

Continuous Delivery for Android Using GitHub Actions

Learn how to create a continuous delivery pipeline in Android to deploy your apps to the Google Play Store.

One of the principles behind the Agile Manifesto states:

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”

But what does this mean in software engineering terms? Well, picture having a repository where you regularly push code. That code doesn't directly provide value to the customer. Instead, the value comes when you *use* the code to build software that you can deliver to your customer.

To deliver software confidently, you need tests to ensure your changes haven't introduced any issues. When you bring these steps together, you get ***continuous delivery***.

Continuous delivery is the practice of:

1. Regularly merging code.
2. Running tests on the codebase.
3. If those tests pass, building a release version of the software.
4. Delivering the software to the customer, preferably in a staged manner.

In this chapter, you'll learn how to use ***GitHub Actions*** to set up a continuous delivery pipeline that:

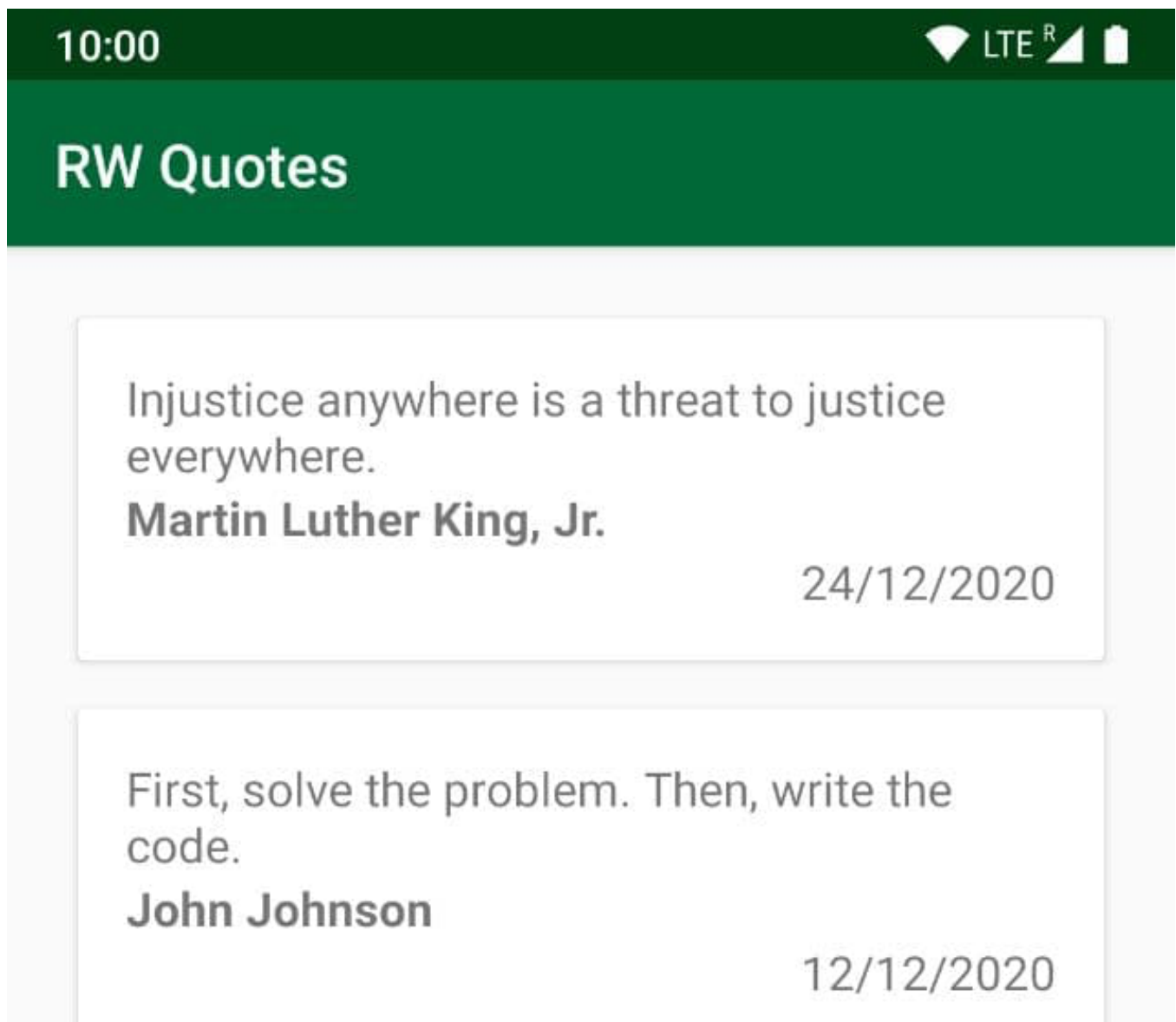
1. Runs tests when you are ready to send a build.

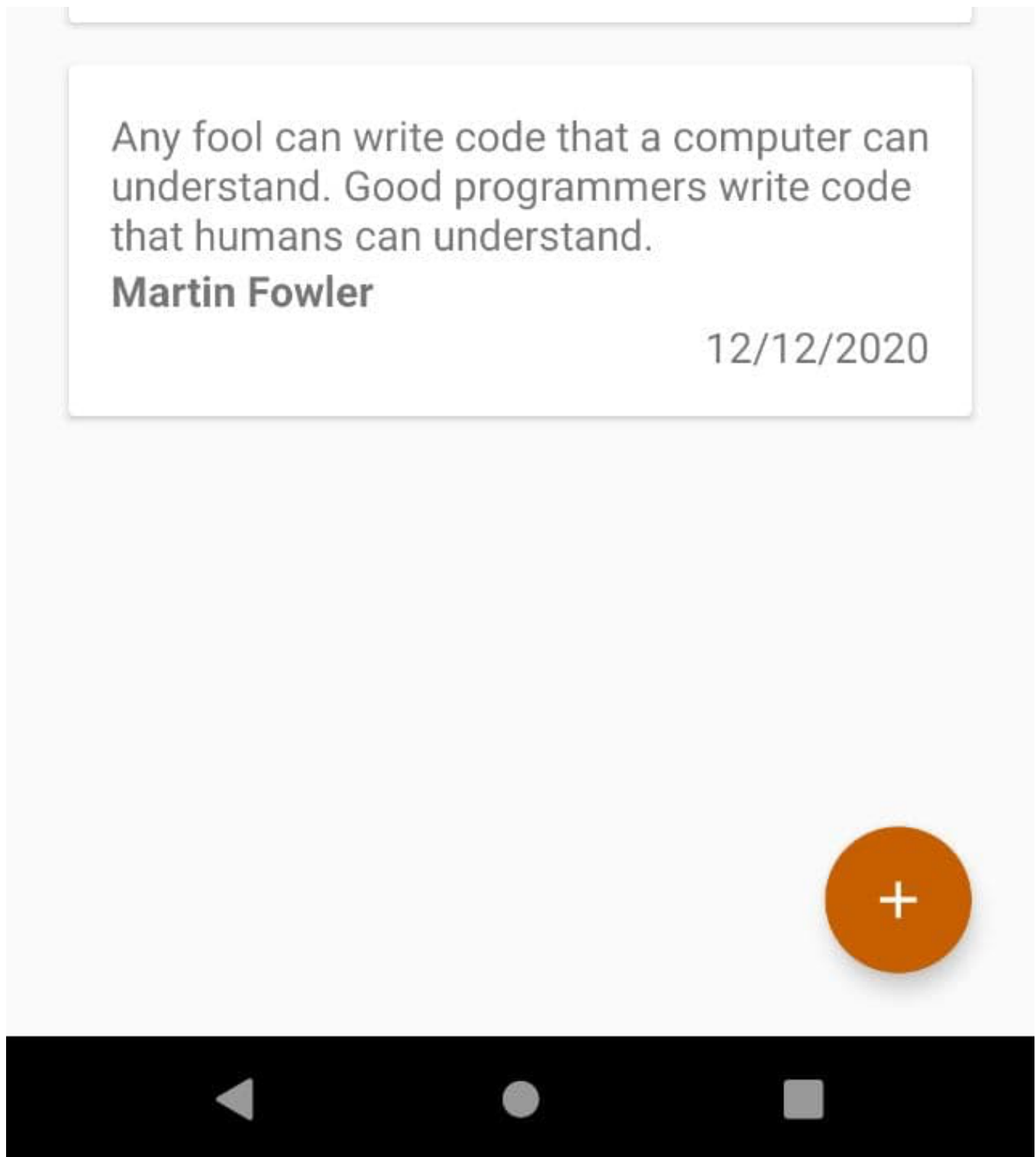
2. Generates a release build, if those tests pass.
3. Pushes the build to Firebase App Distribution to deliver to your Quality Assurance (QA) team.
4. Pushes the build to the Play Store with a rollout percentage after QA approves the changes.

You'll do this while working with the RW Quotes app.

Getting Started

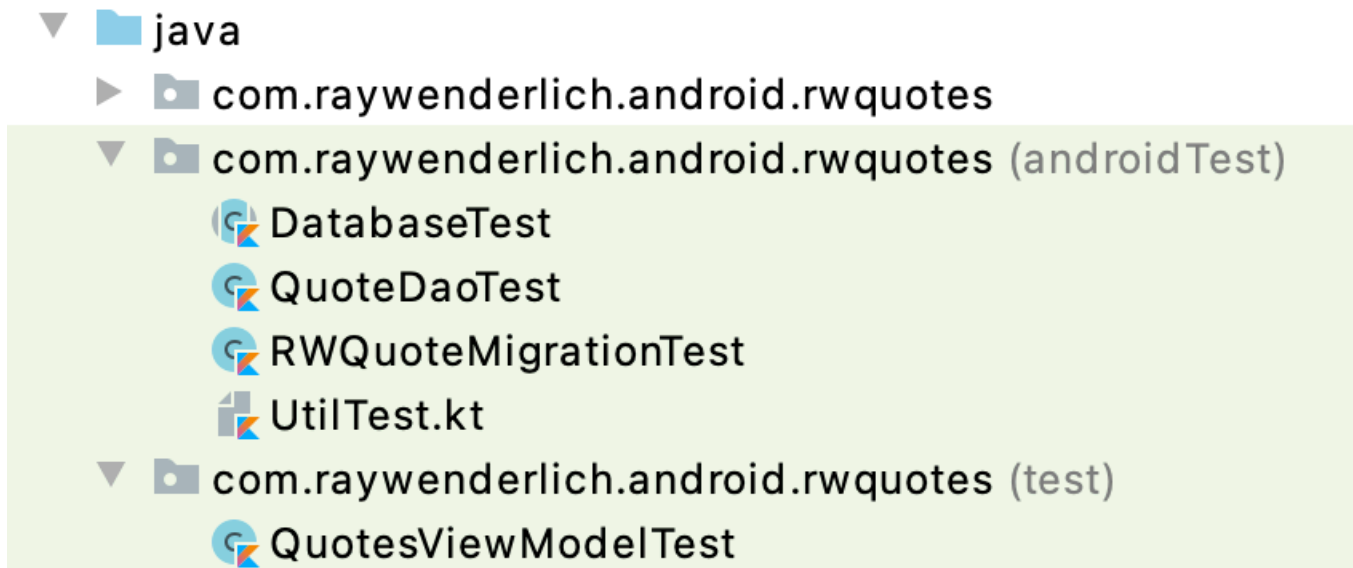
Download the project by clicking the **Download Materials** button at the top or bottom of the tutorial. Open the **starter** project using Android Studio. Build and run and you'll see a screen like the one below:





The app lets you add quotes from different people, then displays those quotes in a list. It also has the option to edit the quotes.

The project contains some unit and instrumentation tests too, as shown in the image below:



For this tutorial, you won't be making any changes to the actual application code, since the app is already finished. Instead, you'll make the changes in an additional file that will control the continuous delivery process.

Using GitHub Actions

GitHub Actions is GitHub's platform for automation workflows. A **workflow** is a sequence of jobs that can run either in series or in parallel. A job usually contains more than one step, where each step is a self-contained function. To learn more about GitHub Actions, go through the tutorial on [Continuous Integration for Android](#).

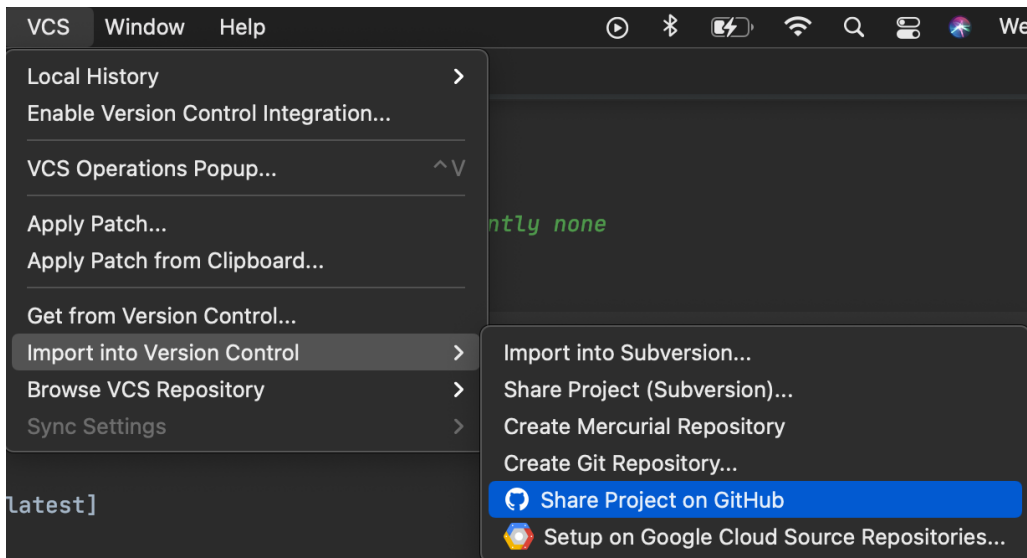
In this tutorial, you'll use multiple first-party as well as third-party actions. To explore the different actions available, visit the [GitHub Marketplace](#).

Uploading the Project to GitHub

The next part of the tutorial will take place mostly on GitHub's website. Therefore, you need to have a GitHub account and to upload the sample project to a repository under your account.

To upload your project from Android Studio, go to **VCS ▶ Import into**

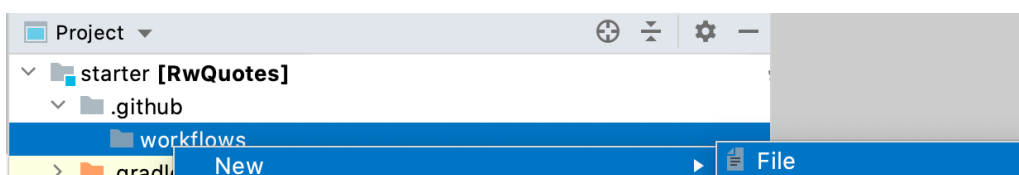
Version Control ▶ Share Project on Github:



Once you've set that up, you can create your first workflow.

Creating a Workflow

To add a new workflow, first, create a new directory with the path **.github/workflows** in the root of your project, either directly in your OS file system or by switching to the Project view in Android Studio. Then, create a file named **check_and_deploy.yml** in the **workflows** directory, as shown in the image below:



All workflows are written in YAML, which is a serialization format commonly used in configuration files. One thing to keep in mind is that proper indentation is extremely important in the YAML format, so make sure to follow the indentation presented in the tutorial exactly for your workflows to work.

Running the Tests

There are two ways you can run tests in your projects:

1. Use the **Run tests** option inside Android Studio.
2. Run the **Gradle** task for the tests from the command line.

To run tests on a remote machine, you have to go with the second option.

Run the following command from the command line to run the unit tests:

```
./gradlew test
```

After the tests run, you'll see a screen like the one below:

```
→ final git:(master) x ./gradlew test
```

```
BUILD SUCCESSFUL in 2s
```

```
49 actionable tasks: 3 executed, 46 up-to-date
```

```
→ final git:(master) x █
```

For the instrumentation tests, you need to either start an emulator or connect a physical device. Once you've done that, run the following command to run the instrumentation tests:

```
./gradlew connectedAndroidTest
```

Once all the tests run, you'll see the following results:

```
→ final git:(master) x ./gradlew connectedAndroidTest
```

```
> Task :app:connectedDebugAndroidTest
```

```
Starting 5 tests on Redmi Note 5 - 10
```

```
BUILD SUCCESSFUL in 10s
```

```
62 actionable tasks: 6 executed, 56 up-to-date
```

Now that you know how to run the tests from the command line, add the following code to ***check_and_deploy.yml***:

```
## 1
name: Test and deploy

## Actions that will be executed when you push code currently
none
on:
  push:

## 2
jobs:
  ## 3
  unit_tests:
    runs-on: [ubuntu-latest]
    steps:
      - uses: actions/checkout@v2

      - name: Unit tests
        run: ./gradlew test

  ## 4
  android_tests:
    runs-on: [ macos-latest ]
    steps:
      - uses: actions/checkout@v2
```

```
- name: Instrumentation Tests
  uses: reactivecircus/android-emulator-runner@v2
  with:
    api-level: 29
    script: ./gradlew connectedAndroidTest
```

The code above does a few things. It:

1. Creates a workflow named ***Test and deploy***.
2. Creates two parallel jobs named ***unit_tests*** and ***android_tests***.
3. The ***unit_tests*** job runs on an ***ubuntu*** runner, which checks out the code and runs the unit tests.
4. The ***android_tests*** job runs on a macOS runner. This job also checks out the code, but runs the instrumentation tests instead. To do this, it uses the ***reactivecircus/android-emulator-runner*** action. The emulator can use hardware acceleration only on the macOS emulator. Therefore, this job needs to run on a macOS runner while others can run on Ubuntu runners.

Next, you'll see how to generate a secure release build on a remote system.

Generating a Signed Release Build

Generating a release build on a remote system is quite different from doing it locally. One of the main aspects is to make sure that the signing secrets remain secret.

Creating a Keystore

To sign your release build, you first need a keystore. If you haven't generated a keystore for your apps yet, follow the tutorial on [Android App Distribution: From Zero to Google Play Store](https://www.raywenderlich.com/19407406-continuous-delivery-for-android-using-github-actions). This guides you through the process of creating a new keystore.

Storing Secrets

For security's sake, it's important not to hard code secrets inside the codebase. A good way to avoid this is by using environment variables to refer to the secrets. GitHub Actions provides a similar mechanism.

Open your repository on GitHub and go to the **Settings** tab. On the left navigation bar, click **Secrets**:

Secrets

[New repository secret](#)

Secrets are environment variables that are **encrypted**. Anyone with **collaborator** access to this repository can use these secrets for Actions.

Secrets are not passed to workflows that are triggered by a pull request from a fork. [Learn more](#).

Environment secrets

There are no secrets for this repository's environments.

Encrypted environment secrets allow you to store sensitive information, such as access tokens, in your repository environments.

[Manage your environments and add environment secrets](#)

Repository secrets

There are no secrets for this repository.

Encrypted secrets allow you to store sensitive information, such as access tokens, in your repository.

Click ***New repository secret*** and add the following four secrets:

1. ***ALIAS***: Alias of your signing key.
2. ***KEY_STORE_PASSWORD***: The password to your signing keystore.
3. ***KEY_PASSWORD***: The private key password for your signing keystore.
4. ***SIGNING_KEY***: The base 64-encoded signing key used to sign your app.

To generate the base 64-encoded key, run the following command in Terminal and copy the output string:

```
openssl base64 < path_to_signing_key | tr -d '\n' | tee
some_signing_key.jks.base64.txt
```

In the code above, replace ***path_to_signing_key*** with the actual path to your keystore.

Signing the Build

Add a new job named `build` to the workflow, indented below the `jobs` tag as shown below:

```
jobs:
  build:
    needs: [ unit_tests, android_tests ]
    runs-on: ubuntu-latest
    steps:
      # 1
      - name: Checkout code
        uses: actions/checkout@v2
      # 2
      - name: Generate Release APK
        run: ./gradlew assembleRelease
      # 3
      - name: Sign APK
        uses: r0adkll/sign-android-release@v1
        # ID used to access action output
        id: sign_app
        with:
          releaseDirectory: app/build/outputs/apk/release
          signingKeyBase64: ${ secrets.SIGNING_KEY }
          alias: ${ secrets.ALIAS }
          keyStorePassword: ${ secrets.KEY_STORE_PASSWORD }
```

```
        keyPassword: ${ secrets.KEY_PASSWORD }}
# 4
- uses: actions/upload-artifact@master
  with:
    name: release.apk
    path: ${ steps.sign_app.outputs.signedReleaseFile }}
# 5
- uses: actions/upload-artifact@master
  with:
    name: mapping.txt
    path: app/build/outputs/mapping/release/mapping.txt
```

In the code above, the build job performs multiple steps. It:

1. Checks out the code.
2. Generates a release APK using the `assembleRelease` Gradle task.
3. Signs the APK using the ***roadkill/sign-android-release*** action, which is a third party action available on the github marketplace linked earlier. This step uses the four secrets you added in the previous section. It also has an ID: ***sign_app***.
4. Uploads the signed APK as an artifact to GitHub. This step uses the ID from the previous step to access its output, named `signedReleaseFile`.
5. Uploads the mapping file as an artifact. You'll use this in a later step, when you upload to the Play Store.

Commit the file to your project and push it to GitHub. Open the GitHub repository and go to the **Actions** tab. You'll see that a workflow named ***Test and deploy*** is running. Wait for a few minutes and the workflow should complete successfully:

Note: If you're using a Windows machine, you may need to change the `gradlew` file permissions such that they can be executed. If your action fails with a permission error, run the following command to set the `gradlew` file

as executable: `git update-index --chmod=+x gradlew`

✓ **Update workflow**





Test and deploy #2: Commit 098406f pushed by SubhrajyotiSen

master

📅 30 minutes ago ...

🕒 9m 0s

Also, notice that the signed APK and mapping file are attached as artifacts, similar to the ones shown below:

Artifacts		
Produced during runtime		
Name	Size	
 mapping.txt	998 KB	
 release.apk	1.58 MB	

Congratulations, you've completed the first part of your continuous delivery pipeline.

Triggering a Release

In the current implementation, the workflow runs every time you push code to your repository. But, you don't want to release a new build every time you push a new commit. Ideally, you want to release a build in the following scenarios:

- You push a version tag to the repository.
- You create a pull request targeting the master branch.

In this section, you'll modify the workflow so it triggers when those conditions occur.

Pushing a Version Tag

A version tag name usually has the **v** prefix. For example, v1.10, v0.31,

v/3.31 etc. You'll use this convention to trigger the workflow when you push a version tag.

Add the following code under `push` in the workflow:

```
tags:
  - 'v*'
```

Here, `v*` is a regular expression that matches any string starting with `v`.

Your workflow's `on` condition will now be:

```
on:
  push:
    tags:
      - 'v*'
```

Commit your changes and push them to GitHub.

Create a new release tag and push it by running the following commands in Terminal:

```
git tag v0.1 -a -m "Release v0.1"
git push --follow-tags
```

The code above creates a tag named **v0.1** and pushes it with the commit message **Release v0.1**.

Once the push is complete, open the **Actions** tab. You'll see that the workflow is running. To verify that the workflow triggers only on the version tag, push an empty commit and check if that triggers the workflow. You can do so by running the following commands from the command line:

```
git commit --allow-empty -m "Empty commit"  
git push
```

In the code above, `--allow-empty` lets you create an empty commit — that is, a commit without any changes.

Open the **Actions** tab in the repository and verify that the workflow hasn't triggered.

Pull Request to Master

A typical workflow among teams is to send a release APK to the QA team whenever a pull request is made to the **master** branch. Your next step is to make this happen automatically.

Add the following code to the **on** section of the workflow:

```
pull_request:  
  branches:  
    - master
```

This code triggers the workflow when a pull that targets the master branch is created.

Commit and push the changes to GitHub, then verify the changes by creating a pull request from any branch to the master branch.

Preventing a Merge if Tests Fail

Before merging any code to master, you want to make sure that the new code builds correctly and all tests pass. If any of these conditions fail, you want to block the pull request from merging. **Branch protection** helps you do this.

Add protection by going to **Settings** ▶ **Branches** on the repository. You'll see a page like the one shown below:

Default branch

The default branch is considered the "base" branch in your repository, against which all pull requests and code commits are automatically made, unless you specify a different branch.

master ▼

Update

Branch protection rules

Add rule

Define branch protection rules to disable force pushing, prevent branches from being deleted, and optionally require status checks before merging. New to branch protection rules? [Learn more.](#)

No branch protection rules defined yet.

Click **Add rule** in the **Branch protection rules** section. You'll see a list of options, each with a checkbox. Check **Require status checks to pass before merging**. Since you want both the build job and the test jobs to pass, you have to check the **build**, **unit_test** and **android_test** tasks.

Finally, click **Create** at the bottom of the page. From now on, any pull request that fails these checks will be unable to merge.

Deploying to Firebase App Distribution

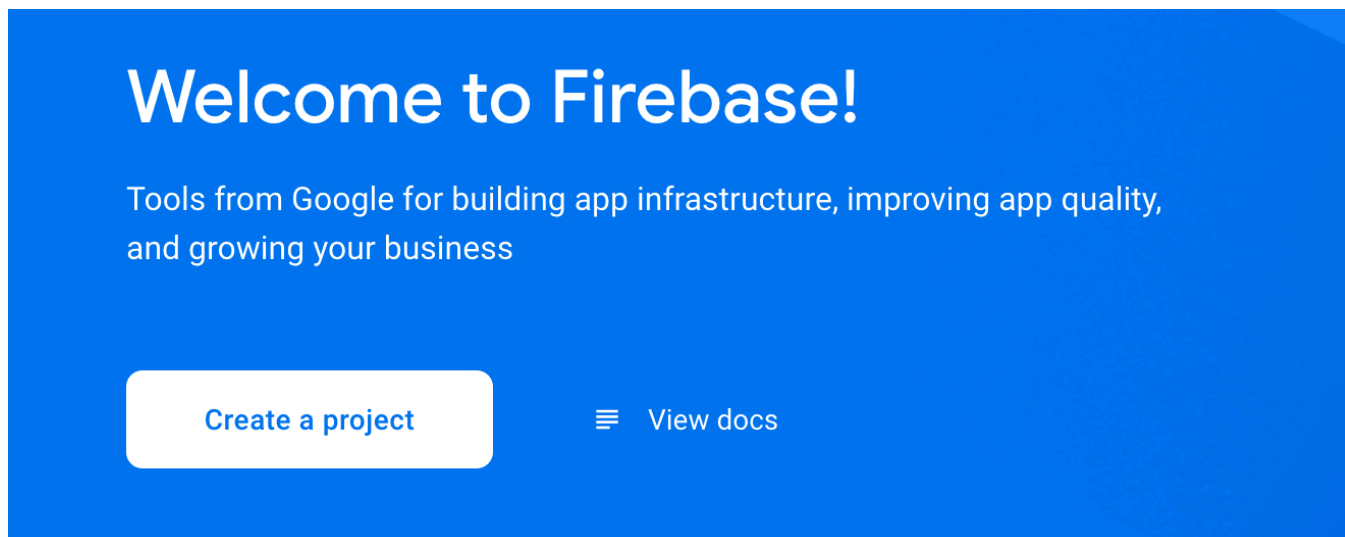
Once the unit and instrumentation tests pass, you want to send the build to your QA team. A structured way of doing this is to use Firebase App Distribution. It lets you keep track of the uploaded builds and notify teams when new builds become available. You can even create groups and distribute a build only to specific groups.

Next, you'll see how to set this up for your project.


Setting up Firebase

First, you need to create a new project on Firebase. Visit

<https://console.firebase.google.com> and click **Create a project**.




Next, choose a project name. For this tutorial, name your project **RW Quotes**, then click **Continue**.

 Create a project (Step 1 of 3)

Let's start with a name for your project[?]

Project name

RW Quotes

 rw-quotes-1daa4

Continue

You don't need Google Analytics for this project, so disable it. Click **Create project**.

× Create a project (Step 2 of 2)

Google Analytics for your Firebase project

Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, Predictions, and Cloud Functions.

Google Analytics enables:

× A/B testing ?

× User segmentation & targeting across Firebase products ?

× Predicting user behavior ?

× Crash-free users ?

× Event-based Cloud Functions triggers ?

× Free unlimited reporting ?

☐

Enable Google Analytics for this project
Recommended

[Previous](#)

Create project

Next, you need to add an Android app to the project. To do this, click the **Android** icon:

Get started by adding Firebase to your app



Add an app to get started

You'll get a short form asking for the package name. The package name of the app is ***com.raywenderlich.android.rwquotes***. Enter the package name and click ***Register app***.

Because you're only using App Distribution, skip the next two sections of the form by clicking ***Next***. Finally, click ***Continue to console*** to return to the home dashboard.

Adding Testers

From the left navigation bar, select ***App Distribution*** in the ***Release & Monitor*** section.

Accept the terms and conditions and click ***Get started***. You'll see the App Distribution dashboard, which contains three tabs: ***Releases***, ***Invite links*** and ***Testers and Groups***. Go to the ***Testers and Groups*** tab.

In the ***Add testers*** input field, enter your email address. Firebase will then send you an invitation to become a tester for the app. Similarly, add the email addresses of the rest of your development and QA teams.

Next, you'll create a group. A group is a collection of testers you want to send a specific build to. To create a group, click **Add group** and enter a name for your group — in this case, QA. Now, you can choose the email addresses of all members of your QA team and add them here.

Fetching the App ID and Token

To give GitHub Actions access to your Firebase account, you'll need two things:

1. **App ID**: An ID specific to your project.
2. **Token**: An authentication token to access Firebase.

Click the **gear** icon at the top of the left navigation bar and select **Project Settings** from the menu. Scroll down until you see the package name of the app. Next to it, you'll see a section named **App ID** with a long alphanumeric string. Copy this string and add it as a **Secret** on the GitHub repository named **FIREBASE_APP_ID**.

Fetching the token is a bit more tricky. You need to install Firebase CLI first. To do so, follow the instructions at <https://firebase.google.com/docs/cli> to install the tool based on your operating system.

Once installed, run the following command from the command line:

```
firebase login:ci
```

This will open your browser and ask you to sign in using a Google account. If you have multiple accounts, choose the one you used to set up Firebase.

Once you've signed in, you have to grant Firebase CLI some permissions. Even though these are simple permissions, you should read through them.

Once you've read and granted the permissions, you'll see a message stating that you can close the browser tab. Go back to the command line you'll see some text:

```
→ final git:(master) firebase login:ci
```

Visit this URL on this device to log in:

https://accounts.google.com/o/oauth2/auth?client_id=563584335869-fgrhgm47bqnekij5i8b5pr03ho849e6.apps.googleapis.com%2Fauth%2Ffirebase%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fcloud-platform&response_type=code&scope=openid%20profile%20email%20https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fcloud-platform

Waiting for authentication...

✓ Success! Use this token to login on a CI server:

```
1//0gXeFZhojZ8
```

Example: `firebase deploy --token "$FIREBASE_TOKEN"`

Copy the token and add it as a GitHub **Secret** with the name ***FIREBASE_TOKEN***.

Upload to Firebase Action

Now that you've set up Firebase and stored the secrets, you need to add the job to upload the app to Firebase App Distribution. For this, you'll use the ***wzieba/Firebase-Distribution-Github-Action*** action.

Add the following code to the workflow:

```
deploy-firebase:
  # 1
  needs: [ build ]
  runs-on: ubuntu-latest
  steps:
    # 2
    - uses: actions/download-artifact@master
      with:
        name: release.apk
    #3
    - name: upload artifact to Firebase App Distribution
```

```
uses: wzieba/Firebase-Distribution-Github-Action@v1
with:
  appId: ${{secrets.FIREBASE_APP_ID}}
  token: ${{secrets.FIREBASE_TOKEN}}
  groups: QA
  file: app-release-unsigned-signed.apk
```

The code above:

1. Uses ***needs*** to specify that the job can only run if the ***build*** job has completed successfully.
2. Uses the ***actions/download-artifact*** action to download the artifact of the release APK.
3. Uploads the downloaded artifact to Firebase App Distribution and makes it available to the ***QA*** group you created earlier. It also uses the two secrets named ***FIREBASE_APP_ID*** and ***FIREBASE_TOKEN***, which you added in the previous section.

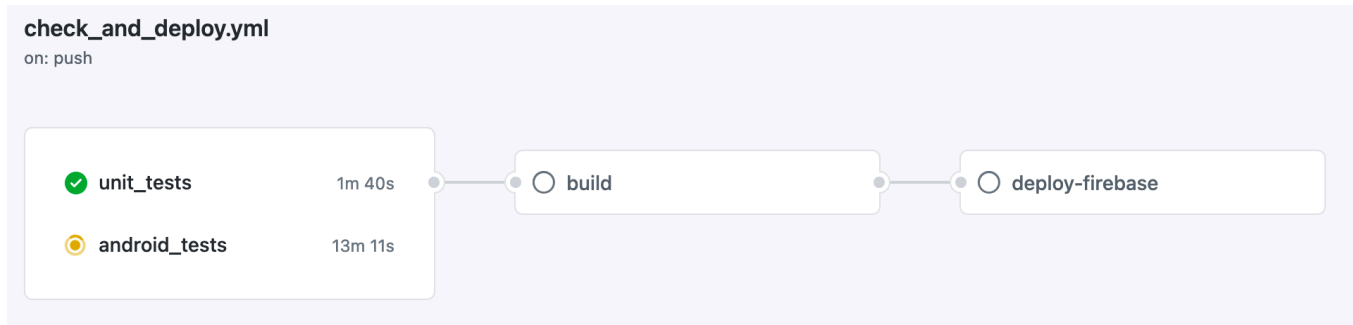
Commit and push the changes to GitHub. Next, create a new tag and push it. Once you push the tag, visit the ***Actions*** tab in the repository. You'll see that the workflow is running.

Once the workflow has successfully finished running, verify that it uploaded the build by visiting the ***Releases*** tab on the ***App Distribution*** page on Firebase.

Congratulations, you've successfully set up a functional continuous delivery pipeline. The only thing left to add is the ability to release your app to the Play Store.

Visualizing a Workflow

On the ***Actions*** tab in the repository, select any workflow and you'll see a page like this:



In the image above, there are three boxes containing:

1. ***unit_tests*** and ***android_tests***
2. The ***build*** job
3. The ***deploy_firebase*** job

Horizontal lines connect the boxes. The box signifies that ***unit_tests*** and ***android_tests*** run in parallel. Only after both of them have completed will the build job run. Similarly, the ***deploy_firebase*** job will run only after the build job completes.

Deploying to Play Store

The action needed to deploy a build to the Play Store is similar to the one you used for app distribution but the setup is a bit more complex. It requires you to use a service account. Don't worry if you haven't heard of the term, you'll create one shortly.

Before you can publish to the Play Store, you need to add a new app to the Google Play Console. To do this, follow the tutorial on [Android App Distribution: From Zero to Google Play Store](#).

Setting up a Service Account

Once you've completed the setup, you need to create the service account. Visit <https://play.google.com/console> and go to **Settings** ▶ **Developer account** ▶ **API access**. Once there, click the **Create new project** radio button and select **Link Project** in the bottom right corner of

the page.

Linked Google Cloud project

- ☐ Link existing project
- ☒ Create new project

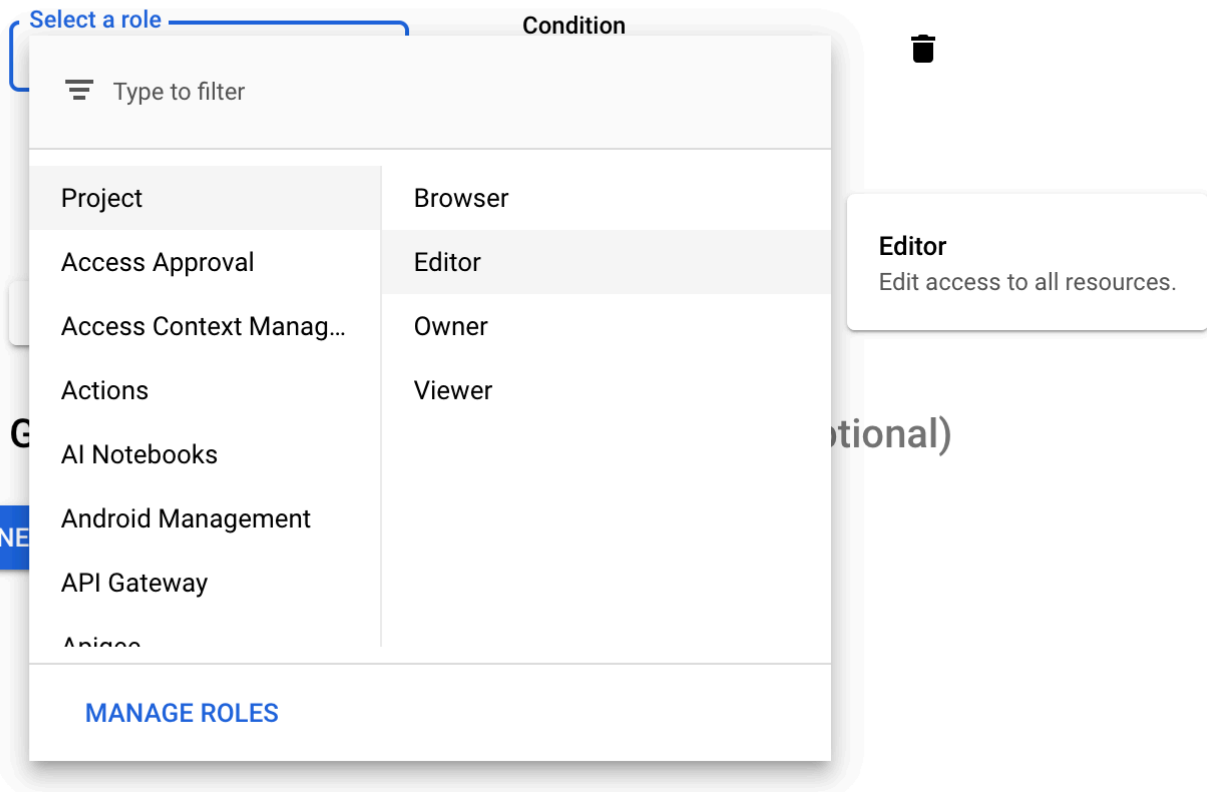
In the **Service accounts** section, click **Create new service account**. Follow the instructions on the pop-up and navigate to the google cloud platform page.

Once there, click the **Create Service Account** button on the top of the page.

Give the service account any name you want. Next, assign the service account the role of **Editor**, as shown below:

2 Grant this service account access to project (optional)

Grant this service account access to Google Play Console Developer so that it has permission to complete specific actions on the resources in your project. [Learn more](#)



After assigning a role, click **Done** to return to the Service Accounts page. Click the three-dot menu next to the service account you just created, select **Create key** and choose the **JSON** option. The key will generate, then download to your computer.

Next, visit [Google Developers Console API Library](#) and search for **Google Play Android Developer API**. From the search result, select the API then click **Enable**, as shown below:



Google Play Android Developer API

Google

Manage your app in the Google Play Store

ENABLE

TRY THIS API [↗](#)

The Google Play Android Developer API lets the service account perform automated operations on the Google Play Console.

Go back to the Play Console tab on your browser and click **Done** on the pop-up. The Service account section will refresh and the account you just created will appear.

You need to grant access to the service account to manage releases. To do this, click **Grant access**, which will open the **Invite user** page. Scroll down and choose the **Account permissions** tab. In the **Releases** section, select **Release to production, exclude devices, and use Play App Signing** and **Release apps to testing tracks**. Finally, click **Invite User**.

Next, go to the **App permissions** tab on the same page and grant access to the app.

Open the service account key file you downloaded and add its contents as a GitHub Secret named **SERVICE_ACCOUNT_JSON**.

The service is now ready.

Upload the Google Play Action

To upload a build to the Play Store Console, use the **r0adkill/upload-google-play** action. You want to deploy to the Play Store only when you

merge a pull request to the master branch. To do this, create a new file named ***play_store_workflow.yml*** in ***.github/workflows*** and add the following to it:

```
name: Deploy to Play Store
```

```
on:
```

```
  push:
```

```
    branches:
```

```
      - master
```

```
jobs:
```

```
  build:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - name: Checkout code
```

```
        uses: actions/checkout@v2
```

```
      - name: Generate Release APK
```

```
        run: ./gradlew assembleRelease
```

```
      - name: Sign APK
```

```
        uses: r0adkll/sign-android-release@v1
```

```
        # ID used to access action output
```

```
        id: sign_app
```

```
        with:
```

```
          releaseDirectory: app/build/outputs/apk/release
```

```
          signingKeyBase64: ${ secrets.SIGNING_KEY }
```

```
          alias: ${ secrets.ALIAS }
```

```
          keyStorePassword: ${ secrets.KEY_STORE_PASSWORD }
```

```
          keyPassword: ${ secrets.KEY_PASSWORD }
```

```
      - uses: actions/upload-artifact@master
```

```
        with:
```

```
          name: release.apk
```

```
          path: ${ steps.sign_app.outputs.signedReleaseFile }
```

```
      - uses: actions/upload-artifact@master
```

```
        with:
```

```
          name: mapping.txt
```

```
          path: app/build/outputs/mapping/release/mapping.txt
```

```
deploy-play-store:
  needs: [build]
  runs-on: ubuntu-latest
  steps:
    # 1
    - uses: actions/download-artifact@master
      with:
        name: release.apk
    - uses: actions/download-artifact@master
      with:
        name: mapping.txt
    # 2
    - name: Publish to Play Store internal test track
      uses: r0adkll/upload-google-play@v1
      with:
        serviceAccountJsonPlainText: ${
secrets.SERVICE_ACCOUNT_JSON }
    # 3
    packageName: com.raywenderlich.android.rwquotes
    releaseFile: app-release-unsigned-signed.apk
    track: internal
    userFraction: 0.50
    mappingFile: mapping.txt
```

The workflow above triggers when you push any change to the master branch. The build job is the same as the one you used in the previous workflow.

The ***deploy-play-store*** job does the following:

1. Downloads the release APK and mapping file using the ***actions/download-artifact*** action.
2. Uploads the APK to the ***internal*** track and makes it available to 50% percent of the users on that track.
3. Specifies the package name of the app and also the APK and

mapping file to upload.

Commit this file and push it to GitHub. To verify the workflow, push an empty commit to the master branch. Once the workflow is complete, visit the **Internal** track on the Google Play Console. You'll see that a build of the app has published, as shown below:

1.1

 Available to 50% of internal testers · Last updated Dec 27 5:01 PM · Available on 13,668 devices

[View release details](#) Manage rollout ▼ Promote release ▼

If you get an error stating that the package name is already in use on the Play Store, open **app/build.gradle** and change the `applicationId` to something specific to your project. Additionally, remember to register a new app on Firebase App Distribution and update the package name in the action as well.

Congratulations, you've successfully built a complete continuous delivery pipeline using GitHub Actions.

Where to Go From Here?

You can download the completed project files by clicking the **Download Materials** button at the top or bottom of the tutorial.

There are many variations of the continuous delivery process, which lets you modify it to adapt to your team's processes. For example, instead of deploying to the internal track when changes are pushed to GitHub, you could deploy it to the production track.

You can also try creating a workflow that extracts the changelog from a commit message and attaches it while deploying the build. Use the changelog either in the **Release notes** section on Firebase App

Distribution or the ***What's new*** section of the Play Console releases.

The possibilities with GitHub Actions are vast and you can continue tweaking the workflows till you get something that completely automates your actual workflow of testing and deploying.

If you have any questions or comments, please join the forum discussion below.

[Download Materials](#)

raywenderlich.com Weekly

The raywenderlich.com newsletter is the easiest way to stay up-to-date on everything you need to know as a mobile developer.

Get a weekly digest of our tutorials and courses, and receive a free in-depth email course as a bonus!

Add a rating for this content