Preparing your code for CodeQL analysis

In this article

You can build a CodeQL database containing the data needed to analyze your code.

GitHub CodeQL is licensed on a per-user basis upon installation. You can use CodeQL only for certain tasks under the license restrictions. For more information, see "About the CodeQL CLI." If you have a GitHub Advanced Security license, you can use CodeQL for automated analysis, continuous integration, and continuous delivery. For more information, see "About GitHub Advanced Security."

About preparing your code for analysis

Before you analyze your code using CodeQL, you need to create a CodeQL database containing all the data required to run queries on your code. You can create CodeQL databases yourself using the CodeQL CLI.

CodeQL analysis relies on extracting relational data from your code, and using it to build a CodeQL database. CodeQL databases contain all of the important information about a codebase, which can be analyzed by executing CodeQL queries against it.

Before you generate a CodeQL database, you need to:

1. Install and set up the CodeQL CLI. For more information, see "Setting up the CodeQL CLI."
2. Check out the code that you want to analyze:
   - For a branch, check out the head of the branch that you want to analyze.
   - For a pull request, check out either the head commit of the pull request, or check out a GitHub-generated merge commit of the pull request.
3. Set up the environment for the codebase, making sure that any dependencies are available. For more information, see "Creating databases for non-compiled languages" and "Creating databases for compiled languages."
4. Find the build command, if any, for the codebase. Typically this is available in a configuration file in the CI system.

Once the codebase is ready, you can run `codeql database create` to create the database.

Running `codeql database create`

CodeQL databases are created by running the following command from the checkout root of your project:

```
codeql database create <database> --language=<language-identifier>
```

You must specify:

- `<database>`: a path to the new database to be created. This directory will be created when you execute the command—you cannot specify an existing directory.
- `--language`: the identifier for the language to create a database for. When used with `--db-cluster`, the option accepts a comma-separated list, or can be specified more than once. CodeQL supports creating databases for the following languages:

Language	Identifier	Optional alternative identifiers (if any)
C/C++	`c-cpp`	`c` or `cpp`
C#	`csharp`	
Go	`go`	
Java/Kotlin	`java-kotlin`	`java` or `kotlin`
JavaScript/TypeScript	`javascript-typescript`	`javascript` or `typescript`
Python	`python`	
Ruby	`ruby`	
Swift	`swift`	

Note: If you specify one of the alternative identifiers, this is equivalent to using the standard language identifier. For example, specifying `javascript` instead of `javascript-typescript` will not exclude analysis of TypeScript code. You can do this in an advanced setup workflow with the `--paths-ignore` option. For more information, see "Customizing your advanced setup for code scanning."

Note: CodeQL analysis for Kotlin and Swift is currently in beta. During the beta, analysis of Kotlin and Swift code, and the accompanying documentation, will not be as comprehensive as for other languages. Additionally, Swift 5.8 is not yet supported.

If your codebase has a build command or script that invokes the build process, we recommend that you specify it as well:

```
codeql database create <database> --command <build> \
      --language=<language-identifier>
```

You can specify additional options depending on the location of your source file, if the code needs to be compiled, and if you want to create CodeQL databases for more than one language.

Option	Required	Usage
`<database>`	✓	Specify the name and location of a directory to create for the CodeQL database. The command will fail if you try to overwrite an existing directory. If you also specify `--db-cluster`, this is the parent directory and a subdirectory is created for each language analyzed.
`--language`	✓	Specify the identifier for the language to create a database for, one of: `c-cpp`, `csharp`, `go`, `java-kotlin`, `javascript-typescript`, `python`, `ruby`, and `swift`. When used with `--db-cluster`, the option accepts a comma-separated list, or can be specified more than once.
`--command`	✗	**Recommended.** Use to specify the build command or script that invokes the build process for the codebase. Commands are run from the current folder or, where it is defined, from `--source-root`. Not needed for Python and JavaScript/TypeScript analysis.
`--db-cluster`	✗	Use in multi-language codebases to generate one database for each language specified by `--language`.
`--no-run-unnecessary-builds`	✗	**Recommended.** Use to suppress the build command for languages where the CodeQL CLI does not need to monitor the build (for example, Python and JavaScript/TypeScript).
`--source-root`	✗	Use if you run the CLI outside the checkout root of the repository. By default, the `database create` command assumes that the current directory is the root directory for the source files, use this option to specify a different location.
`--codescanning-config`	✗	Advanced. Use if you have a configuration file that specifies how to create the CodeQL databases and what queries to run in later steps. For more information, see "Customizing your advanced setup for code scanning" and "database create."

You can specify extractor options to customize the behavior of extractors that create CodeQL databases. For more information, see "Extractor options."

For full details of all the options you can use when creating databases, see "database create."

Single language example

This example creates a single CodeQL database for the repository checked out at `/checkouts/example-repo`. It uses the JavaScript extractor to create a hierarchical representation of the JavaScript and TypeScript code in the repository. The resulting database is stored in `/codeql-dbs/example-repo`.

```
$ codeql database create /codeql-dbs/example-repo --language=javascript-
typescript \
    --source-root /checkouts/example-repo

> Initializing database at /codeql-dbs/example-repo.
> Running command [/codeql-home/codeql/javascript/tools/autobuild.cmd]
    in /checkouts/example-repo.
> [build-stdout] Single-threaded extraction.
> [build-stdout] Extracting
...
> Finalizing database at /codeql-dbs/example-repo.
> Successfully created database at /codeql-dbs/example-repo.
```

Multiple language example

This example creates two CodeQL databases for the repository checked out at `/checkouts/example-repo-multi`. It uses:

- `--db-cluster` to request analysis of more than one language.
- `--language` to specify which languages to create databases for.
- `--command` to tell the tool the build command for the codebase, here `make`.
- `--no-run-unnecessary-builds` to tell the tool to skip the build command for languages where it is not needed (like Python).

The resulting databases are stored in `python` and `cpp` subdirectories of `/codeql-dbs/example-repo-multi`.

```
$ codeql database create /codeql-dbs/example-repo-multi \
    --db-cluster --language python,c-cpp \
    --command make --no-run-unnecessary-builds \
    --source-root /checkouts/example-repo-multi
Initializing databases at /codeql-dbs/example-repo-multi.
Running build command: [make]
[build-stdout] Calling python3 /codeql-bundle/codeql/python/tools/get_venv_lib.py
[build-stdout] Calling python3 -S /codeql-
bundle/codeql/python/tools/python_tracer.py -v -z all -c /codeql-dbs/example-
repo-multi/python/working/trap_cache -p ERROR: 'pip' not installed.
[build-stdout] /usr/local/lib/python3.6/dist-packages -R /checkouts/example-repo-
multi
[build-stdout] [INFO] Python version 3.6.9
[build-stdout] [INFO] Python extractor version 5.16
[build-stdout] [INFO] [2] Extracted file /checkouts/example-repo-multi/hello.py
in 5ms
[build-stdout] [INFO] Processed 1 modules in 0.15s
[build-stdout] <output from calling 'make' to build the C/C++ code>
Finalizing databases at /codeql-dbs/example-repo-multi.
Successfully created databases at /codeql-dbs/example-repo-multi.
$
```

Progress and results

Errors are reported if there are any problems with the options you have specified. For interpreted languages, the extraction progress is displayed in the console. For each source file, the console shows if extraction was successful or if it failed. For compiled languages, the console will display the output of the build system.

When the database is successfully created, you'll find a new directory at the path specified in the command. If you used the `--db-cluster` option to create more than one database, a subdirectory is created for each language. Each CodeQL database directory contains a number of subdirectories, including the relational data (required for analysis) and a source archive—a copy of the source files made at the time the database was created—which is used for displaying analysis results.

Creating databases for non-compiled languages

The CodeQL CLI includes extractors to create databases for non-compiled languages—specifically, JavaScript (and TypeScript), Python, and Ruby. These extractors are automatically invoked when you specify JavaScript, Python, or Ruby as the `--language` option when executing `database create`. When creating databases for these languages you must ensure that all additional dependencies are available.

> **Note:** When you run `database create` for JavaScript, TypeScript, Python, and Ruby, you should not specify a `--command` option. Otherwise this overrides the normal extractor invocation, which will create an empty database. If you create databases for multiple languages and one of them is a compiled language, use the `--no-run-unnecessary-builds` option to skip the command for the languages that don't need to be compiled.

JavaScript and TypeScript

Creating databases for JavaScript requires no additional dependencies, but if the project includes TypeScript files, you must install Node.js 6.x or later. In the command line you can specify `--language=javascript-typescript` to extract both JavaScript and TypeScript files:

```
codeql database create --language=javascript-typescript --source-root <folder-to-extract> <output-folder>/javascript-database
```

Here, we have specified a `--source-root` path, which is the location where database creation is executed, but is not necessarily the checkout root of the codebase.

By default, files in `node_modules` and `bower_components` directories are not extracted.

Python

When creating databases for Python you must ensure:

- You have Python 3 installed and available to the CodeQL extractor.
- You have the version of Python used by your code installed.
- You have access to the pip packaging management system and can install any packages that the codebase depends on.
- You have installed the virtualenv pip module.

In the command line you must specify `--language=python`. For example:

```
codeql database create --language=python <output-folder>/python-database
```

This executes the `database create` subcommand from the code's checkout root, generating a new Python database at `<output-folder>/python-database`.

Ruby

Creating databases for Ruby requires no additional dependencies. In the command line you must specify `--language=ruby`. For example:

```
codeql database create --language=ruby --source-root <folder-to-extract> <output-folder>/ruby-database
```

Here, we have specified a `--source-root` path, which is the location where database creation is executed, but is not necessarily the checkout root of the codebase.

Creating databases for compiled languages

For compiled languages, CodeQL needs to invoke the required build system to generate a database, therefore the build method must be available to the CLI.

Detecting the build system

The CodeQL CLI includes autobuilders for C/C++, C#, Go, Java, and Swift code. CodeQL autobuilders allow you to build projects for compiled languages without specifying any build commands. When an autobuilder is invoked, CodeQL examines the source for evidence of a build system and attempts to run the optimal set of commands required to extract a database. For more information, see "CodeQL code scanning for compiled languages."

An autobuilder is invoked automatically when you execute `codeql database create` for a compiled `--language` if you don't include a `--command` option. For example, for a Java codebase, you would simply run:

```
codeql database create --language=java-kotlin <output-folder>/java-database
```

If a codebase uses a standard build system, relying on an autobuilder is often the simplest way to create a database. For sources that require non-standard build steps, you may need to explicitly define each step in the command line.

Notes:

- If you are building a Go database, install the Go toolchain (version 1.11 or later) and, if there are dependencies, the appropriate dependency manager (such as dep).
- The Go autobuilder attempts to automatically detect code written in Go in a repository, and only runs build scripts in an attempt to fetch dependencies. To force CodeQL to limit extraction to the files compiled by your build script, set the environment variable `CODEQL_EXTRACTOR_GO_BUILD_TRACING=on` or use the `--command` option to specify a build command.

Specifying build commands

The following examples are designed to give you an idea of some of the build commands that you can specify for compiled languages.

Note: The `--command` option accepts a single argument—if you need to use more than one command, specify `--command` multiple times. If you need to pass subcommands and options, the

whole argument needs to be quoted to be interpreted correctly.

- C/C++ project built using `make`:

```
codeql database create cpp-database --language=c-cpp --command=make
```

- C# project built using `dotnet build`:

  It is a good idea to add `/t:rebuild` to ensure that all code will be built, or do a prior `dotnet clean` (code that is not built will not be included in the CodeQL database):

```
codeql database create csharp-database --language=csharp --command='dotnet
build /t:rebuild'
```

- Go project built using the `CODEQL_EXTRACTOR_GO_BUILD_TRACING=on` environment variable:

```
CODEQL_EXTRACTOR_GO_BUILD_TRACING=on codeql database create go-database --
language=go
```

- Go project built using a custom build script:

```
codeql database create go-database --language=go --
command='./scripts/build.sh'
```

- Java project built using Gradle:

```
# Use `--no-daemon` because a build delegated to an existing daemon cannot be
detected by CodeQL:
codeql database create java-database --language=java-kotlin --command='gradle
--no-daemon clean test'
```

- Java project built using Maven:

```
codeql database create java-database --language=java-kotlin --command='mvn
clean install'
```

- Java project built using Ant:

```
codeql database create java-database --language=java-kotlin --command='ant -f
build.xml'
```

- Swift project built from an Xcode project or workspace. By default, the largest Swift target is built:

  It's a good idea to ensure that the project is in a clean state and that there are no build artefacts available.

```
xcodebuild clean -all
codeql database create -l swift swift-database
```

- Swift project built with `swift build`:

```
codeql database create -l swift -c "swift build" swift-database
```

- Swift project built with `xcodebuild`:

```
codeql database create -l swift -c "xcodebuild build -target your-target"
swift-database
```

You can pass the `archive` and `test` options to `xcodebuild`. However, the standard `xcodebuild` command is recommended as it should be the fastest, and should be all that CodeQL requires for a successful scan.

- Swift project built using a custom build script:

```
codeql database create -l swift -c "./scripts/build.sh" swift-database
```

- Project built using Bazel:

```
# Navigate to the Bazel workspace.

# Before building, remove cached objects
# and stop all running Bazel server processes.
bazel clean --expunge

# Build using the following Bazel flags, to help CodeQL detect the build:
# `--spawn_strategy=local`: build locally, instead of using a distributed
build
# `--nouse_action_cache`: turn off build caching, which might prevent
recompilation of source code
# `--noremote_accept_cached`, `--noremote_upload_local_results`: avoid using
a remote cache
codeql database create new-database --language=<language> \
--command='bazel build --spawn_strategy=local --nouse_action_cache --
noremote_accept_cached --noremote_upload_local_results
//path/to/package:target'

# After building, stop all running Bazel server processes.
# This ensures future build commands start in a clean Bazel server process
# without CodeQL attached.
bazel shutdown
```

- Project built using a custom build script:

```
codeql database create new-database --language=<language> --
command='./scripts/build.sh'
```

This command runs a custom script that contains all of the commands required to build the project.

Using indirect build tracing

If the CodeQL CLI autobuilders for compiled languages do not work with your CI workflow and you cannot wrap invocations of build commands with `codeql database trace-command`, you can use indirect build tracing to create a CodeQL database. To use indirect build tracing, your CI system must be able to set custom environment variables for each build action.

To create a CodeQL database with indirect build tracing, run the following command from the checkout root of your project:

```
codeql database init ... --begin-tracing <database>
```

You must specify:

- `<database>`: a path to the new database to be created. This directory will be created

when you execute the command—you cannot specify an existing directory.
- `--begin-tracing`: creates scripts that can be used to set up an environment in which build commands will be traced.

You may specify other options for the `codeql database init` command as normal.

> **Note:** If the build runs on Windows, you must set either `--trace-process-level <number>` or `--trace-process-name <parent process name>` so that the option points to a parent CI process that will observe all build steps for the code being analyzed.

The `codeql database init` command will output a message:

```
Created skeleton <database>. This in-progress database is ready to be populated
by an extractor. In order to initialise tracing, some environment variables need
to be set in the shell your build will run in. A number of scripts to do this
have been created in <database>/temp/tracingEnvironment. Please run one of these
scripts before invoking your build command.

Based on your operating system, we recommend you run: ...
```

The `codeql database init` command creates `<database>/temp/tracingEnvironment` with files that contain environment variables and values that will enable CodeQL to trace a sequence of build steps. These files are named `start-tracing.{json,sh,bat,ps1}`. Use one of these files with your CI system's mechanism for setting environment variables for future steps. You can:

- Read the JSON file, process it, and print out environment variables in the format expected by your CI system. For example, Azure DevOps expects `echo "##vso[task.setvariable variable=NAME]VALUE"`.
- Or, if your CI system persists the environment, source the appropriate `start-tracing` script to set the CodeQL variables in the shell environment of the CI system.

Build your code; optionally, unset the environment variables using an `end-tracing.{json,sh,bat,ps1}` script from the directory where the `start-tracing` scripts are stored; and then run the command `codeql database finalize <database>`.

Once you have created a CodeQL database using indirect build tracing, you can work with it like any other CodeQL database. For example, analyze the database, and upload the results to GitHub if you use code scanning.

Example of creating a CodeQL database using indirect build tracing

> **Note:** If you use Azure DevOps pipelines, the simplest way to create a CodeQL database is to use GitHub Advanced Security for Azure DevOps. For documentation, see Configure GitHub Advanced Security for Azure DevOps in Microsoft Learn.

The following example shows how you could use indirect build tracing in an Azure DevOps pipeline to create a CodeQL database:

```yaml
steps:
    # Download the CodeQL CLI and query packs...
    # Check out the repository ...

    # Run any pre-build tasks, for example, restore NuGet dependencies...

    # Initialize the CodeQL database.
    # In this example, the CodeQL CLI has been downloaded and placed on the PATH.
    - task: CmdLine@1
      displayName: Initialize CodeQL database
```

```
      inputs:
          # Assumes the source code is checked out to the current working
directory.
          # Creates a database at `<current working directory>/db`.
          # Running on Windows, so specifies a trace process level.
          script: "codeql database init --language csharp --trace-process-name
Agent.Worker.exe --source-root . --begin-tracing db"

    # Read the generated environment variables and values,
    # and set them so they are available for subsequent commands
    # in the build pipeline. This is done in PowerShell in this example.
    - task: PowerShell@1
       displayName: Set CodeQL environment variables
       inputs:
          targetType: inline
          script: >
             $json = Get-Content
$(System.DefaultWorkingDirectory)/db/temp/tracingEnvironment/start-tracing.json |
ConvertFrom-Json
             $json.PSObject.Properties | ForEach-Object {
                 $template = "##vso[task.setvariable variable="
                 $template += $_.Name
                 $template += "]"
                 $template += $_.Value
                 echo "$template"
             }

    # Execute the pre-defined build step. Note the `msbuildArgs` variable.
    - task: VSBuild@1
        inputs:
          solution: '**/*.sln'
          msbuildArgs: /p:OutDir=$(Build.ArtifactStagingDirectory)
          platform: Any CPU
          configuration: Release
          # Execute a clean build, in order to remove any existing build
artifacts prior to the build.
          clean: True
       displayName: Visual Studio Build

    # Read and set the generated environment variables to end build tracing. This
is done in PowerShell in this example.
    - task: PowerShell@1
       displayName: Clear CodeQL environment variables
       inputs:
          targetType: inline
          script: >
             $json = Get-Content
$(System.DefaultWorkingDirectory)/db/temp/tracingEnvironment/end-tracing.json |
ConvertFrom-Json
             $json.PSObject.Properties | ForEach-Object {
                 $template = "##vso[task.setvariable variable="
                 $template += $_.Name
                 $template += "]"
                 $template += $_.Value
                 echo "$template"
             }

    - task: CmdLine@2
       displayName: Finalize CodeQL database
       inputs:
          script: 'codeql database finalize db'

    # Other tasks go here, for example:
    # `codeql database analyze`
    # then `codeql github upload-results` ...
```

Next steps

- To learn how to use the CodeQL CLI to analyze the database you created from your code, see "Analyzing your code with CodeQL queries."