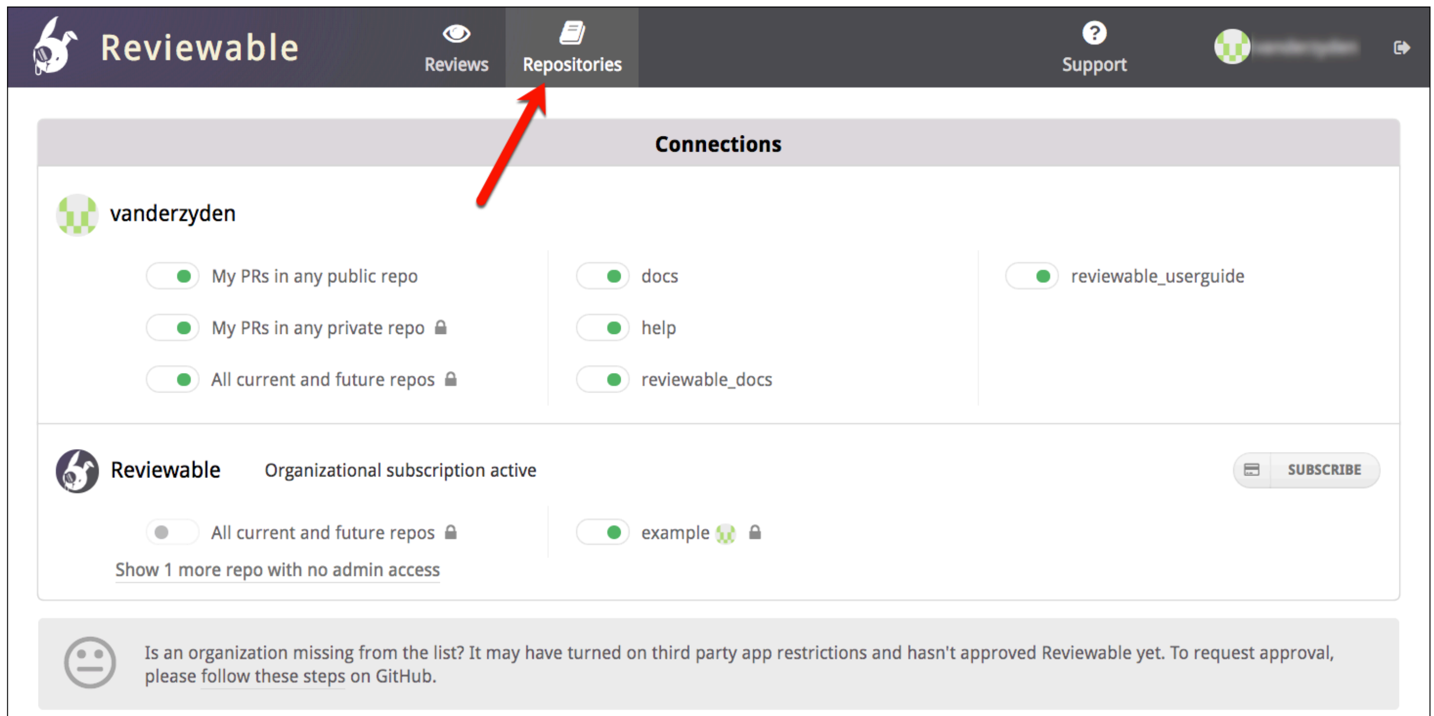
Repositories

The Repositories page lists all of your repos. From this page, you can connect Reviewable to Github repositories, adjust the connections' settings, and manage your Reviewable subscriptions (for which, please see the next chapter).



The repositories are grouped by owner and listed alphabetically. If you don't see an organization of which you're a member, ensure that you click **Show all organizations** if it's there. If the organization is still missing, check whether you need to request approval for Reviewable from your organization owners.

Security concerns

First off, Reviewable will never store your source code on its servers. Each session will fetch the code directly from GitHub into the browser over a secure HTTPS connection. For transparency, here's a non-exhaustive list of the kinds of data stored on Reviewable's servers:

- Comments, including drafts.

- Pull request metadata, such as filenames, and commit and file SHAs.
- Basic account data, such as ID, username, and email address.
- The OAuth access token that you authorized (encrypted for extra security).
- Repo permissions and organization memberships.
- Settings for all levels: organization, repository, user, and review.
- Subscription data, but only the last 4 of the credit card and expiration date are kept.
- Issue titles, commit messages, and GitHub branch protection settings are cached and flushed regularly.

Access is controlled by a set of standalone security rules that are enforced directly by the database. Access permissions are inherited from GitHub and rechecked every 15 minutes to 2 hours, depending on the permission's power. All data is always transmitted across secure connections.

The access token remains encrypted at rest with a key known only to Reviewable servers, and used only to access GitHub on your behalf. Unless you grant explicit written authorization, Reviewable staff will never use the token to access your repository contents or mutate data. (We may use it to test innocuous read-only metadata API calls when debugging an issue specific to your account.)

Reviewable does need write permissions for your repos. See the GitHub authorizations section for a full explanation.

Because Reviewable is an OAuth app, the commit statuses it posts on pull requests cannot be authenticated and could be posted by any user or app with status write permissions under the same context. This could let someone bypass branch protection settings by faking a "review complete" status. If this is a concern in your environment, you may wish to exercise discretion in granting such permissions to users or apps, or institute an audit process that checks whether the statuses' author is the same user who connected the repository to Reviewable.

And of course under no circumstances will we disclose any of your private information to other parties beyond what's needed to provide our service — please see our terms of service and privacy policy for the legal details.

If you need more details about our security architecture or have any other concerns we can address, please contact us at support@reviewable.io.

Connecting repositories

The indicator next to each repository name shows the connection state for this repo. While a repo is connected, Reviewable automatically creates a review for any open PR and inserts a link into all open PRs in the repo.

The toggle's color reflects the state of the connection:

- **Black** — The repo is disconnected. Reviewable will not automatically create reviews for this repo, but it is possible to initiate a review from the Reviews dashboard.
- **Green** — The repo is connected and healthy. Reviewable will automatically create and update reviews for all open PRs and insert a link to the review into the description for each PR. (You can customize this latter behavior in the settings, but must do so *before* connecting the repo!)
- **Red** — The repo was connected, but the connection is now broken. Look for the error message on this page. Though some reviews may be created under this condition, it is necessary to fix the problem to ensure all reviews function properly.

You must have repo admin permissions to connect or disconnect a repo. Connecting to a private organizational repo may cause you to automatically begin the 30-day free trial.

It is entirely safe to connect or disconnect a repo at any time without risk of data loss. After a review is created, it will not be affected by this toggle.

> ?
>
> If you previously connected repos, but later revoked the authorization for Reviewable, you will need to re-authorize access to maintain the connection. You

> will see messages at the top of the repo page that prompt you to take action.

> !
>
> If a user has connected a repo but later leaves an organization, it will be necessary for another admin to toggle the repo off and then on to assume control of the connection. (Reviewable will send a warning email to the original connector if it detects this situation.)

Each connected repository will have an "N open reviews" link under it that will take you to a repository-specific reviews dashboard.

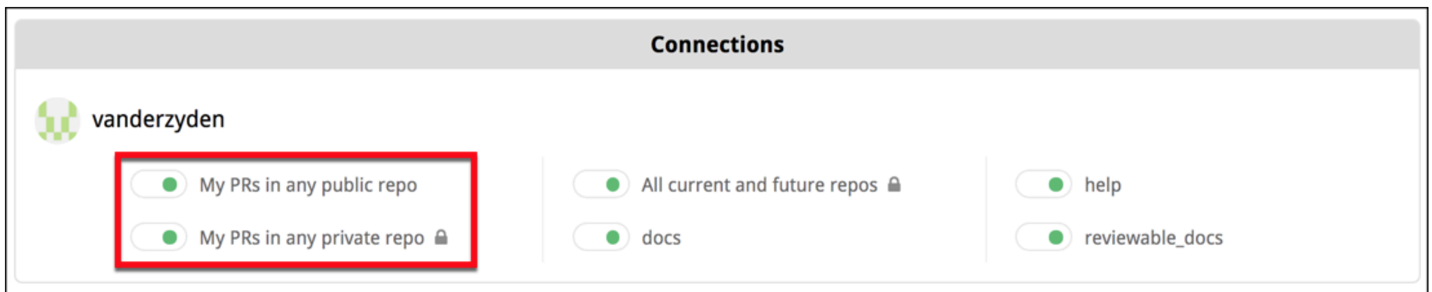
Connect all current and future repos

There's also a special **All current and future repos** toggle. When turned on by an organization owner, Reviewable will connect all current *and future* repos in this organization and automatically create reviews for those repos. Reviewable will not connect any repos that were previously manually toggled off.

> !
>
> You may wish to confirm the settings of current repos and designate a prototype repo for future ones before you turn on this feature. By default, Reviewable will insert a link into all open PRs in all repos unless you've changed this setting beforehand.

Create reviews for your own PRs

You can also get Reviewable to create reviews for all PRs that you author, across all repos. If the **My PRs in any public repo** toggle is on, Reviewable will regularly scan your public PRs and create reviews for them (inserting a link into the PR), covering all of your open source contributions.



The same applies to private PRs for the **My PRs in any private repo** toggle, which gives you the flexibility to have only a subset of users in a private repo use Reviewable.

> !
>
> This last is a legacy feature that may get removed in the future, since it was mainly used to constrain the set of contributors to avoid going over quota, and this can now be specified directly in a subscription's configuration. It will only work if the relevant repo has an active subscription at the time the PR is created and won't backfill if a subscription is created later.

Reviews in connected vs unconnected repos

When you connect a repo, you get links to the reviews from all PRs in that repo and immediate updates whenever anything changes in GitHub.

By contrast, Reviewable doesn't get write access to the repo if you individually connect all your own PRs or create ad-hoc reviews via the dashboard. There are some disadvantages to this approach:

- New commits, GitHub comments, labels, and the PR assignee don't immediately sync with the review, but will only synchronize after somebody loads the review. The dashboard will display stale information for such reviews. Comments posted in Reviewable will propagate immediately.
- Assignee and label directives in GitHub and emailed comments won't apply until a user loads the review.
- Review status checks won't post to the PR, since Reviewable isn't subscribed to repo events and unable to make timely updates.

- Reviewable may not be able to reliably detect and apply your branch protection settings in its UI. (The branch protection settings will be enforced by GitHub no matter what, though, so this is safe but potentially confusing.)
- It may not be possible for Reviewable to pin revision commits, so if you use `git rebase` and `git push --force`, some of them may get garbage collected and will no longer be accessible in the review. They'll usually get pinned when the reviewer (with push authority) accesses the review.

Though the differences above may be minor, it's much more convenient and reliable to connect a repo directly.

> ?
>
> You may find it impractical to use Reviewable for all PRs, especially for small changes. While every pull request from a connected repo will automatically display a button that links it to a Reviewable review, you can simply ignore it and conduct the review in GitHub. Reviewable will close the review when you close the PR. However, if the PRs are in a private organizational repo, each review will count against your contributor maximum — whether you use it or not.

Repository settings

To configure Reviewable repository settings you may use the GUI or the `.reviewable` directory in your project root. Each description below will include setting options using the GUI or by using the `settings.yaml` file in your `.reviewable` directory.

> ?
>
> You can move your review settings from the GUI to the `.reviewable` directory by clicking "Store settings in repo?" link and following the instructions for setting up Reviewable configuration in the repository for your project.

Accessing repository settings with the GUI

Click on a repository name to access the repo settings panel. This works whether the repo is connected or not.

If you make any changes to the settings, click the **Apply** button at the top of the page to commit your changes for the repo you originally chose. Click the adjacent dropdown button to view a panel for specifying additional repos to which these settings will be applied (*all* the settings, not just your current changes). Click **Cancel** to discard any change to the settings.

Prototype settings for new repos

If you are an organization owner, you can set a repo as the settings prototype for any repos not yet accessed or created. Simply click the **Set as prototype for new repos** button and new repos will get a copy of the prototype's settings the first time Reviewable accesses them.

?

This feature is particularly useful if you chose to connect all current and future repos. If you would like more flexibility configuring your connected repositories, you can use a master `settings.yaml` file..

If you would like to see if there is or is not a current prototype repository, and which repository it is, simply hover over the link. A tooltip will come up with one of the following texts:

- "Checking permissions...", "Loading current state..."
    - Please wait a few seconds.
- "Restricted to organization owners."
    - You cannot view or edit this setting unless you're an organization owner.
- "No prototype repository set."
- "This is the current prototype repository."
- "The current prototype repository is ____.".

Store repository settings using the `.reviewable` directory

The `.reviewable` settings directory will allow you to customize your review settings without manually changing settings using the Reviewable user interface. The `.reviewable` directory can contain a `settings.yaml` file and/or a completion condition script (or more than one in case you're using repository-specific overrides).

!

In order to use the `.reviewable` settings directory, the repository needs to be connected. Otherwise, the `.reviewable` directory will be ignored.

!

Reviewable reads the contents of the `.reviewable` directory from the repository's default branch; this is your `main` or `master` branch unless changed

> on GitHub. The contents of other branches don't affect the repository settings until they get merged into the default branch.

The `settings.yaml` file provides several options used to configure the settings for one or more of your repositories. Settings listed at the top level of this file are used as the default settings for the current repository.

View an example `settings.yaml` file here.

> ?
>
> When the `settings.yaml` file is used for your repositories, the settings UI in the repositories section of the Reviewable user interface is hidden and a message will be displayed informing you that the settings for that particular repository are managed via the `settings.yaml` file.

> !
>
> An error is displayed if your `settings.yaml` file contains any options that are not valid, however Reviewable will continue using the last synced value for that option. If the file itself is invalid, Reviewable will default to the last synced value for all settings and the completion condition script if any. Local settings will override any invalid master settings.

Applying a `settings.yaml` file to multiple repositories

Designate a master repository to store your `settings.yaml` file and any completion condition scripts. The settings in this master repository will be used for all repositories in your organization (with the exception of overrides). This master settings file will apply its settings to all repositories, regardless of when they were created.

> ?
>
> You may add a local `settings.yaml` file in an individual repository to override settings from the master settings file.

When a repository is designated as the master, its settings file will be used as the basis for all repositories in the organization. A master settings file can also provide specialized settings for repositories via targeted in-file overrides.

Overrides

When you have one or more repositories with individual `settings.yaml` files, you may use a master repository that will determine default settings for all repositories that belong to an organization. These settings can be overriden in the `overrides` object of the master `settings.yaml` file.

The `overrides` property has two children. The `repositories` object is a list of repositories that will apply the settings specified in the `settings` property of this override. The list of repositories may be a list of strings or `fnmatch` (glob) patterns. For example, if you would like to apply default settings for any repository whose name begins with `dev` or `fire`, you may use the following setting in your master `settings.yaml` file:

```
default-review-style: one-per-commit
overrides:
  - repositories:
    - dev*
    - fire*
  settings:
    # All repositories with names that start with `dev` or `fire` will use "combined commits" for the `default-review-style` setting.
    default-review-style: combined-commits
```

Finalizing master settings

If you wish to manage all repository settings in one central place without allowing the master settings to be overridden by local settings, you can set `final: true` in the master settings which in turn causes any local settings files and completion condition scripts to be ignored.

Reviewable badge

Choose where the Reviewable badge is to be inserted on the GitHub website:

- **Description** — at the top or bottom of the description for the PR. This is convenient since the link will be in a consistent place. However, manual edits to the PR immediately after it's created will race, and might occasionally cause the edits to be lost.
- **Comment** — in a new PR comment. Optionally specify who should be the author of the comment (organization members with access to the repo only). Otherwise, this defaults to the repo connector or review visitor.
- **None** — no badges will be created (private repos only).

If you have a current Reviewable subscription or trial, you may optionally choose when to show the badge:

- **Visited** - show the badge once a review has been accessed for the first time.
- **Started** - only show the badge once a review has been published.
- **Requested** - only show the badge once a review has been requested. This excludes any PR that never left the draft status, since no review is requested for draft PRs.

> ?
>
> Changes here are retroactive (except that an existing description badge won't be moved to a comment), but will be applied lazily as reviews are visited.

```
# settings.yaml

badge:
  # The default setting for `location` is `description-bottom`.
  location: description-bottom | description-top | comment | none
  # `when` is optional.
  when: accessed | published | requested
```

Default review style

Choose the default review style for all reviews in this repo. The choice here affects how commits are grouped into revisions, and the suggested sequence of diffs to review. Please follow a link for a full explanation of the two options.

This setting can be overridden on a particular review by any user with push permissions.

```
# settings.yaml

# The default setting is `combined-commits`.
default-review-style: combined-commits | one-per-commit
```

Default review overlap strategy

Use the current review status of each file to determine how they are presented to a user for review:

- **Defer to user default** - Leave the review strategy up to the users preference.
- **Skip files claimed by others** - Skip any file claimed by another team member in an earlier revision.
- **Review any unreviewed files** - Review any file that requires attention.
- **Review all files personally** - Ensure that you review every file yourself, ignoring other reviewers.

> ?
>
> These settings can be overridden by anyone, but will only apply to the individual user.

```
# settings.yaml

# The default setting is `user-default`
default-review-overlap-strategy: user-default | unclaimed | unreviewed |
personally-unreviewed
```

Approve button output

You can customize the function of the **Approve** button (aka **LGTM** button), which appears on the general discussion when the conditions are right. You can customize what will be inserted into the draft when you click it. By default it inserts `:lgtm:`, which renders a custom LGTM (Looks Good To Me) emoji. But, some teams customize it to insert a form, or a different approval message. The button also always sets the publication status to **Approved**.

```
# settings.yaml

# The `approval-text` option accepts any text.
# The default setting is `:lgtm:`.
approval-text: ":lgtm:" | *
```

Discussion participant dismissers

This setting controls what permissions a user needs to have to be able to dismiss participants from a discussion. By default, anybody with write permissions can do so but you can limit it to only repo admins if a stricter approach is desired.

```
# settings.yaml

# The default setting is `push`
discussion-dismissal-restriction: push | maintain | admin
```

Review status in GitHub PR

This setting determines whether or not to post the current completion status of the review as a commit status on GitHub under the context `code-review/reviewable`. Choose **On for visited reviews** to post only after a review has been visited at least once in Reviewable.

```
# settings.yaml

# The default setting is `accessed`
github-status-updates: accessed | always | never
```

Code coverage

You can configure Reviewable to display code coverage information next to diffs by letting it know where to fetch code coverage reports from. You'll need to enter a URL template that Reviewable can instantiate to grab a report for all the files at a given commit. The template can make use of these variables:

- `{{owner}}`: the repo owner (or organization) username.
- `{{repo}}`: the repo name.
- `{{pr}}`: the pull request number.
- `{{commitSha}}`: the full SHA of the target commit.

If needed, you can also specify one additional header to send with the request. This will typically be an `Authorization` header that passes some kind of access token to enable access to private coverage reports.

> !
>
> The URL template will be available to all users with read permissions on this repo, so make sure to put any sensitive secrets in the headers instead.

> ?
>
> If you added a header we will proxy the request through our server to keep the header's value a secret. However, we have a short list of domains that we're willing to proxy for. If your URL isn't on it you'll get an error and need to get in touch with us to get it whitelisted.

There's a button to let you easily set the report source to Codecov, a popular code coverage report aggregation service. For private repos, you can generate an API access token under your account Settings > Access, and paste it as the value of the `Authorization` header. If you're using a self-hosted instance of Codecov Enterprise then you'll need to set the URL to something like this instead: `https://LOCAL_CODECOV_HOSTNAME/api/ghe/{{owner}}/{{repo}}/commits/{{commitS` `src=extension`, with `LOCAL_CODECOV_HOSTNAME` replaced by the name of the host where you're running Codecov.

The coverage reports must be in a format that Reviewable understands. Currently, we only support the Codecov native API format (both v1 and v2) and Codecov's generic inbound report format. Additionally, if the report has a top-level `error` string property we'll report that through the UI (and ignore any other data), and render any Markdown-style links it contains. If you need support for a different format please let us know and we'll consider it, but in general we're biased towards fetching normalized reports from aggregators.

```
# settings.yaml

coverage:
  # The `url` option allows you to proved a url template for code
coverage reports.
  url: *
```

Completion condition script

The `settings.yaml` file allows you to specify one or more completion condition files for an individual repository, or any repository listed in the `repositories` object of the master `settings.yaml` file. These completion files must be included in the same `.reviewable` directory as your `settings.yaml` file. Reviewable will use a file named `completion.js` by default if it exists and no override specified a different completion file to use.

Below is an example `settings.yaml` file that specifies a default completion conditions for repositories listed in the `overrides` object.

```
overrides:
  - repositories: reviewable-*
    settings:
      completion-file: reviewable-completion.js
  - repositories: hubkit
    settings:
      completion-file: hubkit-completion.js
```

Custom review completion condition

Reviewable allows you to write custom code that determines when a review is complete and controls other details of a review's progress. Typically, you'll use this to customize the number of reviewers required, or switch from the GitHub approval system to a more flexible one based on explicit LGTMs. Some people have created more unusual conditions, though, such as:

- preventing review completion for N hours after a PR was created, so people get a chance to check it out,
- requiring reviews from certain people based on the labels attached to the PR, or
- preventing merging of PRs that have commits with “WIP” in the title.

Development environment

The **Review completion condition** section of the repository settings helps you refine your code in a live evaluation environment.

In the **Condition Code** panel, you can edit the code that determines when a review is complete and otherwise tweaks low-level review data. Simple things are pretty easy to accomplish but you have the power to implement arbitrarily complex logic if you need to. You can find a number of examples in our repository to get you started, and full details follow below.

The condition code will run in an isolated NodeJS 18.x environment, which gets updated regularly. The environment includes the 4.x `lodash` module available through the customary `_`. Note the `lodash` version was updated to `4.x` on *9/9/2021*, so if you have a condition written before the update it will still use the `lodash` 3.x module. You can require other built-in Node modules, though some may be disallowed. Each invocation of your code must return a result within three seconds.

> ?
>
> You can update existing conditions to use `lodash` 4.x by inserting a commmented **dependencies** flag anywhere in your condition code using the following format: `// dependencies: lodash4`

For testing, your code will be continuously evaluated against the **Review state** on the right. It will start off with the current state of some PR in your repo, but you can fill in the state of any PR via the small box above it, or edit the state manually to your liking. See the review state input section below for an explanation of the state's properties.

The results of your code will appear in the **Evaluation result** pane at the bottom of the settings page. They must follow a specific structure described in the condition ouput section below.

Condition function — Examples

```
// This is the built-in review completion condition that
// Reviewable uses by default.

// It checks that all files have been reviewed by at least one
// user and that all discussions have been resolved.  All the
// information about the current review is supplied in a
// predefined `review` variable that will look like the JSON
// structure on the right.  You can edit it here for testing
// how the code will behave in different scenarios.

// You can load other examples from the dropdown menu above.

let reasons = [];  // pieces of the status description
let shortReasons = [];  // pieces of the short status desc.
const summary = review.summary;  // shortcut to summary
const designatedReviewers =
  _.isEmpty(review.pullRequest.requestedReviewers) ?
    review.pullRequest.assignees :
    review.pullRequest.requestedReviewers;
```

Review state — PR #

```
{
  "summary": {
    "lastRevision": "r5",
    "numUnresolvedDiscussions": 7,
    "numFiles": 4,
    "numUnreviewedFiles": 4,
    "numFilesReviewedByAtLeast": [
      4
    ]
  },
  "pullRequest": {
    "number": 1,
    "author": {
      "username": "pkaminski"
    },
    "creationTimestamp": 1410426537000,
    "assignees": [],
    "requestedReviewers": [],
    "requestedTeams": [],
    "approvals": {
```

Evaluation result

```
{ completed: false,
  description: '0 of 4 files reviewed, 7 unresolved discussions',
  shortDescription: '4 files, 7 discussions left',
  pendingReviewers:
   [ { username: 'leereilly' },
     { username: 'josephjang' },
     { username: 'bstrand' },
```

If you would like to test a completion condition before applying the change to your repository, you may use the completion condition playground. The completion condition playground allows you to test a completion script against a pull request or review specified by url. The playground does not allow you to save the completion condition.

Review state input

The current state of the review is accessible to your code via the `review` variable. The sample review state below explains the various properties. All timestamp values indicate milliseconds since the epoch, and all lists are ordered chronologically (when appropriate). If you find that you'd like more data please ask and we'll see what we can do.

```
{
  summary: {
    lastRevision: 'r1',             // The key of the last revision
    numUnresolvedDiscussions: 1,    // The number of unresolved
discussions
    numFiles: 1,                    // Total number of active files in
the review
    numUnreviewedFiles: 1,          // Number of files not reviewed by
anyone at latest revision
    numFilesReviewedByAtLeast: [1]  // Number of files reviewed by at
least N people (as index)
      // e.g., numFilesReviewedByAtLeast[2] is the number of file
reviewed by at least 2 people
    commitsFileReviewed: true
  },
  pullRequest: {
    title: 'Work work work',
    repository: {name: 'Reviewable'},
    number: 44,
    state: 'open',  // one of 'open', 'merged' or 'closed'
    body: 'There is so much work to be done, and this PR does it all.',
    // All users are annotated with a full list of teams they're members
of; if the property is
    // undefined then Reviewable wasn't able to fetch this list.
    author: {username: 'pkaminski', teams: ['reviewable/developers']},
    coauthors: [
      {username: 'pkaminski-test', teams: ['reviewable/semi-developers'],
participating: true}
    ],
    creationTimestamp: 1436825000000,  // added recently, it could be
missing for older reviews
    draft: false,
    assignees: [
      // A user is participating iff they commented or reviewed a file.
      {username: 'pkaminski-test', participating: true},
      {username: 'mdevs5531', participating: false}
    ],
    requestedReviewers: [
      // When executing the condition prior to publishing a review, this
list won't include any
      // reviewers added by the "sync requested reviewers" option if it's
checked.  Doing so would
      // create a dependency cycle.  This only affects the posted message
```

```
-- the condition will be
      // re-evaluated after publishing with the full list of requested
reviewers to determine the
      // actual review status.
      {username: 'pkaminski-test', participating: true}
    ],
    requestedTeams: [
      {slug: 'developers'}
    ],
    sanctions: [
      // Lists pull request reviews by user along with the latest state
(one of 'approved',
      // 'changes_requested', 'commented', or 'dismissed').  Like other
user lists it'll also
      // include each user's team memberships.
      {username: 'pkaminski-state', state: 'changes_requested'}
    ],
    numCommits: 3,
    target: {
      owner: 'pkaminski', repo: 'sample', branch: 'work',
      branchProtected: true,  // whether GitHub's branch protection is
turned on for this branch
      headCommitSha: '3cd017d236fe9174ab22b4a80fefb323dbefb50f'  // may
be missing in old reviews
    },
    source: {owner: 'pkaminski', repo: 'sample', branch: 'pkaminski-
patch-9'},
    // one of dirty, unknown, blocked, behind, unstable, has_hooks,
clean, or draft
    mergeability: 'clean',
    // whether this completion is running on a merge queue commit or on a
normal one
    mergeQueueCheck: false,
    checks: {
      Shippable: {
        state: 'failure',
        descriptio: 'Builds failed on Shippable',
        timestamp: 1432363555000
      }
    }
  },
  pendingReviewers: [  // List of proposed pending reviewers computed by
Reviewable
```

```
    {username: 'pkaminski', teams: ['reviewable/developers']}
    // If the pull request author was added as a last resort, the object
will have `fallback: true`
  ],
  deferringReviewers: [ // List of reviewers who are deferring and will
be removed from pendingReviewers
  // by default unless your completion condition accesses
pendingReviewers or deferringReviewers
    {username: 'cgiroux'}
  ],
  revisions: [  // List of all revisions, in chronological order
    {
      key: 'r1',
      snapshotTimestamp: 1436825047000,  // When this revision was
snapshotted (missing if provisional)
      obsolete: false,
      commitSha: '435ae39a89e6992c9ed72fd154bc3c45290d8a97',
      baseCommitSha: '3cd017d236fe9174ab22b4a80fefb323dbefb50f',
      commits: [
        {sha: '435ae39a89e6992c9ed72fd154bc3c45290d8a97', timestamp:
1436825047000, title: 'Fix foo'}
      ]
    }
  ],
  stage: '2. In progress',  // The latest review stage set by the
completion condition
  labels: [  // List of all labels applied to the pull request
    'Comments only in Reviewable'
  ],
  sentiments: [  // List of sentiments (currently just emojis) extracted
from comments
    {username: 'pkaminski', teams: ['reviewable/developers'], emojis:
['lgtm', 'shipit'], timestamp: 1449045103897}
  ],
  discussions: [  // List of the discussions in the review (metadata
only)
    {
      numMessages: 1,
      resolved: false,  // Whether the overall discussion is resolved
      participants: [
        {
          username: 'pkaminski', teams: ['reviewable/developers'],
          disposition: 'discussing',  // Participant's current
```

```
disposition
          resolved: true,  // False if this participant is blocking
resolution
          read: true,  // False if this participant has unread messages
in this discussion
          lastActivityTimestamp: 1436828040000  // Last time user sent a
message or changed disposition
        }
      ],
      target: {  // Target file location; the top-level discussion
doesn't have a target
        file: 'LICENSE', revision: 'r1', base: false, line: 4
      }
    }
  ],
  files: [  // List of files in the review
    {
      path: 'LICENSE',
      revisions: [  // List of the revisions where this file was changed
        {
          key: 'r1',
          action: 'modified',  // one of 'added', 'modified', 'removed',
or 'renamed' (without changes)
          obsolete: false,
          reverted: false,  // true if this revision of the file is the
same as base
          baseChangesOnly: false,  // true if all changes can be
attributed to the base branch
          reviewers: [  // List of users who marked file as reviewed at
this revision
            {username: 'somebody', timestamp: 1436828040000}  //
timestamp null for legacy or inferred reviews
          ]
        }
      ],
      designatedReviewers: [  // Designations inferred from CODEOWNERS
        {team: 'reviewable/legal'},
        {builtin: 'anyone'}
      ]
    }
  ],
  systemFiles: [ // System files generated by Reviewable, including
commit file
```

```
    {
      path: '-- commits',
      revisions: [  // List of the revisions where this file was changed
        {
          key: 'r1',
          action: 'added',  // one of 'added', 'modified', 'removed'
          obsolete: false,
          reverted: false,
          reviewers: [  // List of users who marked file as reviewed at
this revision
            {username: 'somebody', timestamp: 1436828040000}
          ]
        }
    }
  ]
}
```

The file revision properties require a bit of additional explanation. First, renamed file matching and base change detection is performed only in clients, so the condition will get incomplete input data until a user with appropriate permissions visits the review.

Second, the `baseChangesOnly` flag is computed relative to its revision's *prior revision*, which is not necessarily the immediately preceding one. This becomes important when rebasing multiple commits in a review following "review each commit" style, as Reviewable will do its best to match up each "new" commit to its semantic antecedent. We don't surface these details in the data structure above but our algorithm is fairly robust and biased towards needing strong evidence for a match, so false positive `baseChangesOnly` flags should be extremely rare.

Condition output

Your code must return an object with some or all of the following properties. Any missing properties (at the top level) will be filled in by using the built-in default condition. This means that you can safely return, e.g., just the `disableGitHubApprovals` flag and the rest will be defaulted for you.

If your condition code is asynchronous, you should not return any value synchronously and instead call `done({...})` with your return value once it's ready.

`completed`

A boolean indicating whether the review is complete or not.

`description`

A string describing the current status of the review, such as `2 of 5 files reviewed, 4 unresolved discussions`.

`shortDescription`

A string of no more than 50 characters describing the current status of the review, used for GitHub status checks. If not provided, Reviewable will automatically truncate the `description` instead.

`stage`

A short string describing the stage in some process that this review has reached so far. This value will be saved and returned to the completion condition when it next executes, so it can be used to store a bit of state. It's not currently surfaced in the UI but may be in the future, in which case it's likely that values will be sorted alphabetically so you may want to number your stages. If no value is returned by the condition, Reviewable will automatically assign one of `1. Preparing`, `2. In progress`, or `3. Completed`.

`pendingReviewers`

An array of objects with a `username` property listing the users whose attention is needed to advance the review, like `[{username: 'pkaminski'}]`. The contents of this list will be automatically formatted and appended to the `description` and `shortDescription`. You can either compute this value from scratch, or crib from the `review.pendingReviewers` input value, which contains Reviewable's guess as to who the pending reviewers should be. If you compute your own `pendingReviewers` from scratch, Reviewable will remove any users who are deferring from the list of `pendingReviewers`, unless your code accesses `review.deferringReviewers`.

You can read a description of the default pending reviewers logic and take a look at the code that computes the default value.

`files`

An array of objects that look like `{path: 'full/path/to/file', group: 'Some Group', revisions: [key: 'r1', reviewed: true]}`. (It's OK to just augment the `review.files` structure with additional properties and return the whole thing here.)

- To group files in the file matrix, set an optional `group` property on each file with any name you'd like; all files with the same `group` value will be arranged into a group with that name. Files with no group set will belong to the default, unnamed group. Groups will be sorted alphabetically, so you can force a specific arbitrary order by starting each group name with a digit.
- To mark files as vendored, set an optional `vendored` property to `true` on any such file. These files will default to a special Vendored group, won't participate in file rename matching, and won't display a diff by default. Reviewable has hardcoded path-based heuristics for vendored files as well, which you can override by setting `vendored` to `false` on any files you'd like to exempt.
- To override whether a file has been reviewed at a revision set a `reviewed` boolean property there. By default, a file revision is considered reviewed if it was marked so by at least one user.
- To designate specific people for review, set a `designatedReviewers` property on the file as detailed below.

> ?
>
> If you want to set these properties for system files (such as the Commits file), you'll need to add them to your `files` array explicitly as they're part of `review.systemFiles` rather than `review.files`.

Designated reviewers

Designated reviewers are a list of individuals and teams who have been requested to review a given file. They are grouped into "scopes", which you can use to indicate the

focus of the requested review (e.g., "security"), hint at multiplicity requirements (e.g., "one lead or two devs"), or provide any other context for a group of designations that you'd like. There's an implicit and unnamed default scope; you can mix named scopes with the default one but this can result in confusing UX so it's best avoided.

The per-file `designatedReviewers` property should be an array of any of the following:

- A specific user identified by their username: `{username: 'pkaminski'}`.
- A team identified by their team slug: `{team: 'reviewable/security-team'}`.
- A special marker to indicate that anyone is welcome to review the file: `{builtin: 'anyone'}`. This marker cannot be scoped, but it's fine to mix with scoped designations as it gets special treatment in the UI. Leaving it out won't actually prevent undesignated users from reviewing the file, just make it clear that their review isn't needed.
- A special marker to indicate that a given scope has been fulfilled and no further reviewers are needed for it: `{builtin: 'fulfilled', scope: 'security'}`. If used without a `scope` it indicates that the default scope (which includes `{builtin: 'anyone'}`) is fulfilled. It differs from just removing designations targeting that scope altogether as the scope will still be used to group reviewers and indicate that its review requirements have been fulfilled.

Unless otherwise stated, each entry in the array can be modified with any combination of the following:

- A `scope` property to group it into the given scope, e.g., `{username: 'pkaminski', scope: 'security'}`. A given user or team can be added to multiple scopes (though you'll need one entry per scope), in which case a single review will count against all such scopes at once. A scope can have any number of designations.
- An `omitBaseChanges` flag to indicate that this designatee's reviews should carry over any file revisions affected only by base changes, e.g., `{username: 'pkaminski', omitBaseChanges: true}`.

> !
>
> The contents of `designatedReviewers` are *only* used to compute the file review state and will *not* affect whether a file is considered reviewed or not. You'll need to do that yourself, though you can crib from a sample script that matches

> designated reviewers against actual file reviewers to determine whether each revision of a file has been reviewed, and which scopes have been fulfilled.

Here's an example of a `designatedReviewers` property:

```
file.designatedReviewers = [
  // fahhem to review at the latest revision
  {username: 'fahhem'},
  // pkaminski to review at the latest revision as well, though reviews
at earlier
  // revisions will be accepted if all later revisions are
`baseChangesOnly: true`
  {username: 'pkaminski', omitBaseChanges: true},
  // also need a review from security-team, focused on security
  {team: 'reviewable/security-team', scope: 'security'},
  // and anyone else is welcome to review as well!
  {builtin: 'anyone'}
];
```

If `designatedReviewers` is not set it's treated as if it consisted only of `{builtin: 'anyone'}`. Reviewable will also automatically create scopes for designations inferred from `CODEOWNERS` files (`code owners`), unsolicited reviewers if `{builtin: 'anyone'}` is missing (`unsolicited`), and the author of the pull request if they mark a file as reviewed against recommendations (`author`).

> ?
>
> If you have a `CODEOWNERS` file in the repository, the `review.files` input structure will have precomputed `designatedReviewers` properties inferred from the code owners. You can leave these as-is, tweak them (e.g., by removing `{builtin: 'anyone'}` from the array), or overwrite them altogether. Note that if you leave `designatedReviewers` unset for a file it'll fall back to the code owners default instead of `{builtin: 'anyone'}`.

`refreshTimestamp`

A timestamp in milliseconds since the epoch for when the completion condition should be re-evaluated. Useful if some of your logic depends on the current time. You can obtain the current time in a compatible format via `Date.getTime()`. If you try to schedule a refresh less than 5 minutes from now it'll get clamped to 5 minutes, but on-demand refreshes (e.g., triggered by a review visit) will always fire immediately. Any subsequent executions of the condition will override previous `refreshTimestamp`s.

`webhook`

A URL string that Reviewable will send review status update notifications to. You can hook this up directly to a Slack webhook or, through something like Zapier or Integromat, to most any other communication tool. Specifically, whenever the `completed`, `description`, `pendingReviewers`, or merge state of a review changes, after a short debouncing delay Reviewable will `POST` a JSON structure like the following to the webhook URL:

```
{
  // for Slack, this is Slack's Markdown flavor. See
https://www.markdownguide.org/tools/slack/ for details.
  "text": "<https://reviewable.io/reviews/reviewable/demo/1|*Demo code
review (shared)*>  [Reviewable/demo #1]\nReview in progress: 1 of 4 files
reviewed, 2 unresolved discussions\nWaiting on: *pkaminski*",
  // for other Markdown-based applications using more standard Markdown
  "markdown": "[Demo code review (shared)]
(https://reviewable.io/reviews/reviewable/demo/1)  \\[Reviewable/demo
#1\\]\nReview in progress: 1 of 4 files reviewed, 2 unresolved
discussions\nWaiting on: **pkaminski**",
  // for text-based applications
  "plainText": "Demo code review (shared)   [Reviewable/demo #1]\nReview
in progress: 1 of 4 files reviewed, 2 unresolved discussions\nWaiting on:
pkaminski",
  // for HTML-based applications
  "html": "<b><a
href=\"https://reviewable.io/reviews/reviewable/demo/1\">Demo code review
(shared)</a></b>   [Reviewable/demo #1]<br>Review in progress: 1 of
4 files reviewed, 2 unresolved discussions<br>Waiting on:
<b>pkaminski</b>",
  // for email gateways
  "subject": "Demo code review (shared) [Reviewable/demo #1]",
  "key": "Reviewable/demo/1",  // you can use this identifier for
threading

  // The following is meant for other workflows that separate the subject
from the body, such as email-like applications:
  "htmlBody": "<a
href=\"https://reviewable.io/reviews/reviewable/demo/1\">Review in
progress</a>: 1 of 4 files reviewed, 2 unresolved discussions<br>Waiting
on: <b>pkaminski</b>",
  "body": {
    "text": "<https://reviewable.io/reviews/reviewable/demo/1|*Review in
progress: 1 of 4 files reviewed, 2 unresolved discussions*>\nWaiting on:
*pkaminski*",
    "plainText": "Review in progress: 1 of 4 files reviewed, 2 unresolved
discussions\nWaiting on: pkaminski",
    "html": "<a
href=\"https://reviewable.io/reviews/Reviewable/demo/1\">Review in
progress</a>: 1 of 4 files reviewed, 2 unresolved
discussions\n<br>Waiting on: <b>pkaminski</b>",
    "markdown": "[Review in progress]
```

```
(https://reviewable.io/reviews/Reviewable/demo/1): 1 of 4 files reviewed,
2 unresolved discussions  \nWaiting on: **pkaminski**"
  },
  // And now the ultimate in customizability:
  // For if you want to build your own string, or interconnect with
another system,
  // or really anything else!
  "data": {
    "pullRequest": {
      "title": "Demo code review (shared)",
      "owner": "Reviewable",
      "repository": "Reviewable",
      "number": 1,
      "state": "open"
    },
    "review": {
      "url": "https://reviewable.io/reviews/Reviewable/demo/1",
      "completed": false,
      "status": "0 of 4 files reviewed, 3 unresolved discussions"
    },
    "usernames": {
      "author": "pkaminski",
      "waitingOn": ["pkaminski"],
      "commentsFor": []
    }
  },
}
```

If a webhook request fails the error will be displayed to repository admins on the corresponding review page. (The error message returned by your server will technically be accessible to anyone with pull permissions on the repo; however, the webhook URL itself will never be disclosed.)

Note that archived reviews will not generally update their state even if relevant events occur, and hence will not trigger the webhook.

`disableGitHubApprovals`

A boolean that, if true, will disable the “Approve” and “Request changes” options when publishing via Reviewable. This can be useful to prevent confusion if your condition

uses some other values (e.g., LGTMs) to determine completion, but note that users will still be able to publish approving and blocking reviews directly via GitHub.

`syncRequestedReviewers`

A boolean that, if true, will force synchronization of GitHub requested reviewers from `pendingReviewers`. (You should only set it if the repository is connected to Reviewable.) This can be useful to standardize the workflow (e.g., to make metrics provided by another tool more reliable), but note that users will still be able to manually request and unrequest reviewers anyway. When set to `true`, the server will automatically update requested reviewers whenever `pendingReviewers` changes (including when the PR is first created) using any repo admin account. The client will also force enable (`true`) or disable (`false`) the "Sync requested reviewers" option when publishing via Reviewable.

`requestedTeams`

A list of teams whose review should be requested for this pull request. The elements of the list are in the same format as `review.pullRequest.requestedTeams`, i.e. `{slug: 'org/team-slug'}`. The pull request's requested teams will be adjusted so they end up matching the list given here, adding or removing teams as necessary.

`disableBranchUpdates`

A boolean that, if true, will disable the ability to merge the target (base) branch into the source (head) branch in Reviewable's UI. This is to avoid misclicks in workflows where developers are expected to rebase rather than merge. (It's not possible to trigger a rebase through Reviewable's UI unfortunately.)

`mergeStyle`

One of `'merge'`, `'squash'` or `'rebase'`. If set, forces the merge style for a PR *in Reviewable only*. (Does not affect merging via the GitHub UI or API.) If this conflicts with GitHub's permitted merge styles it's ignored.

`defaultMergeCommitMessage`

A string that will be used as the default commit message when merging a pull request from Reviewable in the normal (Merge) mode. The user can edit it before merging as usual.

`defaultSquashCommitMessage`

A string that will be used as the default commit message when merging a pull request from Reviewable in Squash mode. The user can edit it before merging as usual.

`debug`

Any data structure you'd like to be able to inspect when debugging your condition. It'll be displayed in the **Evaluation result** pane but otherwise ignored.