

Repo template: frontend/ backend/ infra/ + protection

This document contains a ready-to-copy repo layout, full contents for `.github/CODEOWNERS`, branch-protection config (API JSON + [gh example](#)), GitHub Actions to enforce Conventional Commit titles and CI checks, plus an [example PR](#) that would pass.

File tree

```
/ (repo root)
├── backend/
│   └── README.md
├── frontend/
│   └── README.md
├── infra/
│   └── README.md
├── .github/
│   ├── CODEOWNERS
│   ├── workflows/
│   │   ├── ci.yml
│   │   └── pr-title-and-status-check.yml
│   └── branch-protection.json
└── example-pr.md
```

1) `.github/CODEOWNERS`

Put this file at `.github/CODEOWNERS` (or in the repo root). Replace team/user placeholders with your org/team names.

```
# CODEOWNERS: map paths to owners (teams or users)
# Require owners' review when files in their area change.

# Backend code
/backend/ @org/backend-team @alice

# Frontend code
/frontend/ @org/frontend-team @bob

# Infra / IaC
```

```
/infra/ @org/infra-team @carol  
  
# Fallback owners (optional)  
* @org/owners
```

Notes: - Use `@org/team` for GitHub teams or `@username` for individuals. - When branch protection enables **Require review from Code Owners**, these owners will be required reviewers for PRs that change matching files.

2) Branch protection config (API JSON + `gh` example)

You can apply branch protection using the GitHub REST API (`PUT /repos/{owner}/{repo}/branches/{branch}/protection`) or with `gh api` / infrastructure-as-code. Below is an example JSON payload that enforces the requested rules.

```
.github/branch-protection.json
```

```
{  
  "required_status_checks": {  
    "strict": true,  
    "contexts": [  
      "ci/build",  
      "ci/test"  
    ]  
  },  
  "enforce_admins": true,  
  "required_pull_request_reviews": {  
    "dismiss_stale_reviews": true,  
    "require_code_owner_reviews": true,  
    "required_approving_review_count": 2  
  },  
  "restrictions": null,  
  "required_linear_history": true,  
  "allow_force_pushes": false,  
  "allow_deletions": false  
}
```

Apply using `gh` (example):

```
# assuming gh auth setup and working dir in repo  
OWNER=your-org-or-user  
REPO=your-repo  
BRANCH=main
```

```
cat .github/branch-protection.json | gh api --method PUT
-H "Accept: application/vnd.github+json"
/repos/$OWNER/$REPO/branches/$BRANCH/protection
-f admin_enforced=true --input -
```

Notes and mapping to your requirements: - `required_approving_review_count: 2` enforces **at least 2 approvals** before merge. - `require_code_owner_reviews: true` ensures owners listed in `CODEOWNERS` must approve when their files change. - `required_status_checks.strict: true` with `contexts` requires the named CI checks to be green before merging. (Make sure your CI workflow reports those contexts.) - `required_linear_history: true` forces a linear history (no merge commits allowed; use squash or rebase merges). - `enforce_admins: true` applies rules to admins (optional — keep if you want admins to obey rules).

3) Workflows

Two example GitHub Actions workflows: `ci.yml` (runs tests and reports `ci/build` and `ci/test`) and `pr-title-and-status-check.yml` (validates PR title conforms to Conventional Commits).

`.github/workflows/ci.yml`

```
name: CI
on:
  push:
    branches: [main]
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    outputs:
      tests_passed: ${{ steps.tests.outcome }}
    steps:
      - uses: actions/checkout@v4
      - name: Set up Node (example)
        uses: actions/setup-node@v4
        with:
          node-version: '18'

      - name: Install
        run: |
          echo "(install step placeholder)"

      - name: Run build
```

```

id: build
run: |
  echo "Build step (placeholder)"

- name: Run tests
  id: tests
  run: |
    echo "Running tests..."
    # return success (simulate passing tests)
    exit 0

- name: Report check summaries
  run: echo "Checks done"

```

This workflow should set statuses named `ci/build` and `ci/test` (the `required_status_checks.contexts` in branch protection). You can tailor the job names and steps to your stack (Python, Node, Go...). The important part is that the workflow sets status contexts the branch protection expects.

`.github/workflows/pr-title-and-status-check.yml`

```

name: PR title + status guard
on:
  pull_request:
    types: [opened, edited, reopened, synchronize]

jobs:
  check-pr-title:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Validate PR title is Conventional Commit
        uses: amitsingh-007/validate-commit-msg@v1
        with:
          pattern: '^(feat|fix|chore|docs|style|refactor|perf|test|build|ci)\n([a-z0-9\-\_]+\+): .+'
        # If you prefer to validate PR title instead of commits, use a small
        script below

      - name: Fallback PR-title-check (script)
        if: failure()
        run: |
          # simple grep on PR title (using the github event payload)
          TITLE=$(jq -r .pull_request.title < "$GITHUB_EVENT_PATH")
          echo "PR title: $TITLE"
          if ! echo "$TITLE" | grep -E '^(feat|fix|chore|docs|style|refactor|'

```

```
perf|test|build|ci)\([a-z0-9_\-]+\): .+'; then
    echo "PR title does not follow Conventional Commits format" >&2
    exit 1
fi
```

Notes: - The action above validates the PR title with a conventional commit-like regex. Adjust prefixes and scope rules to taste. - Use `pull_request` triggers so the check runs on PR creation and updates and reports a status that block merging until passing.

4) Prevent self-approval

GitHub documentation states that **pull request authors cannot approve their own pull requests** — this prevents self-approval by default for required reviews. Branch protection rules and required reviews will enforce this behavior. (See references below.)

If you want extra safety (for example to prevent the last person who pushed to the branch from approving), you can use rulesets or additional checks or a small Action to fail when an approval comes from the PR author or any contributor listed as a committer on that PR.

5) Example PR that passes (example-pr.md)

This file shows a minimal example of how to craft a PR that meets all requirements.

```
# Example PR: feat(frontend): add login form

Branch: `feat/frontend/login-form`

Commits:
- `feat(frontend): add login form component` (conventional commit)

PR Title:
- `feat(frontend): add login form`

What happens when opened:
1. CI runs and reports `ci/build` and `ci/test` contexts as success.
2. PR title check passes (matches Conventional Commit regex).
3. CODEOWNERS assigns `@org/frontend-team` (or they are required due to code owner rule).
4. Two separate reviewers (not the PR author) approve the PR.
5. Branch protection requires 2 approvals, codeowner approval (if affected), passing checks, and linear history.
6. Merge allowed (squash or rebase) and history stays linear.
```

Notes:

- To simulate this in a real repo: push the branch, create PR, have two reviewers approve, ensure workflows pass, then merge.

6) Extra: small GitHub Action to refuse merges where a required rule isn't met

If you want to **explicitly** prevent self-approval beyond default protections, add an optional check that fails when a reviewer is also an author/committer. Example snippet (can be committed to `.github/actions/no-self-approve/action.yml` plus a JS script). This is optional — GitHub already prevents PR authors from approving their own PRs for required reviews.

7) Next steps / how to use these files

1. Replace `@org/*` and example CI contexts with your real teams and CI check names.
 2. Commit files to your repository root. Push to `main`.
 3. Apply branch protection using the supplied JSON + `gh api` command or your org's policy tooling.
 4. Create a branch using the Conventional Commit style and open a PR. Have two distinct reviewers approve and ensure CI workflows pass.
-

References

- GitHub: About protected branches and [Require linear history](#).
- GitHub: Pull request review docs (authors cannot approve their own PRs).