Building CI checks with a GitHub App

In this article

Build a continuous integration server to run tests using a GitHub App and checks.

Introduction

This tutorial demonstrates how to build a continuous integration (CI) server that runs tests on new code that's pushed to a repository. The tutorial shows how to build and configure a GitHub App to act as a server that receives and responds to `check_run` and `check_suite` webhook events using GitHub's REST API.

In this tutorial, you will use your computer or codespace as a server while you develop your app. Once the app is ready for production use, you should deploy your app to a dedicated server.

This tutorial uses Ruby, but you can use any programming language that you can run on your server.

This tutorial is broken into two parts:

- In part one, you'll learn how to set up the framework for a CI server using GitHub's REST API, create new check runs for CI tests when a repository receives newly pushed commits, and re-run check runs when a user requests that action on GitHub.
- In part two, you'll add functionality to your CI test, by adding a linter test to your CI server. You'll also create annotations that are displayed in the **Checks** and **Files Changed** tab of a pull request, and automatically fix linter recommendations by

exposing a "Fix this" button in the **Checks** tab of the pull request.

About continuous integration (CI)

CI is a software practice that requires frequently committing code to a shared repository. Committing code more often raises errors sooner and reduces the amount of code a developer needs to debug when finding the source of an error. Frequent code updates also make it easier to merge changes from different members of a software development team. This is great for developers, who can spend more time writing code and less time debugging errors or resolving merge conflicts.

A CI server hosts code that runs CI tests such as code linters (which check style formatting), security checks, code coverage, and other checks against new code commits in a repository. CI servers can even build and deploy code to staging or production servers. For examples of the types of CI tests you can create with a GitHub App, see the continuous integration apps that are available in GitHub Marketplace.

About checks

GitHub's REST API allows you to set up CI tests (checks) that are automatically run against each code commit in a repository. The API reports detailed information about each check in the pull request's **Checks** tab on GitHub. You can use checks in a repository to determine when a code commit introduces errors.

Checks include check runs, check suites, and commit statuses.

- A *check run* is an individual CI test that runs on a commit.
- A *check suite* is a group of check runs.
- A *commit status* marks the state of a commit, for example `error`, `failure`, `pending`, or `success`, and is visible in a pull request on GitHub. Both check suites and check runs contain commit statuses.

GitHub automatically creates `check_suite` events for new code commits in a repository using the default flow, although you can change the default settings. For more information, see "Check Suites." Here's how the default flow works:

1. When someone pushes code to the repository, GitHub automatically sends the `check_suite` event with an action of `requested` to all GitHub Apps installed on the repository that have the `checks:write` permission. This event lets the apps know that code was pushed to the repository, and that GitHub has automatically created a new check suite.
2. When your app receives this event, it can add check runs to that suite.
3. Your check runs can include annotations that are displayed on specific lines of code. Annotations are visible in the **Checks** tab. When you create an annotation for a file that is part of the pull request, the annotations are also shown in the **Files changed** tab. For more information, see the `annotations` object in the "Check Runs" documentation.

For more information about checks, see "Checks" and "Using the REST API to interact with checks."

Prerequisites

This tutorial assumes you have a basic understanding of the Ruby programming language.

Before you get started, you may want to familiarize yourself with the following concepts:

- GitHub Apps
- Webhooks
- REST API checks endpoints

Checks are also available to use with the GraphQL API, but this tutorial focuses on the REST API. For more information about the GraphQL objects, see Check Suite and Check Run in the GraphQL documentation.

Setup

The following sections will lead you through setting up the following components:

- A repository to store the code for your app.
- A way to receive webhooks locally.
- A GitHub App that is subscribed to "Check suite" and "Check run" webhook events, has write permission for checks, and uses a webhook URL that you can receive locally.

Create a repository to store code for your GitHub App

1. Create a repository to store the code for your app. For more information, see "Creating a new repository."
2. Clone your repository from the previous step. For more information, see "Cloning a repository." You may use a local clone or GitHub Codespaces.
3. In a terminal, navigate to the directory where your clone is stored.
4. Create a Ruby file named `server.rb`. This file will contain all the code for your app. You will add content to this file later.
5. If the directory doesn't already include a `.gitignore` file, add a `.gitignore` file. You will add content to this file later. For more information about `.gitignore` files, see "Ignoring files."
6. Create a file named `Gemfile`. This file will describe the gem dependencies that your Ruby code needs. Add the following contents to your `Gemfile`:

   Ruby

   ```ruby
   source 'http://rubygems.org'

   gem 'sinatra', '~> 2.0'
   gem 'jwt', '~> 2.1'
   gem 'octokit', '~> 4.0'
   gem 'puma'
   gem 'rubocop'
   gem 'dotenv'
   gem 'git'
   ```

7. Create a file named `config.ru`. This file will configure your Sinatra server to run. Add the following contents to your `config.ru` file:

   Ruby

   ```ruby
   require './server'
   run GHAapp
   ```

Get a webhook proxy URL

In order to develop your app locally, you can use a webhook proxy URL to forward webhook events from GitHub to your computer or codespace. This tutorial uses Smee.io to provide a webhook proxy URL and forward events.

1. In a terminal, run the following command to install the Smee client:

   ```shell
   npm install --global smee-client
   ```

2. In your browser, navigate to https://smee.io/.
3. Click **Start a new channel**.
4. Copy the full URL under "Webhook Proxy URL".
5. In the terminal, run the following command to start the Smee client. Replace `YOUR_DOMAIN` with the Webhook Proxy URL you copied in the previous step.

   ```shell
   smee --url YOUR_DOMAIN --path /event_handler --port 3000
   ```

   You should see output like the following:

   ```
   Forwarding https://smee.io/YOUR_DOMAIN to
   http://127.0.0.1:3000/event_handler
   Connected https://smee.io/YOUR_DOMAIN
   ```

The `smee --url https://smee.io/YOUR_DOMAIN` command tells Smee to forward all webhook events received by the Smee channel to the Smee client running on your computer. The `--path /event_handler` option forwards events to the `/event_handler` route. The `--port 3000` option specifies port 3000, which is the port you will tell your server to listen to, when you add more code later in the tutorial. Using Smee, your machine does not need to be open to the public internet to receive webhooks from GitHub. You can also open that Smee URL in your browser to inspect webhook payloads as they come in.

We recommend leaving this terminal window open and keeping Smee connected while you complete the rest of the steps in this guide. Although you can disconnect and reconnect the Smee client without losing your unique domain, you may find it easier to leave it connected and do other command-line tasks in a different terminal window.

Register a GitHub App

For this tutorial, you must register a GitHub App that:

- Has webhooks active
- Uses a webhook URL that you can receive locally
- Has the "Checks" repository permission
- Subscribes to the "Check suite" and "Check run" webhook events

The following steps will guide you through configuring a GitHub App with these settings. For more information about GitHub App settings, see "Registering a GitHub App."

1. In the upper-right corner of any page on GitHub, click your profile photo.
2. Navigate to your account settings.
   - For a GitHub App owned by a personal account, click **Settings**.
   - For a GitHub App owned by an organization:
     a. Click **Your organizations**.
     b. To the right of the organization, click **Settings**.
3. In the left sidebar, click <> **Developer settings**.
4. In the left sidebar, click **GitHub Apps**.
5. Click **New GitHub App**.
6. Under "GitHub App name", enter a name for your app. For example, `USERNAME-ci-test-app` where `USERNAME` is your GitHub username.
7. Under "Homepage URL", enter a URL for your app. For example, you can use the URL of the repository that you created to store the code for your app.
8. Skip the "Identifying and authorizing users" and "Post installation" sections for this tutorial.
9. Make sure that **Active** is selected under "Webhooks".
10. Under "Webhook URL", enter your webhook proxy URL from earlier. For more information, see "Get a webhook proxy URL."
11. Under "Webhook secret", enter a random string. This secret is used to verify that webhooks are sent by GitHub. Save this string; you will use it later.
12. Under "Repository permissions", next to "Checks", select **Read & write**.
13. Under "Subscribe to events", select **Check suite** and **Check run**.
14. Under "Where can this GitHub App be installed?", select **Only on this account**. You can change this later if you want to publish your app.
15. Click **Create GitHub App**.

Store your app's identifying information and credentials

This tutorial will show you how to store your app's credentials and identifying information as environment variables in a `.env` file. When you deploy your app, you should change how you store the credentials. For more information, see "Deploy your app."

Make sure that you are on a secure machine before performing these steps, since you will store your credentials locally.

1. In your terminal, navigate to the directory where your clone is stored.
2. Create a file called `.env` at the top level of this directory.
3. Add `.env` to your `.gitignore` file. This will prevent you from accidentally

committing your app's credentials.

4. Add the following contents to your `.env` file. You will update the values in a later step.

```shell
GITHUB_APP_IDENTIFIER="YOUR_APP_ID"
GITHUB_WEBHOOK_SECRET="YOUR_WEBHOOK_SECRET"
GITHUB_PRIVATE_KEY="YOUR_PRIVATE_KEY"
```

5. Navigate to the settings page for your app:

   a. In the upper-right corner of any page on GitHub, click your profile photo.

   b. Navigate to your account settings.

      - For a GitHub App owned by a personal account, click **Settings**.
      - For a GitHub App owned by an organization:

        a. Click **Your organizations**.

        b. To the right of the organization, click **Settings**.

   c. In the left sidebar, click **Developer settings**.

   d. In the left sidebar, click **GitHub Apps**.

   e. Next to your app's name, click **Edit**.

6. On your app's settings page, next to "App ID", find the app ID for your app.

7. In your `.env` file, replace `YOUR_APP_ID` with the app ID of your app.

8. In your `.env` file, replace `YOUR_WEBHOOK_SECRET` with the webhook secret for your app. If you have forgotten your webhook secret, under "Webhook secret (optional)", click **Change secret**. Enter a new secret, then click **Save changes**.

9. On your app's settings page, under "Private keys", click **Generate a private key**. You will see a private key `.pem` file downloaded to your computer.

10. Open the `.pem` file with a text editor, or use the following command on the command line to display the contents of the file: `cat PATH/TO/YOUR/private-key.pem`.

11. Copy and paste the entire contents of the file into your `.env` file as the value of `GITHUB_PRIVATE_KEY`, and add double quotes around the entire value.

    Here is an example .env file:

```
GITHUB_APP_IDENTIFIER=12345
GITHUB_WEBHOOK_SECRET=your webhook secret
GITHUB_PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
...
HkVN9...
...
-----END DSA PRIVATE KEY-----"
```

Add code for your GitHub App

This section will show you how to add some basic template code for your GitHub App, and it will explain what the code does. Later in the tutorial, you will learn how to modify and add to this code, to build out your app's functionality.

Add the following template code to your `server.rb` file:

Ruby

```ruby
require 'sinatra/base'  # Use the Sinatra web framework
require 'octokit'       # Use the Octokit Ruby library to interact with GitHub's
REST API
require 'dotenv/load'   # Manages environment variables
require 'json'          # Allows your app to manipulate JSON data
require 'openssl'       # Verifies the webhook signature
require 'jwt'           # Authenticates a GitHub App
require 'time'          # Gets ISO 8601 representation of a Time object
require 'logger'        # Logs debug statements

# This code is a Sinatra app, for two reasons:
#   1. Because the app will require a landing page for installation.
#   2. To easily handle webhook events.

class GHAapp < Sinatra::Application

  # Sets the port that's used when starting the web server.
  set :port, 3000
  set :bind, '0.0.0.0'

  # Expects the private key in PEM format. Converts the newlines.
  PRIVATE_KEY = OpenSSL::PKey::RSA.new(ENV['GITHUB_PRIVATE_KEY'].gsub('\n',
"\n"))

  # Your registered app must have a webhook secret.
  # The secret is used to verify that webhooks are sent by GitHub.
  WEBHOOK_SECRET = ENV['GITHUB_WEBHOOK_SECRET']

  # The GitHub App's identifier (type integer).
  APP_IDENTIFIER = ENV['GITHUB_APP_IDENTIFIER']

  # Turn on Sinatra's verbose logging during development
  configure :development do
    set :logging, Logger::DEBUG
  end

  # Executed before each request to the `/event_handler` route
  before '/event_handler' do
    get_payload_request(request)
    verify_webhook_signature

    # If a repository name is provided in the webhook, validate that
    # it consists only of latin alphabetic characters, `-`, and `_`.
    unless @payload['repository'].nil?
      halt 400 if (@payload['repository']['name'] =~ /[0-9A-Za-z\-\_]+/).nil?
    end

    authenticate_app
    # Authenticate the app installation in order to run API operations
    authenticate_installation(@payload)
  end

  post '/event_handler' do

    # ADD EVENT HANDLING HERE #

    200 # success status
```

```ruby
  end

  helpers do

    # ADD CREATE_CHECK_RUN HELPER METHOD HERE #

    # ADD INITIATE_CHECK_RUN HELPER METHOD HERE #

    # ADD CLONE_REPOSITORY HELPER METHOD HERE #

    # ADD TAKE_REQUESTED_ACTION HELPER METHOD HERE #

    # Saves the raw payload and converts the payload to JSON format
    def get_payload_request(request)
      # request.body is an IO or StringIO object
      # Rewind in case someone already read it
      request.body.rewind
      # The raw text of the body is required for webhook signature verification
      @payload_raw = request.body.read
      begin
        @payload = JSON.parse @payload_raw
      rescue => e
        fail  'Invalid JSON (#{e}): #{@payload_raw}'
      end
    end

    # Instantiate an Octokit client authenticated as a GitHub App.
    # GitHub App authentication requires that you construct a
    # JWT (https://jwt.io/introduction/) signed with the app's private key,
    # so GitHub can be sure that it came from the app an not altererd by
    # a malicious third party.
    def authenticate_app
      payload = {
          # The time that this JWT was issued, _i.e._ now.
          iat: Time.now.to_i,

          # JWT expiration time (10 minute maximum)
          exp: Time.now.to_i + (10 * 60),

          # Your GitHub App's identifier number
          iss: APP_IDENTIFIER
      }

      # Cryptographically sign the JWT.
      jwt = JWT.encode(payload, PRIVATE_KEY, 'RS256')

      # Create the Octokit client, using the JWT as the auth token.
      @app_client ||= Octokit::Client.new(bearer_token: jwt)
    end

    # Instantiate an Octokit client, authenticated as an installation of a
    # GitHub App, to run API operations.
    def authenticate_installation(payload)
      @installation_id = payload['installation']['id']
      @installation_token =
@app_client.create_app_installation_access_token(@installation_id)[:token]
      @installation_client = Octokit::Client.new(bearer_token:
@installation_token)
    end

    # Check X-Hub-Signature to confirm that this webhook was generated by
    # GitHub, and not a malicious third party.
    #
    # GitHub uses the WEBHOOK_SECRET, registered to the GitHub App, to
    # create the hash signature sent in the `X-HUB-Signature` header of each
    # webhook. This code computes the expected hash signature and compares it to
    # the signature sent in the `X-HUB-Signature` header. If they don't match,
    # this request is an attack, and you should reject it. GitHub uses the HMAC
    # hexdigest to compute the signature. The `X-HUB-Signature` looks something
    # like this: 'sha1=123456'.
    def verify_webhook_signature
```

```
      their_signature_header = request.env['HTTP_X_HUB_SIGNATURE'] || 'sha1='
      method, their_digest = their_signature_header.split('=')
      our_digest = OpenSSL::HMAC.hexdigest(method, WEBHOOK_SECRET, @payload_raw)
      halt 401 unless their_digest == our_digest

      # The X-GITHUB-EVENT header provides the name of the event.
      # The action value indicates the which action triggered the event.
      logger.debug "---- received event #{request.env['HTTP_X_GITHUB_EVENT']}"
      logger.debug "----    action #{@payload['action']}" unless
@payload['action'].nil?
    end

  end

  # Finally some logic to let us run this server directly from the command line,
  # or with Rack. Don't worry too much about this code. But, for the curious:
  # $0 is the executed file
  # __FILE__ is the current file
  # If they are the same—that is, we are running this file directly, call the
  # Sinatra run method
  run! if __FILE__ == $0
end
```

The rest of this section will explain what the template code does. There aren't any steps that you need to complete in this section. If you're already familiar with the template code, you can skip ahead to "Start the server."

Understand the template code

Open the `server.rb` file in a text editor. You'll see comments throughout the file that provide additional context for the template code. We recommend reading those comments carefully and even adding your own comments to accompany new code you write.

Below the list of required files, the first code you'll see is the `class GHApp < Sinatra::Application` declaration. You'll write all of the code for your GitHub App inside this class. The following sections explain in detail what the code does inside this class.

- Set the port
- Read the environment variables
- Turn on logging
- Define a `before` filter
- Define the route handler
- Define the helper methods

Set the port

The first thing you'll see inside the `class GHApp < Sinatra::Application` declaration is `set :port 3000`. This sets the port used when starting the web server, to match the port you redirected your webhook payloads to in "Get a Webhook Proxy URL."

```
  # Sets the port that's used when starting the web server.
  set :port, 3000
  set :bind, '0.0.0.0'
```

Read the environment variables

Next, this class reads the three environment variables you set in "Store your app's identifying information and credentials," and stores them in variables to use later.

```
# Expects the private key in PEM format. Converts the newlines.
PRIVATE_KEY = OpenSSL::PKey::RSA.new(ENV['GITHUB_PRIVATE_KEY'].gsub('\n', "\n"))

# Your registered app must have a webhook secret.
# The secret is used to verify that webhooks are sent by GitHub.
WEBHOOK_SECRET = ENV['GITHUB_WEBHOOK_SECRET']

# The GitHub App's identifier (type integer).
APP_IDENTIFIER = ENV['GITHUB_APP_IDENTIFIER']
```

Turn on logging

Next is a code block that enables logging during development, which is the default environment in Sinatra. This code turns on logging at the `DEBUG` level to show useful output in the terminal while you are developing the app.

```
# Turn on Sinatra's verbose logging during development
configure :development do
  set :logging, Logger::DEBUG
end
```

Define a `before` filter

Sinatra uses `before` filters that allow you to execute code before the route handler. The `before` block in the template calls four helper methods: `get_payload_request`, `verify_webhook_signature`, `authenticate_app`, and `authenticate_installation`. For more information, see "Filters" and "Helpers" in the Sinatra documentation.

```
  # Executed before each request to the `/event_handler` route
  before '/event_handler' do
    get_payload_request(request)
    verify_webhook_signature

    # If a repository name is provided in the webhook, validate that
    # it consists only of latin alphabetic characters, `-`, and `_`.
    unless @payload['repository'].nil?
      halt 400 if (@payload['repository']['name'] =~ /[0-9A-Za-z\-\_]+/).nil?
    end

    authenticate_app
    # Authenticate the app installation in order to run API operations
    authenticate_installation(@payload)
  end
```

Each of these helper methods are defined later in the code, in the code block that starts with `helpers do`. For more information, see "Define the helper methods."

Under `verify_webhook_signature`, the code that starts with `unless @payload` is a security measure. If a repository name is provided with a webhook payload, this code validates that the repository name contains only Latin alphabetic characters, hyphens, and underscores. This helps ensure that a bad actor isn't attempting to execute arbitrary commands or inject false repository names. Later, in the code block that starts with `helpers do`, the `verify_webhook_signature` helper method also validates incoming webhook payloads as an additional security measure.

Define a route handler

An empty route is included in the template code. This code handles all `POST` requests to the `/event_handler` route. You will add more code to this later.

```
post '/event_handler' do

end
```

Define the helper methods

Four helper methods are called in the `before` block of the template code. The `helpers do` code block defines each of these helper methods.

Handling the webhook payload

The first helper method `get_payload_request` captures the webhook payload and converts it to JSON format, which makes accessing the payload's data much easier.

Verifying the webhook signature

The second helper method `verify_webhook_signature` performs verification of the webhook signature to ensure that GitHub generated the event. To learn more about the code in the `verify_webhook_signature` helper method, see "Validating webhook deliveries." If the webhooks are secure, this method will log all incoming payloads to your terminal. The logger code is helpful in verifying your web server is working.

Authenticating as a GitHub App

The third helper method `authenticate_app` allows your GitHub App to authenticate, so it can request an installation token.

To make API calls, you'll be using the Octokit library. Doing anything interesting with this library will require your GitHub App to authenticate. For more information about the Octokit library, see the Octokit documentation.

GitHub Apps have three methods of authentication:

- Authenticating as a GitHub App using a JSON Web Token (JWT).
- Authenticating as a specific installation of a GitHub App using an installation access token.
- Authenticating on behalf of a user. This tutorial won't use this method of authentication.

You'll learn about authenticating as an installation in the next section, "Authenticating as an installation."

Authenticating as a GitHub App lets you do a couple of things:

- You can retrieve high-level management information about your GitHub App.
- You can request access tokens for an installation of the app.

For example, you would authenticate as a GitHub App to retrieve a list of the accounts (organization and personal) that have installed your app. But this authentication method doesn't allow you to do much with the API. To access a repository's data and perform operations on behalf of the installation, you need to authenticate as an installation. To do that, you'll need to authenticate as a GitHub App first to request an installation access token. For more information, see "About authentication with a GitHub App."

Before you can use the Octokit.rb library to make API calls, you'll need to initialize an Octokit client authenticated as a GitHub App, using the `authenticate_app` helper method.

```
# Instantiate an Octokit client authenticated as a GitHub App.
```

```ruby
# GitHub App authentication requires that you construct a
# JWT (https://jwt.io/introduction/) signed with the app's private key,
# so GitHub can be sure that it came from the app an not altered by
# a malicious third party.
def authenticate_app
  payload = {
      # The time that this JWT was issued, _i.e._ now.
      iat: Time.now.to_i,

      # JWT expiration time (10 minute maximum)
      exp: Time.now.to_i + (10 * 60),

      # Your GitHub App's identifier number
      iss: APP_IDENTIFIER
  }

  # Cryptographically sign the JWT
  jwt = JWT.encode(payload, PRIVATE_KEY, 'RS256')

  # Create the Octokit client, using the JWT as the auth token.
  @app_client ||= Octokit::Client.new(bearer_token: jwt)
end
```

The code above generates a JSON Web Token (JWT) and uses it (along with your app's private key) to initialize the Octokit client. GitHub checks a request's authentication by verifying the token with the app's stored public key. To learn more about how this code works, see "Generating a JSON Web Token (JWT) for a GitHub App."

Authenticating as an installation

The fourth and final helper method, `authenticate_installation`, initializes an Octokit client authenticated as an installation, which you can use to make authenticated calls to the API.

An *installation* refers to any user or organization account that has installed the app. Even if someone grants the app access to more than one repository on that account, it only counts as one installation because it's within the same account.

```ruby
# Instantiate an Octokit client authenticated as an installation of a
# GitHub App to run API operations.
def authenticate_installation(payload)
  installation_id = payload['installation']['id']
  installation_token =
@app_client.create_app_installation_access_token(installation_id)[:token]
  @installation_client = Octokit::Client.new(bearer_token: installation_token)
end
```

The `create_app_installation_access_token` Octokit method creates an installation token. For more information, see "create_installation_access_token" in the Octokit documentation.

This method accepts two arguments:

- Installation (integer): The ID of a GitHub App installation
- Options (hash, defaults to `{}`): A customizable set of options

Any time a GitHub App receives a webhook, it includes an `installation` object with an `id`. Using the client authenticated as a GitHub App, you pass this ID to the `create_app_installation_access_token` method to generate an access token for each installation. Since you're not passing any options to the method, the options default to an empty hash. The response for `create_app_installation_access_token` includes two fields: `token` and `expired_at`. The template code selects the token in the response and initializes an installation client.

With this method in place, each time your app receives a new webhook payload, it creates a client for the installation that triggered the event. This authentication process enables your GitHub App to work for all installations on any account.

Start the server

Your app doesn't do anything yet, but at this point, you can get it running on the server.

1. In your terminal, make sure that Smee is still running. For more information, see "Get a webhook proxy URL."
2. Open a new tab in your terminal, and `cd` into the directory where you cloned the repository that you created earlier in the tutorial. For more information, see "Create a repository to store code for your GitHub App." The Ruby code in this repository will start up a Sinatra web server.
3. Install the dependencies by running the following two commands one after the other:

   Shell
   ```
   gem install bundler
   ```

   Shell
   ```
   bundle install
   ```

4. After installing the dependencies, start the server by running this command:

   Shell
   ```
   bundle exec ruby server.rb
   ```

   You should see a response like this:

   ```
   > == Sinatra (v2.2.3) has taken the stage on 3000 for development with
   backup from Puma
   > Puma starting in single mode...
   > * Puma version: 6.3.0 (ruby 3.1.2-p20) ("Mugi No Toki Itaru")
   > *  Min threads: 0
   > *  Max threads: 5
   > *  Environment: development
   > *          PID: 14915
   > * Listening on http://0.0.0.0:3000
   > Use Ctrl-C to stop
   ```

   If you see an error, make sure you've created the `.env` file in the directory that contains `server.rb`.

5. To test the server, navigate in your browser to `http://localhost:3000`.

   If you see an error page that says "Sinatra doesn't know this ditty," the app is working as expected. Even though it's an error page, it's a Sinatra error page, which means your app is connected to the server as expected. You're seeing this message because you haven't given the app anything else to show.

Test that the server is listening to your app

You can test that the server is listening to your app by triggering an event for it to receive. You'll do that by installing the app on a test repository, which will send the `installation` event to your app. If the app receives it, you should see output in the terminal tab where you're running `server.rb`.

1. Create a new repository to use for testing your tutorial code. For more information, see "Creating a new repository."
2. Install the GitHub App on the repository you just created. For more information, see "Installing your own GitHub App." During the installation process, choose **Only select repositories**, and select the repository you created in the previous step.
3. After you click **Install**, look at the output in the terminal tab where you're running `server.rb`. You should see something like this:

   ```
   > D, [2023-06-08T15:45:43.773077 #30488] DEBUG -- : ---- received event
   installation
   > D, [2023-06-08T15:45:43.773141 #30488]] DEBUG -- : ----    action created
   > 192.30.252.44 - - [08/Jun/2023:15:45:43 -0400] "POST /event_handler
   HTTP/1.1" 200 - 0.5390
   ```

   If you see output like this, it means your app received a notification that it was installed on your GitHub account. The app is running on the server as expected.

   If you don't see this output, make sure Smee is running correctly in another terminal tab. If you need to restart Smee, note that you'll also need to *uninstall* and *reinstall* the app to send the `installation` event to your app again and see the output in terminal.

If you're wondering where the terminal output above is coming from, it's written in the app template code you added to `server.rb` in "Add code for your GitHub App."

Part 1. Creating the Checks API interface

In this part, you will add the code necessary to receive `check_suite` webhook events, and create and update check runs. You'll also learn how to create check runs when a check was re-requested on GitHub. At the end of this section, you'll be able to view the check run you created in a GitHub pull request.

Your check run will not perform any checks on the code in this section. You'll add that functionality in "Part 2: Creating the Octo RuboCop CI test."

You should already have a Smee channel configured that is forwarding webhook payloads to your local server. Your server should be running and connected to the GitHub App you registered and installed on a test repository.

These are the steps you'll complete in Part 1:

1. Add event handling
2. Create a check run
3. Update a check run

Step 1.1. Add event handling

Because your app is subscribed to the **Check suite** and **Check run** events, it will receive the `check_suite` and `check_run` webhooks. GitHub sends webhook payloads as `POST` requests. Because you forwarded your Smee webhook payloads to `http://localhost:3000/event_handler`, your server will receive the `POST` request payloads at the `post '/event_handler'` route.

Open the `server.rb` file that you created in "Add code for your GitHub App," and look for the following code. An empty `post '/event_handler'` route is already included in the template code. The empty route looks like this:

```
post '/event_handler' do

  # ADD EVENT HANDLING HERE #

  200 # success status
end
```

In the code block that starts with `post '/event_handler' do`, where it says `# ADD EVENT HANDLING HERE #`, add the following code. This route will handle the `check_suite` event.

Ruby

```
    # Get the event type from the HTTP_X_GITHUB_EVENT header
    case request.env['HTTP_X_GITHUB_EVENT']
    when 'check_suite'
      # A new check_suite has been created. Create a new check run with status
queued
      if @payload['action'] == 'requested' || @payload['action'] == 'rerequested'
        create_check_run
      end
      # ADD CHECK_RUN METHOD HERE #
    end
```

Every event that GitHub sends includes a request header called `HTTP_X_GITHUB_EVENT`, which indicates the type of event in the `POST` request. Right now, you're only interested in events of type `check_suite`, which are emitted when a new check suite is created. Each event has an additional `action` field that indicates the type of action that triggered the events. For `check_suite`, the `action` field can be `requested`, `rerequested`, or `completed`.

The `requested` action requests a check run each time code is pushed to the repository, while the `rerequested` action requests that you re-run a check for code that already exists in the repository. Because both the `requested` and `rerequested` actions require creating a check run, you'll call a helper called `create_check_run`. Let's write that method now.

Step 1.2. Create a check run

You'll add this new method as a Sinatra helper in case you want other routes to use it too.

In the code block that starts with `helpers do`, where it says `# ADD CREATE_CHECK_RUN HELPER METHOD HERE #`, add the following code:

Ruby

```
    # Create a new check run with status "queued"
    def create_check_run
      @installation_client.create_check_run(
        # [String, Integer, Hash, Octokit Repository object] A GitHub repository.
```

```
        @payload['repository']['full_name'],
        # [String] The name of your check run.
        'Octo RuboCop',
        # [String] The SHA of the commit to check
        # The payload structure differs depending on whether a check run or a
check suite event occurred.
        @payload['check_run'].nil? ? @payload['check_suite']['head_sha'] :
@payload['check_run']['head_sha'],
        # [Hash] 'Accept' header option, to avoid a warning about the API not
being ready for production use.
        accept: 'application/vnd.github+json'
      )
    end
```

This code calls the "Checks" endpoint using the Octokit create_check_run method.

To create a check run, only two input parameters are required: `name` and `head_sha`. In this code, we name the check run "Octo RuboCop," because we'll use RuboCop to implement the CI test later in the tutorial. But you can choose any name you'd like for the check run. For more information about RuboCop, see the RuboCop documentation.

You're only supplying the required parameters now to get the basic functionality working, but you'll update the check run later as you collect more information about the check run. By default, GitHub sets the `status` to `queued`.

GitHub creates a check run for a specific commit SHA, which is why `head_sha` is a required parameter. You can find the commit SHA in the webhook payload. Although you're only creating a check run for the `check_suite` event right now, it's good to know that the `head_sha` is included in both the `check_suite` and `check_run` objects in the event payloads.

The code above uses a ternary operator, which works like an `if/else` statement, to check if the payload contains a `check_run` object. If it does, you read the `head_sha` from the `check_run` object, otherwise you read it from the `check_suite` object.

Test the code

The following steps will show you how to test that the code works, and that it successfully creates a new check run.

1. Run the following command to restart the server from your terminal. If the server is already running, first enter `Ctrl-C` in your terminal to stop the server, and then run the following command to start the server again.

   Shell

   ```
   ruby server.rb
   ```

2. Create a pull request in the test repository you created in "Test that the server is listening to your app." This is the repository that you granted the app access to.

3. In the pull request you just created, navigate to the **Checks** tab. You should see a check run with the name "Octo RuboCop," or whichever name you chose earlier for the check run.

If you see other apps in the **Checks** tab, it means you have other apps installed on your repository that have **Read & write** access to checks and are subscribed to **Check suite** and **Check run** events. It may also mean that you have GitHub Actions workflows on the repository that are triggered by the `pull_request` or `pull_request_target` event.

So far you've told GitHub to create a check run. The check run status in the pull request is set to queued with a yellow icon. In the next step, you will wait for GitHub to create the check run and update its status.

Step 1.3. Update a check run

When your `create_check_run` method runs, it asks GitHub to create a new check run. When GitHub finishes creating the check run, you'll receive the `check_run` webhook event with the `created` action. That event is your signal to begin running the check.

You'll update your event handler to look for the `created` action. While you're updating the event handler, you can add a conditional for the `rerequested` action. When someone re-runs a single test on GitHub by clicking the "Re-run" button, GitHub sends the `rerequested` check run event to your app. When a check run is `rerequested`, you'll start the process all over and create a new check run. To do that, you'll include a condition for the `check_run` event in the `post '/event_handler'` route.

In the code block that starts with `post '/event_handler' do`, where it says `# ADD CHECK_RUN METHOD HERE #`, add the following code:

Ruby

```ruby
    when 'check_run'
      # Check that the event is being sent to this app
      if @payload['check_run']['app']['id'].to_s === APP_IDENTIFIER
        case @payload['action']
        when 'created'
          initiate_check_run
        when 'rerequested'
          create_check_run
        # ADD REQUESTED_ACTION METHOD HERE #
        end
      end
```

GitHub sends all events for `created` check runs to every app installed on a repository that has the necessary checks permissions. That means that your app will receive check runs created by other apps. A `created` check run is a little different from a `requested` or `rerequested` check suite, which GitHub sends only to apps that are being requested to run a check. The code above looks for the check run's application ID. This filters out all check runs for other apps on the repository.

Next you'll write the `initiate_check_run` method, which is where you'll update the check run status and prepare to kick off your CI test.

In this section, you're not going to kick off the CI test yet, but you'll walk through how to update the status of the check run from `queued` to `pending` and then from `pending` to `completed` to see the overall flow of a check run. In "Part 2: Creating the Octo RuboCop CI test," you'll add the code that actually performs the CI test.

Let's create the `initiate_check_run` method and update the status of the check run.

In the code block that starts with `helpers do`, where it says `# ADD INITIATE_CHECK_RUN HELPER METHOD HERE #`, add the following code:

Ruby

```ruby
    # Start the CI process
    def initiate_check_run
      # Once the check run is created, you'll update the status of the check run
      # to 'in_progress' and run the CI process. When the CI finishes, you'll
      # update the check run status to 'completed' and add the CI results.
```

```
    @installation_client.update_check_run(
      @payload['repository']['full_name'],
      @payload['check_run']['id'],
      status: 'in_progress',
      accept: 'application/vnd.github+json'
    )

    # ***** RUN A CI TEST *****

    # Mark the check run as complete!
    @installation_client.update_check_run(
      @payload['repository']['full_name'],
      @payload['check_run']['id'],
      status: 'completed',
      conclusion: 'success',
      accept: 'application/vnd.github+json'
    )

  end
```

The code above calls the "Update a check run" endpoint using the `update_check_run` Octokit method, and updates the check run that you already created.

Here's what this code is doing. First, it updates the check run's status to `in_progress` and implicitly sets the `started_at` time to the current time. In Part 2 of this tutorial, you'll add code that kicks off a real CI test under `***** RUN A CI TEST *****`. For now, you'll leave that section as a placeholder, so the code that follows it will just simulate that the CI process succeeds and all tests pass. Finally, the code updates the status of the check run again to `completed`.

When you use the REST API to provide a check run status of `completed`, the `conclusion` and `completed_at` parameters are required. The `conclusion` summarizes the outcome of a check run and can be `success`, `failure`, `neutral`, `cancelled`, `timed_out`, `skipped`, or `action_required`. You'll set the conclusion to `success`, the `completed_at` time to the current time, and the status to `completed`.

You could also provide more details about what your check is doing, but you'll get to that in the next section.

Test the code

The following steps will show you how to test that the code works, and that the new "Re-run all" button you created works.

1. Run the following command to restart the server from your terminal. If the server is already running, first enter `Ctrl-C` in your terminal to stop the server, and then run the following command to start the server again.

   Shell

   ```
   ruby server.rb
   ```

2. Create a pull request in the test repository you created in "Test that the server is listening to your app." This is the repository that you granted the app access to.
3. In the pull request you just created, navigate to the **Checks** tab. You should see a "Re-run all" button.
4. Click the "Re-run all" button in the upper right corner. The test should run again, and end with `success`.

Part 2. Creating a CI test

Now that you've got the interface created to receive API events and create check runs, you can create a check run that implements a CI test.

RuboCop is a Ruby code linter and formatter. It checks Ruby code to ensure that it complies with the Ruby Style Guide. For more information, see the RuboCop documentation.

RuboCop has three primary functions:

- Linting to check code style
- Code formatting
- Replaces the native Ruby linting capabilities using `ruby -w`

Your app will run RuboCop on the CI server, and create check runs (CI tests in this case) that report the results that RuboCop reports to GitHub.

The REST API allows you to report rich details about each check run, including statuses, images, summaries, annotations, and requested actions.

Annotations are information about specific lines of code in a repository. An annotation allows you to pinpoint and visualize the exact parts of the code you'd like to show additional information for. For example, you could show that information as a comment, error, or warning on a specific line of code. This tutorial uses annotations to visualize RuboCop errors.

To take advantage of requested actions, app developers can create buttons in the **Checks** tab of pull requests. When someone clicks one of these buttons, the click sends a `requested_action` `check_run` event to the GitHub App. The action that the app takes is completely configurable by the app developer. This tutorial will walk you through adding a button that allows users to request that RuboCop fix the errors it finds. RuboCop supports automatically fixing errors using a command-line option, and you'll configure the `requested_action` to take advantage of this option.

These are the steps you'll complete in this section:

1. Add a Ruby file
2. Clone the repository
3. Run RuboCop
4. Collect RuboCop errors
5. Update the check run with CI test results
6. Automatically fix RuboCop errors

Step 2.1. Add a Ruby file

You can pass specific files or entire directories for RuboCop to check. In this tutorial, you'll run RuboCop on an entire directory. RuboCop only checks Ruby code. To test your GitHub App, you'll need to add a Ruby file in your repository that contains errors for RuboCop to find. After adding the following Ruby file to your repository, you will update your CI check to run RuboCop on the code.

1. Navigate to the test repository you created in "Test that the server is listening to

your app." This is the repository that you granted the app access to.

2. Create a new file named `myfile.rb`. For more information, see "Creating new files."
3. Add the following content to `myfile.rb`:

```ruby
# frozen_string_literal: true

# The Octocat class tells you about different breeds of Octocat
class Octocat
  def initialize(name, *breeds)
    # Instance variables
    @name = name
    @breeds = breeds
  end

  def display
    breed = @breeds.join("-")

    puts "I am of #{breed} breed, and my name is #{@name}."
  end
end

m = Octocat.new("Mona", "cat", "octopus")
m.display
```

4. If you created the file locally, make sure you commit and push the file to your repository on GitHub.

Step 2.2. Allow RuboCop to clone the test repository

RuboCop is available as a command-line utility. That means, if you want to run RuboCop on a repository, your GitHub App will need to clone a local copy of the repository on the CI server so RuboCop can parse the files. To do that, your code will need to be able to run Git operations, and your GitHub App will need to have the correct permissions to clone a repository.

Allow Git operations

To run Git operations in your Ruby app, you can use the ruby-git gem. The `Gemfile` you created in "Setup" already includes the ruby-git gem, and you installed it when you ran `bundle install` in "Start the server."

Now, at the top of your `server.rb` file, below the other `require` items, add the following code:

```ruby
require 'git'
```

Update your app permissions

Next you'll need to update your GitHub App's permissions. Your app will need read permission for "Contents" to clone a repository. And later in this tutorial, it will need write

permission to push contents to GitHub. To update your app's permissions:

1. Select your app from the app settings page, and click **Permissions & events** in the sidebar.
2. Under "Repository permissions", next to "Contents", select **Read & write**.
3. Click **Save changes** at the bottom of the page.
4. If you've installed the app on your account, check your email and follow the link to accept the new permissions. Any time you change your app's permissions or webhooks, users who have installed the app (including yourself) will need to accept the new permissions before the changes take effect. You can also accept the new permissions by navigating to your installations page. You'll see a link under the app name, letting you know that the app is requesting different permissions. Click "Review request", and then click "Accept new permissions."

Add code to clone a repository

To clone a repository, the code will use your GitHub App's permissions and the Octokit SDK to create an installation token for your app (`x-access-token:TOKEN`) and use it in the following clone command:

```
git clone https://x-access-token:TOKEN@github.com/OWNER/REPO.git
```

The command above clones a repository over HTTP. It requires the full repository name, which includes the repository owner (user or organization) and the repository name. For example, the octocat Hello-World repository has a full name of `octocat/hello-world` .

Open your `server.rb` file. In the code block that starts with `helpers do` , where it says `# ADD CLONE_REPOSITORY HELPER METHOD HERE #` , add the following code:

Ruby

```ruby
    # Clones the repository to the current working directory, updates the
    # contents using Git pull, and checks out the ref.
    #
    # full_repo_name  - The owner and repo. Ex: octocat/hello-world
    # repository      - The repository name
    # ref             - The branch, commit SHA, or tag to check out
    def clone_repository(full_repo_name, repository, ref)
      @git = Git.clone("https://x-access-token:#
{@installation_token.to_s}@github.com/#{full_repo_name}.git", repository)
      pwd = Dir.getwd()
      Dir.chdir(repository)
      @git.pull
      @git.checkout(ref)
      Dir.chdir(pwd)
    end
```

The code above uses the `ruby-git` gem to clone the repository using the app's installation token. It clones the code in the same directory as `server.rb` . To run Git commands in the repository, the code needs to change into the repository directory. Before changing directories, the code stores the current working directory in a variable (`pwd`) to remember where to return before exiting the `clone_repository` method.

From the repository directory, this code fetches and merges the latest changes (`@git.pull`), and checks out the specifig Git ref (`@git.checkout(ref)`). The code to do all of this fits nicely into its own method. To perform these operations, the method needs the name and full name of the repository and the ref to checkout. The ref can be a

commit SHA, branch, or tag. When it's done, the code changes the directory back to the original working directory (`pwd`).

Now you've got a method that clones a repository and checks out a ref. Next, you need to add code to get the required input parameters and call the new `clone_repository` method.

In the code block that starts with `helpers do` , in the `initiate_check_run` helper method where it says `# ***** RUN A CI TEST *****` , add the following code:

Ruby

```ruby
    full_repo_name = @payload['repository']['full_name']
    repository     = @payload['repository']['name']
    head_sha       = @payload['check_run']['head_sha']

    clone_repository(full_repo_name, repository, head_sha)

    # ADD CODE HERE TO RUN RUBOCOP #
```

The code above gets the full repository name and the head SHA of the commit from the `check_run` webhook payload.

Step 2.3. Run RuboCop

So far, your code clones the repository and creates check runs using your CI server. Now you'll get into the details of the RuboCop linter and checks annotations.

First, you'll add code to run RuboCop and save the style code errors in JSON format.

In the code block that starts with `helpers do` , find the `initiate_check_run` helper method. Inside that helper method, under `clone_repository(full_repo_name, repository, head_sha)` , where it says `# ADD CODE HERE TO RUN RUBOCOP #` , add the following code:

Ruby

```ruby
        # Run RuboCop on all files in the repository
        @report = `rubocop '#{repository}' --format json`
        logger.debug @report
        `rm -rf #{repository}`
        @output = JSON.parse @report

        # ADD ANNOTATIONS CODE HERE #
```

The code above runs RuboCop on all files in the repository's directory. The option `--format json` saves a copy of the linting results in a machine-parsable format. For more information, and an example of the JSON format, see "JSON Formatter" in the RuboCop docs. This code also parses the JSON so you can easily access the keys and values in your GitHub App using the `@output` variable.

After running RuboCop and saving the linting results, this code runs the command `rm -rf` to remove the checkout of the repository. Because the code stores the RuboCop results in a `@report` variable, it can safely remove the checkout of the repository.

The `rm -rf` command cannot be undone. To keep your app secure, the code in this tutorial checks incoming webhooks for injected malicious commands that could be used to remove a different directory than intended by your app. For example, if a bad actor sent a webhook with the repository name `./` , your app would remove the root directory. The `verify_webhook_signature` method validates the sender of the webhook. The `verify_webhook_signature` event handler also checks that the repository name is valid.

For more information, see "Define a `before` filter."

Test the code

The following steps will show you how to test that the code works and view the errors reported by RuboCop.

1. Run the following command to restart the server from your terminal. If the server is already running, first enter `Ctrl-C` in your terminal to stop the server, and then run the following command to start the server again.

   Shell

   ```
   ruby server.rb
   ```

2. In the repository where you added the `myfile.rb` file, create a new pull request.

3. In your terminal tab where the server is running, you should see debug output that contains linting errors. The linting errors are printed without any formatting. You can copy and paste your debug output into a web tool like JSON formatter, to format your JSON output like the following example:


```
{
  "metadata": {
    "rubocop_version": "0.60.0",
    "ruby_engine": "ruby",
    "ruby_version": "2.3.7",
    "ruby_patchlevel": "456",
    "ruby_platform": "universal.x86_64-darwin18"
  },
  "files": [
    {
      "path": "Octocat-breeds/octocat.rb",
      "offenses": [
        {
          "severity": "convention",
          "message": "Style/StringLiterals: Prefer single-quoted strings
when you don't need string interpolation or special symbols.",
          "cop_name": "Style/StringLiterals",
          "corrected": false,
          "location": {
            "start_line": 17,
            "start_column": 17,
            "last_line": 17,
            "last_column": 22,
            "length": 6,
            "line": 17,
            "column": 17
          }
        },
        {
          "severity": "convention",
          "message": "Style/StringLiterals: Prefer single-quoted strings
when you don't need string interpolation or special symbols.",
          "cop_name": "Style/StringLiterals",
          "corrected": false,
          "location": {
            "start_line": 17,
            "start_column": 25,
            "last_line": 17,
            "last_column": 29,
            "length": 5,
            "line": 17,
            "column": 25
```

```
          }
        }
      ]
    }
  ],
  "summary": {
    "offense_count": 2,
    "target_file_count": 1,
    "inspected_file_count": 1
  }
}
```

Step 2.4. Collect RuboCop errors

The `@output` variable contains the parsed JSON results of the RuboCop report. As shown in the example output in the previous step, the results contain a `summary` section that your code can use to quickly determine if there are any errors. The following code will set the check run conclusion to `success` when there are no reported errors. RuboCop reports errors for each file in the `files` array, so if there are errors, you'll need to extract some data from the file object.

The REST API endpoints to manage check runs allow you to create annotations for specific lines of code. When you create or update a check run, you can add annotations. In this tutorial you will update the check run with annotations, using the "Update a check run" endpoint.

The API limits the number of annotations to a maximum of 50 per request. To create more than 50 annotations, you will have to make multiple requests to the "Update a check run" endpoint. For example, to create 105 annotations you would need to make three separate requests to the API. The first two requests would each have 50 annotations, and the third request would include the five remaining annotations. Each time you update the check run, annotations are appended to the list of annotations that already exist for the check run.

A check run expects annotations as an array of objects. Each annotation object must include the `path`, `start_line`, `end_line`, `annotation_level`, and `message`. RuboCop provides the `start_column` and `end_column` too, so you can include those optional parameters in the annotation. Annotations only support `start_column` and `end_column` on the same line. For more information, see the `annotations` object in "Check Runs."

Now you'll add code to extract the required information from RuboCop that's needed to create each annotation.

Under the code you added in the previous step, where it says `# ADD ANNOTATIONS CODE HERE #`, add the following code:

Ruby

```ruby
annotations = []
# You can create a maximum of 50 annotations per request to the Checks
# API. To add more than 50 annotations, use the "Update a check run" API
# endpoint. This example code limits the number of annotations to 50.
# See /rest/reference/checks#update-a-check-run
# for details.
max_annotations = 50

# RuboCop reports the number of errors found in "offense_count"
if @output['summary']['offense_count'] == 0
  conclusion = 'success'
else
  conclusion = 'neutral'
```

```
      @output['files'].each do |file|

        # Only parse offenses for files in this app's repository
        file_path = file['path'].gsub(/#{repository}\//,'')
        annotation_level = 'notice'

        # Parse each offense to get details and location
        file['offenses'].each do |offense|
          # Limit the number of annotations to 50
          next if max_annotations == 0
          max_annotations -= 1

          start_line   = offense['location']['start_line']
          end_line     = offense['location']['last_line']
          start_column = offense['location']['start_column']
          end_column   = offense['location']['last_column']
          message      = offense['message']

          # Create a new annotation for each error
          annotation = {
            path: file_path,
            start_line: start_line,
            end_line: end_line,
            start_column: start_column,
            end_column: end_column,
            annotation_level: annotation_level,
            message: message
          }
          # Annotations only support start and end columns on the same line
          if start_line == end_line
            annotation.merge({start_column: start_column, end_column:
end_column})
          end

          annotations.push(annotation)
        end
      end
    end

    # ADD CODE HERE TO UPDATE CHECK RUN SUMMARY #
```

This code limits the total number of annotations to 50. But you can modify this code to update the check run for each batch of 50 annotations. The code above includes the variable `max_annotations` that sets the limit to 50, which is used in the loop that iterates through the offenses.

When the `offense_count` is zero, the CI test is a `success`. If there are errors, this code sets the conclusion to `neutral` in order to prevent strictly enforcing errors from code linters. But you can change the conclusion to `failure` if you would like to ensure that the check suite fails when there are linting errors.

When errors are reported, the code above iterates through the `files` array in the RuboCop report. For each file, it extracts the file path and sets the annotation level to `notice`. You could go even further and set specific warning levels for each type of RuboCop Cop, but to keep things simpler in this tutorial, all errors are set to a level of `notice`.

This code also iterates through each error in the `offenses` array and collects the location of the offense and error message. After extracting the information needed, the code creates an annotation for each error and stores it in the `annotations` array. Because annotations only support start and end columns on the same line, `start_column` and `end_column` are only added to the `annotation` object if the start and end line values are the same.

This code doesn't yet create an annotation for the check run. You'll add that code in the next section.

Step 2.5. Update the check run with CI test results

Each check run from GitHub contains an `output` object that includes a `title`, `summary`, `text`, `annotations`, and `images`. The `summary` and `title` are the only required parameters for the `output`, but those alone don't offer much detail, so this tutorial also adds `text` and `annotations`.

For the `summary`, this example uses the summary information from RuboCop and adds newlines (`\n`) to format the output. You can customize what you add to the `text` parameter, but this example sets the `text` parameter to the RuboCop version. The following code sets the `summary` and `text`.

Under the code you added in the previous step, where it says `# ADD CODE HERE TO UPDATE CHECK RUN SUMMARY #`, add the following code:

Ruby

```
        # Updated check run summary and text parameters
        summary = "Octo RuboCop summary\n-Offense count: #{@output['summary']
['offense_count']}\n-File count: #{@output['summary']['target_file_count']}\n-
Target file count: #{@output['summary']['inspected_file_count']}"
        text = "Octo RuboCop version: #{@output['metadata']['rubocop_version']}"
```

Now your code should have all the information it needs to update your check run. In "Step 1.3. Update a check run," you added code to set the status of the check run to `success`. You'll need to update that code to use the `conclusion` variable you set based on the RuboCop results (to `success` or `neutral`). Here's the code you added previously to your `server.rb` file:

```
# Mark the check run as complete!
@installation_client.update_check_run(
  @payload['repository']['full_name'],
  @payload['check_run']['id'],
  status: 'completed',
  conclusion: 'success',
  accept: 'application/vnd.github+json'
)
```

Replace that code with the following code:

Ruby

```
        # Mark the check run as complete! And if there are warnings, share them.
        @installation_client.update_check_run(
          @payload['repository']['full_name'],
          @payload['check_run']['id'],
          status: 'completed',
          conclusion: conclusion,
          output: {
            title: 'Octo RuboCop',
            summary: summary,
            text: text,
            annotations: annotations
          },
          actions: [{
            label: 'Fix this',
            description: 'Automatically fix all linter notices.',
            identifier: 'fix_rubocop_notices'
          }],
          accept: 'application/vnd.github+json'
        )
```

Now that your code sets a conclusion based on the status of the CI test, and adds the output from the RuboCop results, you've created a CI test.

The code above also adds a feature called requested actions to your CI server, via the `actions` object. (Note this is not related to GitHub Actions.) For more information, see "Request further actions from a check run." Requested actions add a button in the **Checks** tab on GitHub that allows someone to request the check run to take additional action. The additional action is completely configurable by your app. For example, because RuboCop has a feature to automatically fix the errors it finds in Ruby code, your CI server can use a requested actions button to allow people to request automatic error fixes. When someone clicks the button, the app receives the `check_run` event with a `requested_action` action. Each requested action has an `identifier` that the app uses to determine which button was clicked.

The code above doesn't have RuboCop automatically fix errors yet. You'll add that later in the tutorial.

Test the code

The following steps will show you how to test that the code works and view the CI test that you just created.

1. Run the following command to restart the server from your terminal. If the server is already running, first enter `Ctrl-C` in your terminal to stop the server, and then run the following command to start the server again.

   Shell

   ```
   ruby server.rb
   ```

2. In the repository where you added the `myfile.rb` file, create a new pull request.
3. In the pull request you just created, navigate to the **Checks** tab. You should see annotations for each of the errors that RuboCop found. Also notice the "Fix this" button that you created by adding a requested action.

Step 2.6. Automatically fix RuboCop errors

So far you've created a CI test. In this section, you'll add one more feature that uses RuboCop to automatically fix the errors it finds. You already added the "Fix this" button in "Step 2.5. Update the check run with CI test results." Now you'll add the code to handle the `requested_action` check run event that's triggered when someone clicks the "Fix this" button.

The RuboCop tool offers the `--auto-correct` command-line option to automatically fix the errors it finds. For more information, see "Autocorrecting offenses" in the RuboCop documentation. When you use the `--auto-correct` feature, the updates are applied to the local files on the server. You'll need to push the changes to GitHub after RuboCop makes the fixes.

To push to a repository, your app must have write permissions for "Contents" in a repository. You already set that permission to **Read & write** back in "Step 2.2. Cloning the repository."

To commit files, Git must know which username and email address to associate with the commit. Next you'll add environment variables to store the name and email address that your app will use when it makes Git commits.

1. Open the `.env` file you created earlier in this tutorial.
2. Add the following environment variables to your `.env` file. Replace `APP_NAME` with the name of your app, and `EMAIL_ADDRESS` with any email you'd like to use for this example.

```shell
GITHUB_APP_USER_NAME="APP_NAME"
GITHUB_APP_USER_EMAIL="EMAIL_ADDRESS"
```

Next you'll need to add code to read the environment variables and set the Git configuration. You'll add that code soon.

When someone clicks the "Fix this" button, your app receives the check run webhook with the `requested_action` action type.

In "Step 1.3. Updating a check run" you updated the `event_handler` in your `server.rb` file to look for actions in the `check_run` event. You already have a case statement to handle the `created` and `rerequested` action types:

```ruby
when 'check_run'
  # Check that the event is being sent to this app
  if @payload['check_run']['app']['id'].to_s === APP_IDENTIFIER
    case @payload['action']
    when 'created'
      initiate_check_run
    when 'rerequested'
      create_check_run
    # ADD REQUESTED_ACTION METHOD HERE #
  end
end
```

After the `rerequested` case, where it says `# ADD REQUESTED_ACTION METHOD HERE #`, add the following code:

```ruby
    when 'requested_action'
      take_requested_action
```

This code calls a new method that will handle all `requested_action` events for your app.

In the code block that starts with `helpers do`, where it says `# ADD TAKE_REQUESTED_ACTION HELPER METHOD HERE #`, add the following helper method:

```ruby
    # Handles the check run `requested_action` event
    # See /webhooks/event-payloads/#check_run
    def take_requested_action
      full_repo_name = @payload['repository']['full_name']
      repository     = @payload['repository']['name']
      head_branch    = @payload['check_run']['check_suite']['head_branch']

      if (@payload['requested_action']['identifier'] == 'fix_rubocop_notices')
        clone_repository(full_repo_name, repository, head_branch)

        # Sets your commit username and email address
        @git.config('user.name', ENV['GITHUB_APP_USER_NAME'])
```

```
        @git.config('user.email', ENV['GITHUB_APP_USER_EMAIL'])

        # Automatically correct RuboCop style errors
        @report = `rubocop '#{repository}/*' --format json --auto-correct`

        pwd = Dir.getwd()
        Dir.chdir(repository)
        begin
          @git.commit_all('Automatically fix Octo RuboCop notices.')
          @git.push("https://x-access-token:#
{@installation_token.to_s}@github.com/#{full_repo_name}.git", head_branch)
        rescue
          # Nothing to commit!
          puts 'Nothing to commit'
        end
        Dir.chdir(pwd)
        `rm -rf '#{repository}'`
      end
    end
```

The code above clones a repository, just like the code you added in "Step 2.2. Clone the repository." An `if` statement checks that the requested action's identifier matches the RuboCop button identifier (`fix_rubocop_notices`). When they match, the code clones the repository, sets the Git username and email, and runs RuboCop with the option `--auto-correct` . The `--auto-correct` option applies the changes to the local CI server files automatically.

The files are changed locally, but you'll still need to push them to GitHub. You'll use the `ruby-git` gem to commit all of the files. Git has a single command that stages all modified or deleted files and commits them: `git commit -a` . To do the same thing using `ruby-git` , the code above uses the `commit_all` method. Then the code pushes the committed files to GitHub using the installation token, using the same authentication method as the Git `clone` command. Finally, it removes the repository directory to ensure the working directory is prepared for the next event.

The code you have written now completes your continuous integration server that you built using a GitHub App and checks. To see the full final code for your app, see "Full code example."

Test the code

The following steps will show you how to test that the code works, and that RuboCop can automatically fix the errors it finds.

1. Run the following command to restart the server from your terminal. If the server is already running, first enter `Ctrl-C` in your terminal to stop the server, and then run the following command to start the server again.

   Shell

   ```
   ruby server.rb
   ```

2. In the repository where you added the `myfile.rb` file, create a new pull request.
3. In the new pull request you created, navigate to the **Checks** tab, and click the "Fix this" button to automatically fix the errors RuboCop found.
4. Navigate to the **Commits** tab. You should see a new commit by the username you set in your Git configuration. You may need to refresh your browser to see the update.

5. Navigate to the **Checks** tab. You should see a new check suite for Octo RuboCop. But this time there should be no errors, because RuboCop fixed them all.

Full code example

This is what the final code in `server.rb` should look like, after you've followed all of the steps in this tutorial. There are also comments throughout the code that provide additional context.

Ruby

```ruby
require 'sinatra/base'  # Use the Sinatra web framework
require 'octokit'       # Use the Octokit Ruby library to interact with GitHub's
REST API
require 'dotenv/load'   # Manages environment variables
require 'json'          # Allows your app to manipulate JSON data
require 'openssl'       # Verifies the webhook signature
require 'jwt'           # Authenticates a GitHub App
require 'time'          # Gets ISO 8601 representation of a Time object
require 'logger'        # Logs debug statements

# This code is a Sinatra app, for two reasons:
#   1. Because the app will require a landing page for installation.
#   2. To easily handle webhook events.

class GHAapp < Sinatra::Application

  # Sets the port that's used when starting the web server.
  set :port, 3000
  set :bind, '0.0.0.0'

  # Expects the private key in PEM format. Converts the newlines.
  PRIVATE_KEY = OpenSSL::PKey::RSA.new(ENV['GITHUB_PRIVATE_KEY'].gsub('\n',
"\n"))

  # Your registered app must have a webhook secret.
  # The secret is used to verify that webhooks are sent by GitHub.
  WEBHOOK_SECRET = ENV['GITHUB_WEBHOOK_SECRET']

  # The GitHub App's identifier (type integer).
  APP_IDENTIFIER = ENV['GITHUB_APP_IDENTIFIER']

  # Turn on Sinatra's verbose logging during development
  configure :development do
    set :logging, Logger::DEBUG
  end

  # Executed before each request to the `/event_handler` route
  before '/event_handler' do
    get_payload_request(request)
    verify_webhook_signature

    # If a repository name is provided in the webhook, validate that
    # it consists only of latin alphabetic characters, `-`, and `_`.
    unless @payload['repository'].nil?
      halt 400 if (@payload['repository']['name'] =~ /[0-9A-Za-z\-\_]+/).nil?
    end

    authenticate_app
    # Authenticate the app installation in order to run API operations
    authenticate_installation(@payload)
  end

  post '/event_handler' do
```

```
    # Get the event type from the HTTP_X_GITHUB_EVENT header
    case request.env['HTTP_X_GITHUB_EVENT']

    when 'check_suite'
      # A new check_suite has been created. Create a new check run with status
queued
      if @payload['action'] == 'requested' || @payload['action'] == 'rerequested'
        create_check_run
      end

    when 'check_run'
      # Check that the event is being sent to this app
      if @payload['check_run']['app']['id'].to_s === APP_IDENTIFIER
        case @payload['action']
        when 'created'
          initiate_check_run
        when 'rerequested'
          create_check_run
        when 'requested_action'
          take_requested_action
        end
      end
    end

    200 # success status
  end

  helpers do

    # Create a new check run with status "queued"
    def create_check_run
      @installation_client.create_check_run(
        # [String, Integer, Hash, Octokit Repository object] A GitHub repository.
        @payload['repository']['full_name'],
        # [String] The name of your check run.
        'Octo RuboCop',
        # [String] The SHA of the commit to check
        # The payload structure differs depending on whether a check run or a
check suite event occurred.
        @payload['check_run'].nil? ? @payload['check_suite']['head_sha'] :
@payload['check_run']['head_sha'],
        # [Hash] 'Accept' header option, to avoid a warning about the API not
being ready for production use.
        accept: 'application/vnd.github+json'
      )
    end

    # Start the CI process
    def initiate_check_run
      # Once the check run is created, you'll update the status of the check run
      # to 'in_progress' and run the CI process. When the CI finishes, you'll
      # update the check run status to 'completed' and add the CI results.

      @installation_client.update_check_run(
        @payload['repository']['full_name'],
        @payload['check_run']['id'],
        status: 'in_progress',
        accept: 'application/vnd.github+json'
      )

      full_repo_name = @payload['repository']['full_name']
      repository     = @payload['repository']['name']
      head_sha       = @payload['check_run']['head_sha']

      clone_repository(full_repo_name, repository, head_sha)

      # Run RuboCop on all files in the repository
      @report = `rubocop '#{repository}' --format json`
      logger.debug @report
      `rm -rf #{repository}`
      @output = JSON.parse @report
```

```
      annotations = []
      # You can create a maximum of 50 annotations per request to the Checks
      # API. To add more than 50 annotations, use the "Update a check run" API
      # endpoint. This example code limits the number of annotations to 50.
      # See /rest/reference/checks#update-a-check-run
      # for details.
      max_annotations = 50

      # RuboCop reports the number of errors found in "offense_count"
      if @output['summary']['offense_count'] == 0
        conclusion = 'success'
      else
        conclusion = 'neutral'
        @output['files'].each do |file|

          # Only parse offenses for files in this app's repository
          file_path = file['path'].gsub(/#{repository}\//,'')
          annotation_level = 'notice'

          # Parse each offense to get details and location
          file['offenses'].each do |offense|
            # Limit the number of annotations to 50
            next if max_annotations == 0
            max_annotations -= 1

            start_line   = offense['location']['start_line']
            end_line     = offense['location']['last_line']
            start_column = offense['location']['start_column']
            end_column   = offense['location']['last_column']
            message      = offense['message']

            # Create a new annotation for each error
            annotation = {
              path: file_path,
              start_line: start_line,
              end_line: end_line,
              start_column: start_column,
              end_column: end_column,
              annotation_level: annotation_level,
              message: message
            }
            # Annotations only support start and end columns on the same line
            if start_line == end_line
              annotation.merge({start_column: start_column, end_column:
end_column})
            end

            annotations.push(annotation)
          end
        end
      end

      # Updated check run summary and text parameters
      summary = "Octo RuboCop summary\n-Offense count: #{@output['summary']
['offense_count']}\n-File count: #{@output['summary']['target_file_count']}\n-
Target file count: #{@output['summary']['inspected_file_count']}"
      text = "Octo RuboCop version: #{@output['metadata']['rubocop_version']}"

      # Mark the check run as complete! And if there are warnings, share them.
      @installation_client.update_check_run(
        @payload['repository']['full_name'],
        @payload['check_run']['id'],
        status: 'completed',
        conclusion: conclusion,
        output: {
          title: 'Octo RuboCop',
          summary: summary,
          text: text,
          annotations: annotations
        },
```

```
        actions: [{
          label: 'Fix this',
          description: 'Automatically fix all linter notices.',
          identifier: 'fix_rubocop_notices'
        }],
        accept: 'application/vnd.github+json'
      )
    end

    # Clones the repository to the current working directory, updates the
    # contents using Git pull, and checks out the ref.
    #
    # full_repo_name  - The owner and repo. Ex: octocat/hello-world
    # repository      - The repository name
    # ref             - The branch, commit SHA, or tag to check out
    def clone_repository(full_repo_name, repository, ref)
      @git = Git.clone("https://x-access-token:#
{@installation_token.to_s}@github.com/#{full_repo_name}.git", repository)
      pwd = Dir.getwd()
      Dir.chdir(repository)
      @git.pull
      @git.checkout(ref)
      Dir.chdir(pwd)
    end

    # Handles the check run `requested_action` event
    # See /webhooks/event-payloads/#check_run
    def take_requested_action
      full_repo_name = @payload['repository']['full_name']
      repository     = @payload['repository']['name']
      head_branch    = @payload['check_run']['check_suite']['head_branch']

      if (@payload['requested_action']['identifier'] == 'fix_rubocop_notices')
        clone_repository(full_repo_name, repository, head_branch)

        # Sets your commit username and email address
        @git.config('user.name', ENV['GITHUB_APP_USER_NAME'])
        @git.config('user.email', ENV['GITHUB_APP_USER_EMAIL'])

        # Automatically correct RuboCop style errors
        @report = `rubocop '#{repository}/*' --format json --auto-correct`

        pwd = Dir.getwd()
        Dir.chdir(repository)
        begin
          @git.commit_all('Automatically fix Octo RuboCop notices.')
          @git.push("https://x-access-token:#
{@installation_token.to_s}@github.com/#{full_repo_name}.git", head_branch)
        rescue
          # Nothing to commit!
          puts 'Nothing to commit'
        end
        Dir.chdir(pwd)
        `rm -rf '#{repository}'`
      end
    end

    # Saves the raw payload and converts the payload to JSON format
    def get_payload_request(request)
      # request.body is an IO or StringIO object
      # Rewind in case someone already read it
      request.body.rewind
      # The raw text of the body is required for webhook signature verification
      @payload_raw = request.body.read
      begin
        @payload = JSON.parse @payload_raw
      rescue => e
        fail  'Invalid JSON (#{e}): #{@payload_raw}'
      end
    end
```

```ruby
    # Instantiate an Octokit client authenticated as a GitHub App.
    # GitHub App authentication requires that you construct a
    # JWT (https://jwt.io/introduction/) signed with the app's private key,
    # so GitHub can be sure that it came from the app an not altererd by
    # a malicious third party.
    def authenticate_app
      payload = {
          # The time that this JWT was issued, _i.e._ now.
          iat: Time.now.to_i,

          # JWT expiration time (10 minute maximum)
          exp: Time.now.to_i + (10 * 60),

          # Your GitHub App's identifier number
          iss: APP_IDENTIFIER
      }

      # Cryptographically sign the JWT.
      jwt = JWT.encode(payload, PRIVATE_KEY, 'RS256')

      # Create the Octokit client, using the JWT as the auth token.
      @app_client ||= Octokit::Client.new(bearer_token: jwt)
    end

    # Instantiate an Octokit client, authenticated as an installation of a
    # GitHub App, to run API operations.
    def authenticate_installation(payload)
      @installation_id = payload['installation']['id']
      @installation_token =
@app_client.create_app_installation_access_token(@installation_id)[:token]
      @installation_client = Octokit::Client.new(bearer_token:
@installation_token)
    end

    # Check X-Hub-Signature to confirm that this webhook was generated by
    # GitHub, and not a malicious third party.
    #
    # GitHub uses the WEBHOOK_SECRET, registered to the GitHub App, to
    # create the hash signature sent in the `X-HUB-Signature` header of each
    # webhook. This code computes the expected hash signature and compares it to
    # the signature sent in the `X-HUB-Signature` header. If they don't match,
    # this request is an attack, and you should reject it. GitHub uses the HMAC
    # hexdigest to compute the signature. The `X-HUB-Signature` looks something
    # like this: 'sha1=123456'.
    def verify_webhook_signature
      their_signature_header = request.env['HTTP_X_HUB_SIGNATURE'] || 'sha1='
      method, their_digest = their_signature_header.split('=')
      our_digest = OpenSSL::HMAC.hexdigest(method, WEBHOOK_SECRET, @payload_raw)
      halt 401 unless their_digest == our_digest

      # The X-GITHUB-EVENT header provides the name of the event.
      # The action value indicates the which action triggered the event.
      logger.debug "---- received event #{request.env['HTTP_X_GITHUB_EVENT']}"
      logger.debug "----    action #{@payload['action']}" unless
@payload['action'].nil?
    end

  end

  # Finally some logic to let us run this server directly from the command line,
  # or with Rack. Don't worry too much about this code. But, for the curious:
  # $0 is the executed file
  # __FILE__ is the current file
  # If they are the same—that is, we are running this file directly, call the
  # Sinatra run method
  run! if __FILE__ == $0
end
```

Next steps

You should now have an app that receives API events, creates check runs, uses RuboCop to find Ruby errors, creates annotations in a pull request, and automatically fixes linter errors. Next you might want to expand your app's code, deploy your app, and make your app public.

If you have any questions, start a GitHub Community discussion in the API and Webhooks category.

Modify the app code

This tutorial demonstrated how to create a "Fix this" button that is always displayed in pull requests in the repository. Try updating the code to display the "Fix this" button only when RuboCop finds errors.

If you'd prefer that RuboCop doesn't commit files directly to the head branch, update the code to instead create a pull request with a new branch that's based on the head branch.

Deploy your app

This tutorial demonstrated how to develop your app locally. When you are ready to deploy your app, you need to make changes to serve your app and keep your app's credential secure. The steps you take depend on the server that you use, but the following sections offer general guidance.

Host your app on a server

This tutorial used your computer or codespace as a server. Once the app is ready for production use, you should deploy your app to a dedicated server. For example, you can use Azure App Service.

Update the webhook URL

Once you have a server that is set up to receive webhook traffic from GitHub, update the webhook URL in your app settings. You should not use Smee.io to forward your webhooks in production.

Update the `:port` setting

When you deploy your app, you will want to change the port where your server is listening. The code already tells your server to listen to all available network interfaces by setting `:bind` to `0.0.0.0`.

For example, you can set a `PORT` variable in your `.env` file on your server to indicate the port where your server should listen. Then, you can update the place where your code sets `:port` so that your server listens on your deployment port:

Ruby

```ruby
set :port, ENV['PORT']
```

Secure your app's credentials

You should never publicize your app's private key or webhook secret. This tutorial stored your app's credentials in a gitignored `.env` file. When you deploy your app, you should

choose a secure way to store the credentials and update your code to get the value accordingly. For example, you can store the credentials with a secret management service like Azure Key Vault. When your app runs, it can retrieve the credentials and store them in environment variables on the server where your app is deployed.

For more information, see "Best practices for creating a GitHub App."

Share your app

If you want to share your app with other users and organizations, make your app public. For more information, see "Making a GitHub App public or private."

Follow best practices

You should aim to follow best practices with your GitHub App. For more information, see "Best practices for creating a GitHub App."