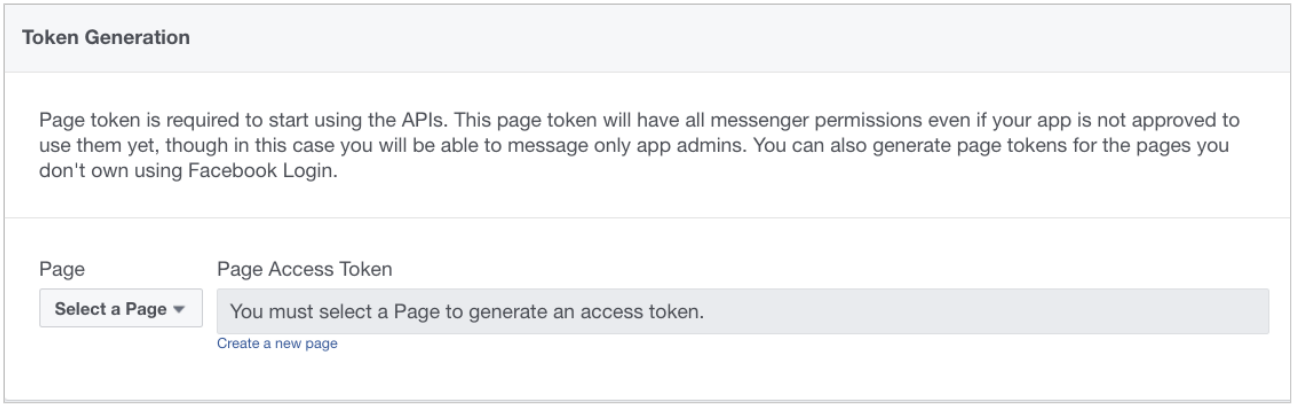
If you have not already, follow our app setup guide to set up your Facebook app for use with the Messenger Platform.

> **Your App is in Development Mode**
> Until your app has been submitted and approved for public use on Messenger, page tokens only allow your bot to interact with Facebook accounts that have been granted the Administrator, Developer, or Tester role for your app.
>
> To grant these roles to other Facebook accounts, go to the 'Roles' tab of your app settings.

2 Generate a page access token

All requests to Messenger Platform APIs are authenticated by including a page-level access token in the `access_token` parameter of the query string.

If you did not already do so when you set up your Facebook app, generate a page access token, by doing the following:



1. In the 'Token Generation' section of your app's Messenger settings, select the Facebook Page you want to generate a token for from the 'Page' dropdown. An access token will appear in the 'Page Access Token' field.
2. Click the 'Page Access Token' field to copy the token to your clipboard.

The generated token will NOT be saved in this UI. Each time you select a Page from the dropdown, a new token will be generated. If a new token is generated, previously created tokens will continue to function.

3 Save your page token as an environment variable

It is recommended that you keep sensitive information like your page access token secure by not hardcoding it into your webhook. To do this, add the following to your environment variables, where `<PAGE_ACCESS_TOKEN>` is the access token you just generated:

```
PAGE_ACCESS_TOKEN="<PAGE_ACCESS_TOKEN>"
```

> **Environment Variables in Glitch**
> If you are using Glitch, set your environment variables in the provided `.env` file to ensure they are not visible to other Glitch users.

4 Add your page and verify tokens to your webhook

Now all you have to do is add your page access token and verify token to your webhook code:

1. Add your page access token to the top of your `app.js` file:

```
const PAGE_ACCESS_TOKEN = process.env.PAGE_ACCESS_TOKEN;
```

2. Set your verify token in your webhook verification code:

```
app.get('/webhook', (req, res) => {

  const VERIFY_TOKEN = "<YOUR_VERIFY_TOKEN>";

  ...

})
```

🍪 🍪 🍪 🍪 🍪

Setup Complete

Let's build your first Messenger bot!

💥 💥 💥 💥 💥

Build the Bot

In this tutorial we will build a simple Messenger bot that does the following:

Parses the message and sender's page-scoped ID from an incoming webhook event.

Handles `messages` and `messaging_postbacks` webhook events.

Sends messages via the Send API.

Responds to text messages with a text message.

Responds to an image attachment with a generic template that uses the received image.

Responds conditionally to a postback payload.

1 Stub out handler functions

To start, we will stub out three functions that will handle the incoming webhook event types we want to support, as well as responding via the send API. To do this, append the following to your `app.js` file:

```
// Handles messages events
function handleMessage(sender_psid, received_message) {

}

// Handles messaging_postbacks events
function handlePostback(sender_psid, received_postback) {

}

// Sends response messages via the Send API
function callSendAPI(sender_psid, response) {
```

```
}
```

2 Get the sender's page-scoped ID

To respond to people on Messenger, the first thing we need is to know who they are. In Messenger bot's this is accomplished by getting the message sender's page-scoped ID (PSID) from the incoming webhook event.

> 💡 **What is a PSID?**
>
> A person is assigned a unique page-scoped ID (PSID) for each Facebook Page they start a conversation with. The PSID is used by your Messenger bot to identify a person when sending messages.

If you completed one of the options in the Requirements section above, you should have a basic `/webhook` endpoint that accepts `POST` requests and logs the body of received webhook events that looks like this:

```
app.post('/webhook', (req, res) => {

  // Parse the request body from the POST
  let body = req.body;

  // Check the webhook event is from a Page subscription
  if (body.object === 'page') {

    // Iterate over each entry - there may be multiple if batched
    body.entry.forEach(function(entry) {

      // Get the webhook event. entry.messaging is an array, but
      // will only ever contain one event, so we get index 0
      let webhook_event = entry.messaging[0];
      console.log(webhook_event);

    });

    // Return a '200 OK' response to all events
    res.status(200).send('EVENT_RECEIVED');
```

```
  } else {
    // Return a '404 Not Found' if event is not from a page subscription
    res.sendStatus(404);
  }

});
```

To get the sender's PSID, update the `body.entry.forEach` block with the following code to extract the PSID from the `sender.id` property of the event:

```
body.entry.forEach(function(entry) {

  // Gets the body of the webhook event
  let webhook_event = entry.messaging[0];
  console.log(webhook_event);

  // Get the sender PSID
  let sender_psid = webhook_event.sender.id;
  console.log('Sender PSID: ' + sender_psid);

});
```

Test It!

Open Messenger and send a message to the Facebook Page associated with your bot. You will not receive a response in Messenger, but you should see a message with the your PSID logged to the console where your webhook is running:

```
Sender PSID: 1254938275682919
```

3 Parse the webhook event type

We want our bot to be able to handle two types of webhook events: `messages` and `messaging_postback`. The name of the event type is not included in the event body, but we can determine it by checking for certain object properties.

What are webhook events?

The Messenger Platform sends webhook events to notify your Messenger bot of actions that occur in Messenger. Events are sent in JSON format as `POST` requests to your webhook. For more information, see Webhook Events.

To do this, update the `body.entry.forEach` block of your webhook with a conditional that checks whether the received event contains a `message` or `postback` property. We will also add calls to the `handleMessage()` and `handlePostback()` functions that we stubbed out earlier:

```
body.entry.forEach(function(entry) {

  // Gets the body of the webhook event
  let webhook_event = entry.messaging[0];
  console.log(webhook_event);


  // Get the sender PSID
  let sender_psid = webhook_event.sender.id;
  console.log('Sender PSID: ' + sender_psid);

  // Check if the event is a message or postback and
  // pass the event to the appropriate handler function
  if (webhook_event.message) {
    handleMessage(sender_psid, webhook_event.message);
  } else if (webhook_event.postback) {
    handlePostback(sender_psid, webhook_event.postback);
  }

});
```

4 Handle text messages

Now that our incoming messages are being routed to the appropriate handler function, we will update `handleMessage()` to handle and respond to basic text messages. To do this, update the code to define the message payload of our response, then pass that payload to `callSendAPI()`. We want to respond with a basic text message, so we define a JSON object with a `"text"` property:

```
function handleMessage(sender_psid, received_message) {

  let response;

  // Check if the message contains text
  if (received_message.text) {

    // Create the payload for a basic text message
    response = {
      "text": `You sent the message: "${received_message.text}". Now send m
    }
  }

  // Sends the response message
  callSendAPI(sender_psid, response);
}
```

5 Send a message with the Send API

Time to send your first message with the Messenger Platform's Send API!

In `handleMessage()`, we are calling `callSendAPI()` so now we need to update it to construct the full request body and send it to the Messenger Platform. A request to the Send API has two properties:

- `recipient`: Sets the intended message recipient. In this case, we identify the person by their PSID.
- `message`: Sets the details of the message to be sent. Here, we will set it to the message object we passed in from our `handleMessage()` function.

To construct the request body, update the stub for `callSendAPI()` to the following:

```
function callSendAPI(sender_psid, response) {
  // Construct the message body
  let request_body = {
    "recipient": {
      "id": sender_psid
    },
    "message": response
```

```
    }
}
```

Now all we have to do is send our message by submitting a `POST` request to the Send API at `https://graph.facebook.com/v2.6/me/messages`.

Note that you must append your `PAGE_ACCESS_TOKEN` in the `access_token` parameter of the URL query string.

Making HTTP Requests

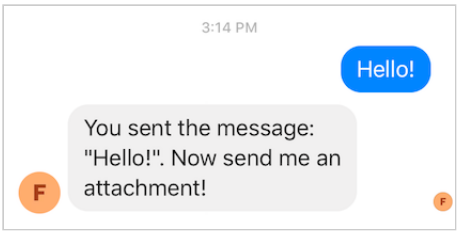
In this quick start, we are using the Node.js request module for sending HTTP requests back to the Messenger Platform, but you can use any HTTP client you like.

To install the request module, run `npm install request --save` from the command line, then import it by adding the following to the top of `app.js`:

```
const request = require('request');
```

```
function callSendAPI(sender_psid, response) {
  // Construct the message body
  let request_body = {
    "recipient": {
      "id": sender_psid
    },
    "message": response
  }

  // Send the HTTP request to the Messenger Platform
  request({
    "uri": "https://graph.facebook.com/v2.6/me/messages",
    "qs": { "access_token": PAGE_ACCESS_TOKEN },
    "method": "POST",
    "json": request_body
  }, (err, res, body) => {
    if (!err) {
      console.log('message sent!')
    } else {
      console.error("Unable to send message:" + err);
    }
  });
}
```



Test It!

In Messenger, send another text message to your Facebook Page. You should receive an automated response from your Messenger bot that echoes back your message and prompts you to send an image.

6 Handle attachments

Since our response prompts the message recipient to send an image, our next step is to update our code to handle an attachment. Sent attachments are automatically saved by the Messenger Platform and made available via a URL in the `payload.url` property of each index in the `attachments` array, so we will also extract this from the event.

What attachment types are supported?

Your Messenger bot can send and receive most asset types, including images, audio, video, and files. Media is displayed and is even playable in the conversation, allowing you to create media-rich experiences.

To determine if the message is an attachment, update the conditional in your `handleMessage()` function to check the `received_message` for an `attachments` property, then extract the URL for it. In a real-world bot we would iterate the array to check for multiple attachments, but for the purpose of this quick start, we will just get the first attachment.

```
function handleMessage(sender_psid, received_message) {

  let response;

  // Checks if the message contains text
  if (received_message.text) {
```

```
    // Creates the payload for a basic text message, which
    // will be added to the body of our request to the Send API
    response = {
      "text": `You sent the message: "${received_message.text}". Now send m
    }

  } else if (received_message.attachments) {

    // Gets the URL of the message attachment
    let attachment_url = received_message.attachments[0].payload.url;

  }

  // Sends the response message
  callSendAPI(sender_psid, response);
}
```

7 Send a structured message

Next, we will respond to the image with a generic template message. The generic template is the most commonly used structured message type, and allows you to send an image, text, and buttons in one message.

> **Are other message templates available?**
>
> Yes! The Messenger Platform provides a set of useful message templates, each designed to support a different, common message structure, including lists, receipts, buttons, and more. For complete details, see Templates.

Message templates are defined in the `attachment` property of the message, which contains `type` and `payload` properties. The `payload` is where we set the details of our generic template in the following properties:

- `template_type`: Sets the type of template used for the message. We are using the generic template, so the value is 'generic'.
- `elements`: Sets the custom properties of our template. For the generic template we will specify a title, subtitle, image, and two postback buttons.

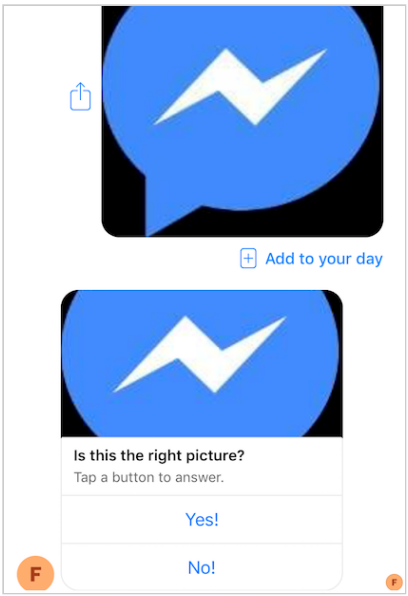
For our structured message, we will use the `attachment_url` that was sent to us as the `image_url` to display in our template, and include a couple postback buttons to allow the message recipient to respond. To construct the message payload and send the generic template, update `handleMessage()` to the following:

```
function handleMessage(sender_psid, received_message) {
  let response;

  // Checks if the message contains text
  if (received_message.text) {
    // Create the payload for a basic text message, which
    // will be added to the body of our request to the Send API
    response = {
      "text": `You sent the message: "${received_message.text}". Now send m
    }
  } else if (received_message.attachments) {
    // Get the URL of the message attachment
    let attachment_url = received_message.attachments[0].payload.url;
    response = {
      "attachment": {
        "type": "template",
        "payload": {
          "template_type": "generic",
          "elements": [{
            "title": "Is this the right picture?",
            "subtitle": "Tap a button to answer.",
            "image_url": attachment_url,
            "buttons": [
              {
                "type": "postback",
                "title": "Yes!",
                "payload": "yes",
              },
              {
                "type": "postback",
                "title": "No!",
                "payload": "no",
              }
            ],
          }]
        }
      }
    }
  }

  // Send the response message
```

```
  callSendAPI(sender_psid, response);
}
```



Test It!

In Messenger, send an image to your Facebook Page. Your Messenger bot should respond with a generic template.

8 Handle postbacks

Our last step is to handle the `messaging_postbacks` webhook event that will be sent when the message recipient taps one of the postback buttons in our generic template.

What can I do with postbacks?

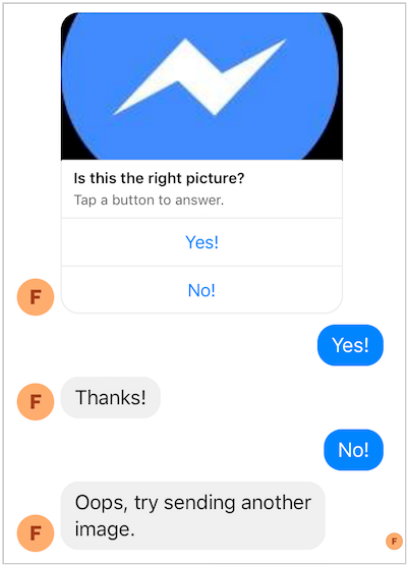
The postback button sends a `messaging_postbacks` webhook event to your webhook that includes a custom string of up to 1,000 character in the `payload` property. This allows you to easily implement different postback payloads that you can parse and respond to with specific behaviors.

Since our generic template allows the message recipient to choose from two postback buttons, we will respond based on the value of the `payload` property of the postback event. To do this, update your `handlePostback()` stub to the following:

```
function handlePostback(sender_psid, received_postback) {
  let response;

  // Get the payload for the postback
  let payload = received_postback.payload;

  // Set the response based on the postback payload
  if (payload === 'yes') {
    response = { "text": "Thanks!" }
  } else if (payload === 'no') {
    response = { "text": "Oops, try sending another image." }
  }
  // Send the message to acknowledge the postback
  callSendAPI(sender_psid, response);
}
```



Test It!

In Messenger, tap each of the postback buttons on the generic template. You should receive a different text response for each button.

9 😀 💥 🎉 💖 🙌 😻

If everything went well, you just finished building your first Messenger bot!