Creating GitHub CLI extensions

In this article

Learn how to share new GitHub CLI commands with other users by creating custom extensions for GitHub CLI.

About GitHub CLI extensions

GitHub CLI extensions are custom GitHub CLI commands that anyone can create and use. For more information about how to use GitHub CLI extensions, see "Using GitHub CLI extensions."

You need a repository for each extension that you create. The repository name must start with `gh-`. The rest of the repository name is the name of the extension. The repository must have an executable file at its root with the same name as the repository or a set of precompiled binary executables attached to a release.

Note: When relying on an executable script, we recommend using a bash script because bash is a widely available interpreter. You may use non-bash scripts, but the user must have the necessary interpreter installed in order to use the extension. If you would prefer to not rely on users having interpreters installed, consider a precompiled extension.

Creating an interpreted extension with `gh extension create`

Note: Running `gh extension create` with no arguments will start an interactive wizard.

You can use the `gh extension create` command to create a project for your extension, including a bash script that contains some starter code.

1. Set up a new extension by using the `gh extension create` subcommand. Replace `EXTENSION-NAME` with the name of your extension.

   ```
   gh extension create EXTENSION-NAME
   ```

2. Follow the printed instructions to finalize and optionally publish your extension.

Creating a precompiled extension in Go with `gh extension create`

You can use the `--precompiled=go` argument to create a Go-based project for your extension, including Go scaffolding, workflow scaffolding, and starter code.

1. Set up a new extension by using the `gh extension create` subcommand. Replace `EXTENSION-NAME` with the name of your extension and specify `--precompiled=go`.

   ```
   gh extension create --precompiled=go EXTENSION-NAME
   ```

2. Follow the printed instructions to finalize and optionally publish your extension.

Creating a non-Go precompiled extension with `gh extension create`

You can use the `--precompiled=other` argument to create a project for your non-Go precompiled extension, including workflow scaffolding.

1. Set up a new extension by using the `gh extension create` subcommand. Replace `EXTENSION-NAME` with the name of your extension and specify `--precompiled=other`.

   ```
   gh extension create --precompiled=other EXTENSION-NAME
   ```

2. Add some initial code for your extension in your compiled language of choice.
3. Fill in `script/build.sh` with code to build your extension to ensure that your extension can be built automatically.
4. Follow the printed instructions to finalize and optionally publish your extension.

Creating an interpreted extension manually

1. Create a local directory called `gh-EXTENSION-NAME` for your extension. Replace `EXTENSION-NAME` with the name of your extension. For example, `gh-whoami`.
2. In the directory that you created, add an executable file with the same name as the directory.

   > **Note:** Make sure that your file is executable. On Unix, you can execute `chmod +x file_name` in the command line to make `file_name` executable. On Windows, you can run `git init -b main`, `git add file_name`, then `git update-index --chmod=+x file_name`.

3. Write your script in the executable file. For example:

   ```
   #!/usr/bin/env bash
   set -e
   exec gh api user --jq '"You are @\(.login) (\(.name))."'
   ```

4. From your directory, install the extension as a local extension.

```
gh extension install .
```

5. Verify that your extension works. Replace `EXTENSION-NAME` with the name of your extension. For example, `whoami`.

```
gh EXTENSION-NAME
```

6. From your directory, create a repository to publish your extension. Replace `EXTENSION-NAME` with the name of your extension.

```
git init -b main
git add . && git commit -m "initial commit"
gh repo create gh-EXTENSION-NAME --source=. --public --push
```

7. Optionally, to help other users discover your extension, add the repository topic `gh-extension`. This will make the extension appear on the `gh-extension` topic page. For more information about how to add a repository topic, see "Classifying your repository with topics."

Tips for writing interpreted GitHub CLI extensions

Handling arguments and flags

All command line arguments following a `gh my-extension-name` command will be passed to the extension script. In a bash script, you can reference arguments with `$1`, `$2`, etc. You can use arguments to take user input or to modify the behavior of the script.

For example, this script handles multiple flags. When the script is called with the `-h` or `--help` flag, the script prints help text instead of continuing execution. When the script is called with the `--name` flag, the script sets the next value after the flag to `name_arg`. When the script is called with the `--verbose` flag, the script prints a different greeting.

```
#!/usr/bin/env bash
set -e

verbose=""
name_arg=""
while [ $# -gt 0 ]; do
  case "$1" in
  --verbose)
    verbose=1
    ;;
  --name)
    name_arg="$2"
    shift
    ;;
  -h|--help)
    echo "Add help text here."
    exit 0
    ;;
  esac
  shift
done
```

```
if [ -z "$name_arg" ]
then
  echo "You haven't told us your name."
elif [ -z "$verbose" ]
then
  echo "Hi $name_arg"
else
  echo "Hello and welcome, $name_arg"
fi
```

Calling core commands in non-interactive mode

Some GitHub CLI core commands will prompt the user for input. When scripting with those commands, a prompt is often undesirable. To avoid prompting, supply the necessary information explicitly via arguments.

For example, to create an issue programmatically, specify the title and body:

```
gh issue create --title "My Title" --body "Issue description"
```

Fetching data programmatically

Many core commands support the `--json` flag for fetching data programmatically. For example, to return a JSON object listing the number, title, and mergeability status of pull requests:

```
gh pr list --json number,title,mergeStateStatus
```

If there is not a core command to fetch specific data from GitHub, you can use the `gh api` command to access the GitHub API. For example, to fetch information about the current user:

```
gh api user
```

All commands that output JSON data also have options to filter that data into something more immediately usable by scripts. For example, to get the current user's name:

```
gh api user --jq '.name'
```

For more information, see `gh help formatting`.

Creating a precompiled extension manually

1. Create a local directory called `gh-EXTENSION-NAME` for your extension. Replace `EXTENSION-NAME` with the name of your extension. For example, `gh-whoami`.
2. In the directory you created, add some source code. For example:

   ```
   package main
   import (
     "github.com/cli/go-gh"
     "fmt"
   )

   func main() {
     args := []string{"api", "user", "--jq", `"You are @\(.login) (\(.name))"`
   }
   ```

```
  stdOut, _, err := gh.Exec(args...)
  if err != nil {
    fmt.Println(err)
    return
  }
  fmt.Println(stdOut.String())
}
```

3. From your directory, install the extension as a local extension.

```
gh extension install .
```

4. Build your code. For example, with Go, replacing `YOUR-USERNAME` with your GitHub username:

```
go mod init github.com/YOUR-USERNAME/gh-whoami
go mod tidy
go build
```

5. Verify that your extension works. Replace `EXTENSION-NAME` with the name of your extension. For example, `whoami`.

```
gh EXTENSION-NAME
```

6. From your directory, create a repository to publish your extension. Replace `EXTENSION-NAME` with the name of your extension.

> **Note:** Be careful not to commit the binary produced by your compilation step to version control.

```
 git init -b main
echo "gh-EXTENSION-NAME" >> .gitignore
git add main.go go.* .gitignore && git commit -m 'Initial commit'
gh repo create "gh-EXTENSION-NAME"
```

7. Create a release to share your precompiled extension with others. Compile for each platform you want to support, attaching each binary to a release as an asset. Binary executables attached to releases must follow a naming convention and have a suffix of OS-ARCHITECTURE[EXTENSION].

   For example, an extension named `whoami` compiled for Windows 64bit would have the name `gh-whoami-windows-amd64.exe` while the same extension compiled for Linux 32bit would have the name `gh-whoami-linux-386`. To see an exhaustive list of OS and architecture combinations recognized by `gh`, see this source code.

> **Note:** For your extension to run properly on Windows, its asset file must have a `.exe` extension. No extension is needed for other operating systems.

   Releases can be created from the command line. For example:

```
git tag v1.0.0
git push origin v1.0.0
GOOS=windows GOARCH=amd64 go build -o gh-EXTENSION-NAME-windows-amd64.exe
GOOS=linux GOARCH=amd64 go build -o gh-EXTENSION-NAME-linux-amd64
GOOS=darwin GOARCH=amd64 go build -o gh-EXTENSION-NAME-darwin-amd64
```

```
gh release create v1.0.0 ./*amd64*
```

8. Optionally, to help other users discover your extension, add the repository topic `gh-extension`. This will make the extension appear on the `gh-extension` topic page. For more information about how to add a repository topic, see "Classifying your repository with topics."

Tips for writing precompiled GitHub CLI extensions

Automating releases

Consider adding the gh-extension-precompile action to a workflow in your project. This action will automatically produce cross-compiled Go binaries for your extension and supplies build scaffolding for non-Go precompiled extensions.

Using GitHub CLI features from Go-based extensions

Consider using go-gh, a Go library that exposes pieces of `gh` functionality for use in extensions.

Next steps

To see more examples of GitHub CLI extensions, look at repositories with the `gh-extension` topic.