Creating CI tests with the Checks API

In this article

Build a continuous integration server to run tests using a GitHub App and the Checks API.

Introduction

This guide will introduce you to GitHub Apps and the Checks API, which you'll use to build a continuous integration (CI) server that runs tests.

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 some examples of the types of CI tests you can create with a GitHub App, check out the continuous integration apps available in GitHub Marketplace.

Note: This guide demonstrates the app development process using the Ruby programming language. However, there are many flavors of Octokit. If you prefer JavaScript, you can use Probot and Node.js to develop GitHub Apps.

Checks API overview

The Checks API allows you to set up CI tests that are automatically run against each code commit in a repository. The Checks API reports detailed information about each check on GitHub in the pull request's **Checks** tab. With the Checks API, you can create annotations with additional details for 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.

A *check suite* is a group of *check runs* (individual CI tests). Both the suite and the runs contain *statuses* that are visible in a pull request on GitHub. You can use statuses to determine when a code commit introduces errors. Using these statuses with protected branches can prevent people from merging pull requests prematurely. See "About protected branches" for more details.

The Checks API sends the `check_suite` webhook event to all GitHub Apps installed on a repository each time new code is pushed to the repository. To receive all Checks API event actions, the app must have the `checks:write` permission. GitHub automatically creates `check_suite` events for new code commits in a repository using the default flow, although Update repository preferences for check suites if you'd like. Here's how the default flow works:

1. Whenever someone pushes code to the repository, GitHub 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 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.

In this guide, you'll learn how to:

- Part 1: Set up the framework for a CI server using the Checks API.
  - Configure a GitHub App as a server that receives Checks API events.
  - Create new check runs for CI tests when a repository receives newly pushed commits.
  - Re-run check runs when a user requests that action on GitHub.
- Part 2: Build on the CI server framework you created by adding a linter CI test.
  - Update a check run with a `status`, `conclusion`, and `output` details.
  - Create annotations on lines of code that GitHub displays in the **Checks** and **Files Changed** tab of a pull request.
  - Automatically fix linter recommendations by exposing a "Fix this" button in the **Checks** tab of the pull request.

Prerequisites

Before you get started, you may want to familiarize yourself with GitHub Apps, Webhooks, and the Checks API, if you're not already. You'll find more APIs in the REST API docs. The Checks API is also available to use in GraphQL, but this quickstart focuses on REST. See the GraphQL Checks Suite and Check Run objects for more details.

You'll use the Ruby programming language, the Smee webhook payload delivery service, the Octokit.rb Ruby library for the GitHub REST API, and the Sinatra web framework to create your Checks API CI server app.

You don't need to be an expert in any of these tools or concepts to complete this project. This guide will walk you through all the required steps. Before you begin creating CI tests with the Checks API, you'll need to do the following:

1. Clone the Creating CI tests with the Checks API repository.

   ```
   $ git clone https://github.com/github-developer/creating-ci-tests-with-the-chec
   ```

   Inside the directory, you'll find a `template_server.rb` file with the template code you'll use in this quickstart and a `server.rb` file with the completed project code.

2. Follow the steps in the "Setting up your development environment to create a GitHub App" quickstart to configure and run the app server. **Note:** Instead of cloning the GitHub App template repository, use the `template_server.rb` file in the repository you cloned in the previous step in this quickstart.

   If you've completed a GitHub App quickstart before, make sure to register a *new* GitHub App and start a new Smee channel to use with this quickstart.

   See the troubleshooting section if you are running into problems setting up your template 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 be performing 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. If you haven't completed the steps in "Setting up your development environment to create a GitHub App," you'll need to do that before you can continue.

Let's get started! These are the steps you'll complete in Part 1:

1. Updating app permissions
2. Adding event handling
3. Creating a check run
4. Updating a check run

Step 1.1. Updating app permissions

When you first registered your app, you accepted the default permissions, which means your app doesn't have access to most resources. For this example, your app will need permission to read and write checks.

To update your app's permissions:

1. Select your app from the app settings page and click **Permissions & Webhooks** in the sidebar.
2. In the "Permissions" section, find "Checks", and select **Read & write** in the Access dropdown next to it.
3. In the "Subscribe to events" section, select **Check suite** and **Check run** to subscribe to these events.
4. Click **Save changes** at the bottom of the page.
5. 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 and clicking on "Configure" next to your app. You'll see a banner at the top of the page letting you know that the app is requesting different permissions. Click "Details" and click "Accept new permissions."

Great! Your app has permission to do the tasks you want it to do. Now you can add the code to handle the events.

Step 1.2. Adding event handling

Now that your app is subscribed to the **Check suite** and **Check run** events, it will start receiving 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.

An empty `post '/event_handler'` route is already included in the `template_server.rb` file, which you downloaded in the prerequisites section. The empty route looks like this:

```
  post '/event_handler' do

    # # # # # # # # # # # #
    # ADD YOUR CODE HERE  #
    # # # # # # # # # # # #

    200 # success status
  end
```

Use this route to handle the `check_suite` event by adding the following code:

```
# 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
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 1.3. Creating a check run

You'll add this new method as a Sinatra helper in case you want other routes to use it too. Under `helpers do`, add this `create_check_run` method:

```
# 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 sui
    @payload['check_run'].nil? ? @payload['check_suite']['head_sha'] : @payload['ch
    # [Hash] 'Accept' header option, to avoid a warning about the API not being rea
    accept: 'application/vnd.github+json'
  )
end
```

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

To create a check run, only two input parameters are required: `name` and `head_sha`. We will use RuboCop to implement the CI test later in this quickstart, which is why the name "Octo RuboCop" is used here, but you can choose any name you'd like for the check run.

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.

In the code above, you're using the 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.

To test this code, restart the server from your terminal:

```
$ ruby template_server.rb
```

> **Note:** You'll need to restart the Sinatra server before you can test changes. Enter `Ctrl-C` to stop the server, and then run `ruby template_server.rb` again. If you don't want to do this every time you change your app code, you can look into reloading.

Now open a pull request in the repository where you installed your app. Your app should

respond by creating a check run on your pull request. Click on the **Checks** tab, and 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.

Great! You've told GitHub to create a check run. You can see the check run status is set to `queued` next to a yellow icon. Next, you'll want to wait for GitHub to create the check run and update its status.

Step 1.4. Updating 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 want to 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 want to start the process all over and create a new check run.

To include a condition for the `check_run` event in the `post '/event_handler'` route, add the following code under `case request.env['HTTP_X_GITHUB_EVENT']`:

```
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
    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. Add the following code to the helpers section:

```
# 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.
```

```ruby
  @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 "Checks" API endpoint using the `update_check_run` Octokit method to update 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 quickstart, 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`.

You'll notice in the "Checks" docs that when you provide a 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. Let's test this code again by re-running `template_server.rb`:

```
$ ruby template_server.rb
```

Head over to your open pull request and click the **Checks** tab. Click the "Re-run all" button in the upper right corner. You should see the check run move from `pending` to `in_progress` and end with `success`.

Part 2. Creating the Octo RuboCop CI test

RuboCop is a Ruby code linter and formatter. It checks Ruby code to ensure that it complies with the "Ruby Style Guide." RuboCop has three primary functions:

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

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

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 Checks 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. That information can be anything: for example, a comment, an error, or a warning. This quickstart 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 quickstart 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.

Let's get started! These are the steps you'll complete in this section:

1. Adding a Ruby file
2. Cloning the repository
3. Running RuboCop
4. Collecting RuboCop errors
5. Updating the check run with CI test results
6. Automatically fixing RuboCop errors
7. Security tips

Step 2.1. Adding a Ruby file

You can pass specific files or entire directories for RuboCop to check. In this quickstart, you'll run RuboCop on an entire directory. Because RuboCop only checks Ruby code, you'll want at least one Ruby file in your repository that contains errors. The example file provided below contains a few errors. Add this example Ruby file to the repository where your app is installed (make sure to name the file with an `.rb` extension, as in `myfile.rb`):

```ruby
# 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
```

Step 2.2. Cloning the repository

RuboCop is available as a command-line utility. That means your GitHub App will need to clone a local copy of the repository on the CI server so RuboCop can parse the files. To

run Git operations in your Ruby app, you can use the ruby-git gem.

The `Gemfile` in the `building-a-checks-api-ci-server` repository already includes the ruby-git gem, and you installed it when you ran `bundle install` in the prerequisite steps. To use the gem, add this code to the top of your `template_server.rb` file:

```
require 'git'
```

Your app needs read permission for "Repository contents" to clone a repository. Later in this quickstart, you'll need to push contents to GitHub, which requires write permission. Go ahead and set your app's "Repository contents" permission to **Read & write** now so you don't need to update it again later. To update your app's permissions:

1. Select your app from the app settings page and click **Permissions & Webhooks** in the sidebar.
2. In the "Permissions" section, find "Repository contents", and select **Read & write** in the "Access" dropdown next to it.
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 and clicking on "Configure" next to your app. You'll see a banner at the top of the page letting you know that the app is requesting different permissions. Click "Details" and click "Accept new permissions."

To clone a repository using your GitHub App's permissions, you can use the app's installation token (`x-access-token:<token>`) shown in the example below:

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

The code 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`.

After your app clones the repository, it needs to pull the latest code changes and check out a specific Git ref. The code to do all of this will fit 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. Add the following new method to the helper method section in `template_server.rb`:

```
# 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/#
  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. This code clones the code in the same directory as `template_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`), checks out the ref (`@git.checkout(ref)`), then 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. Add the following code under the `***** RUN A CI TEST *****` comment in your `initiate_check_run` helper method:

```
# ***** RUN A CI TEST *****
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)
```

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

Step 2.3. Running RuboCop

Great! You're cloning the repository and creating check runs using your CI server. Now you'll get into the nitty gritty details of the RuboCop linter and Checks API annotations.

The following code runs RuboCop and saves the style code errors in JSON format. Add this code below the call to `clone_repository` you added in the previous step and above the code that updates the check run to complete.

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

The code above runs RuboCop on all files in the repository's directory. The option `--format json` is a handy way to save a copy of the linting results in a machine-parsable format. See the RuboCop docs for details and an example of the JSON format.

Because this code stores the RuboCop results in a `@report` variable, it can safely remove the checkout of the repository. This code also parses the JSON so you can easily access the keys and values in your GitHub App using the `@output` variable.

> **Note:** The command used to remove the repository (`rm -rf`) cannot be undone. See Step 2.7. Security tips to learn how to check 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. If for some reason you're *not* using the method `verify_webhook_signature` (which is included in `template_server.rb`) to validate the sender of the webhook, make sure you check that the repository name is valid.

You can test that this code works and see the errors reported by RuboCop in your server's debug output. Start up the `template_server.rb` server again and create a new

pull request in the repository where you're testing your app:

```
$ ruby template_server.rb
```

You should see the linting errors in the debug output, although they aren't printed with formatting. You can use a web tool like JSON formatter to format your JSON output like this formatted linting error output:


```
{
  "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 d
          "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 d
          "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. Collecting RuboCop errors

The `@output` variable contains the parsed JSON results of the RuboCop report. As shown above, 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 Checks API allows you to create annotations for specific lines of code. When you create or update a check run, you can add annotations. In this quickstart you are updating the check run with annotations.

The Checks API limits the number of annotations to a maximum of 50 per API request. To create more than 50 annotations, you have to make multiple requests to the Checks endpoint. For example, to create 105 annotations you'd need to call the Checks endpoint three times. 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. See the `annotations` object reference documentation for details.

You'll extract the required information from RuboCop needed to create each annotation. Append the following code to the code you added in the previous section:

```
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
```

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 quickstart, 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. Updating 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 quickstart adds `text` and `annotations` too. The code here doesn't add an image, but feel free to add one if you'd like!

For the `summary`, this example uses the summary information from RuboCop and adds some 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. To set the `summary` and `text`, append this code to the code you added in the previous section:

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

Now you've got all the information you need to update your check run. In the first half of this quickstart, you added this code to set the status of the check run to `success`:

```
# 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'
)
```

You'll need to update that code to use the `conclusion` variable you set based on the RuboCop results (to `success` or `neutral`). You can update the code with the following:

```
# 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 you're setting a conclusion based on the status of the CI test and you've added the output from the RuboCop results, you've created a CI test! Congratulations.

The code above also adds a feature to your CI server called requested actions via the `actions` object. 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 in the next section. But first, take a look at the CI test that you just created by starting up the `template_server.rb` server again and creating a new pull request:

```
$ ruby template_server.rb
```

The annotations will show up in the **Checks** tab. Also notice the "Fix this" button that you created by adding a requested action.

If the annotations are related to a file already included in the PR, the annotations will also show up in the **Files changed** tab.

Step 2.6. Automatically fixing RuboCop errors

If you've made it this far, kudos! You've already 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 the previous section. Now you'll add the code to handle the `requested_action` check run event triggered when someone clicks the "Fix this" button.

The RuboCop tool offers the `--auto-correct` command-line option to automatically fix errors it finds. 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 does its magic.

To push to a repository, your app must have write permissions for "Repository contents." You set that permission back in Step 2.2. Cloning the repository to **Read & write**, so you're all set.

In order to commit files, Git must know which username and email to associate with the commit. Add two more environment variables in your `.env` file to store the name (`GITHUB_APP_USER_NAME`) and email (`GITHUB_APP_USER_EMAIL`) settings. Your name can be the name of your app and the email can be any email you'd like for this example. For example:

```
GITHUB_APP_USER_NAME=Octoapp
GITHUB_APP_USER_EMAIL=octoapp@octo-org.com
```

Once you've updated your `.env` file with the name and email of the author and committer, you'll be ready 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 1.4. Updating a check run you updated the your `event_handler` to handle 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
  end
end
```

Add another `when` statement after the `rerequested` case to handle the `rerequested_action` event:

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

This code calls a new method that will handle all `requested_action` events for your app. Add the following method to the helper methods section of your code:

```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/#{fu
    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. Cloning 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 handy `ruby-git` gem again 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.

That's it! The code you have written now completes your Checks API CI server. Restart your `template_server.rb` server again and create a new pull request:

```
$ ruby template_server.rb
```

Note: You'll need to restart the Sinatra server before you can test changes. Enter `Ctrl-C` to stop the server, and then run `ruby template_server.rb` again. If you don't want to do this every time you change your app code, you can look into reloading.

This time, click the "Fix this" button to automatically fix the errors RuboCop found from the **Checks** tab.

In the **Commits** tab, you'll see a brand new commit by the username you set in your Git configuration. You may need to refresh your browser to see the update.

Because a new commit was pushed to the repo, you'll see a new check suite for Octo RuboCop in the **Checks** tab. But this time there are no errors because RuboCop fixed them all.

You can find the completed code for the app you just built in the `server.rb` file in the Creating CI tests with the Checks API repository.

Step 2.7. Security tips

The template GitHub App code already has a method to verify incoming webhook payloads to ensure they are from a trusted source. If you are not validating webhook payloads, you'll need to ensure that when repository names are included in the webhook

payload, the webhook does not contain arbitrary commands that could be used maliciously. The code below validates that the repository name only contains Latin alphabetic characters, hyphens, and underscores. To provide you with a complete example, the complete `server.rb` code available in the companion repository for this quickstart includes both the method of validating incoming webhook payloads and this check to verify the repository name.

```
# This quickstart example uses the repository name in the webhook with
# command-line utilities. For security reasons, you should validate the
# repository name to ensure that a bad actor isn't attempting to execute
# arbitrary commands or inject false repository names. 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
```

Troubleshooting

Here are a few common problems and some suggested solutions. If you run into any other trouble, you can ask for help or advice in the APIs and Integrations discussions on GitHub Community.

- **Q:** My app isn't pushing code to GitHub. I don't see the fixes that RuboCop automatically makes!

  A: Make sure you have **Read & write** permissions for "Repository contents," and that you are cloning the repository with your installation token. See Step 2.2. Cloning the repository for details.

- **Q:** I see an error in the `template_server.rb` debug output related to cloning my repository.

  A: If you see the following error, you haven't deleted the checkout of the repository in one or both of the `initiate_check_run` or `take_requested_action` methods:

  ```
  2018-11-26 16:55:13 - Git::GitExecuteError - git  clone '--' 'https://x-access-t
  ```

  Compare your code to the `server.rb` file to ensure you have the same code in your `initiate_check_run` and `take_requested_action` methods.

- **Q:** New check runs are not showing up in the "Checks" tab on GitHub.

  A: Restart Smee and re-run your `template_server.rb` server.

- **Q:** I do not see the "Re-run all" button in the "Checks" tab on GitHub.

  A: Restart Smee and re-run your `template_server.rb` server.

Conclusion

After walking through this guide, you've learned the basics of using the Checks API to create a CI server! To review, you:

- Configured your server to receive Checks API events and create check runs.
- Used RuboCop to check code in repositories and create annotations for the errors.
- Implemented a requested action that automatically fixes linter errors.

Next steps

Here are some ideas for what you can do next:

- Currently, the "Fix this" button is always displayed. Update the code you wrote 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, you can update the code to create a pull request with a new branch based on the head branch.