Git and GitHub

1. Commits

1.1 Commands

> *git add -p*

- Stage changes in a more interactive and granular manner.
- Using `git add -p` gives you greater control over your commits, allows for better code review, and helps keep your commit history more organized and meaningful. It's especially helpful when working with large changes or when you want to create separate commits for specific features or bug fixes within a single file.

```
git add -p index.html
```

Other essential commands:

Git Command	Short Definition	Usage
git status	Display the current state of your working directory and index	Check which files are modified, added, or deleted, and view the changes staged for the next commit.
git commit	Create a new commit that records staged changes	Save changes permanently to the Git history after staging with `git add`, providing a commit message.
git log	Display commit history in reverse chronological order	Review commit messages, dates, and authors' information, track development progress, and changes.

1.2 Commit Messages

1. The format should be as follows:

```
<type>[optional scope]: <description> [optional body] [optional footer(s)]
```

1. Use meaningful types: The commit type should reflect the nature of the changes. Use `fix` for bug fixes, `feat` for new features, and other types like `build`, `chore`, `ci`, `docs`, `style`, `refactor`, `perf`, `test`, etc., as appropriate.

2. Include breaking changes explicitly: If your commit introduces breaking changes, indicate it using either a footer with `BREAKING CHANGE: <description>` or by appending `!` to the type/scope prefix.
3. Be concise and clear in the description: The description should be a short summary of the changes made in the commit. Keep it clear, concise, and descriptive.
4. Provide additional context in the body (if needed): For more complex changes, you can use the body section to provide additional context or explanations. Start the body section one blank line after the description.
5. Use scopes for contextual information: If your project is divided into different sections or modules, consider using scopes in the commit message to provide additional contextual information.
6. Follow the specified footer format: If you include footers, make sure they follow the convention similar to the git trailer format. Each footer should consist of a word token, followed by either a `:<space>` or `<space>#` separator, and then the value.
7. Be consistent and case-insensitive: Be consistent in your commit messages and avoid treating the Conventional Commits components as case-sensitive, except for `BREAKING CHANGE` which should be uppercase.

Examples:

```
feat(lang): add Spanish language support
```

```
fix: resolve issue with file parsing Introduce a more efficient parsing
algorithm forlarge files.
```

```
chore!: update dependencies BREAKING CHANGE: Major library dependencies were upgraded.
```

1.2.1 Message pattern (JIRA integration):

```
## Start Commit Message from below
[ticket-id] Overview commit message

Detailed commit message
## End Commit message a line above
```

Example:

```
[GLAD-123] Allow client users to invite people into the project

Create an invitation component inside the ‘Team Members’ component which
is viewable only by those users with the client role in the project by
typing an email address and clicking the dropdown after it is shown.
```

Note: *Each Jira ticket MUST have a corresponding GitHub branch*

1.2.2 The following are rules for the commit messages.

- Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." This convention matches up with commit messages generated by commands like git merge and git revert.
- **First line:** JIRA issue ID in all caps (if applicable), followed by a brief description (~ 50 characters)
- **Second line:** blank
- **Following lines:** more detailed description, line-wrapped at 72 characters. May contain multiple paragraphs, separated by blank lines. Link to the JIRA issue, if applicable.
- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, followed by a single space, with blank lines in between, but conventions vary here
- Use a hanging indent

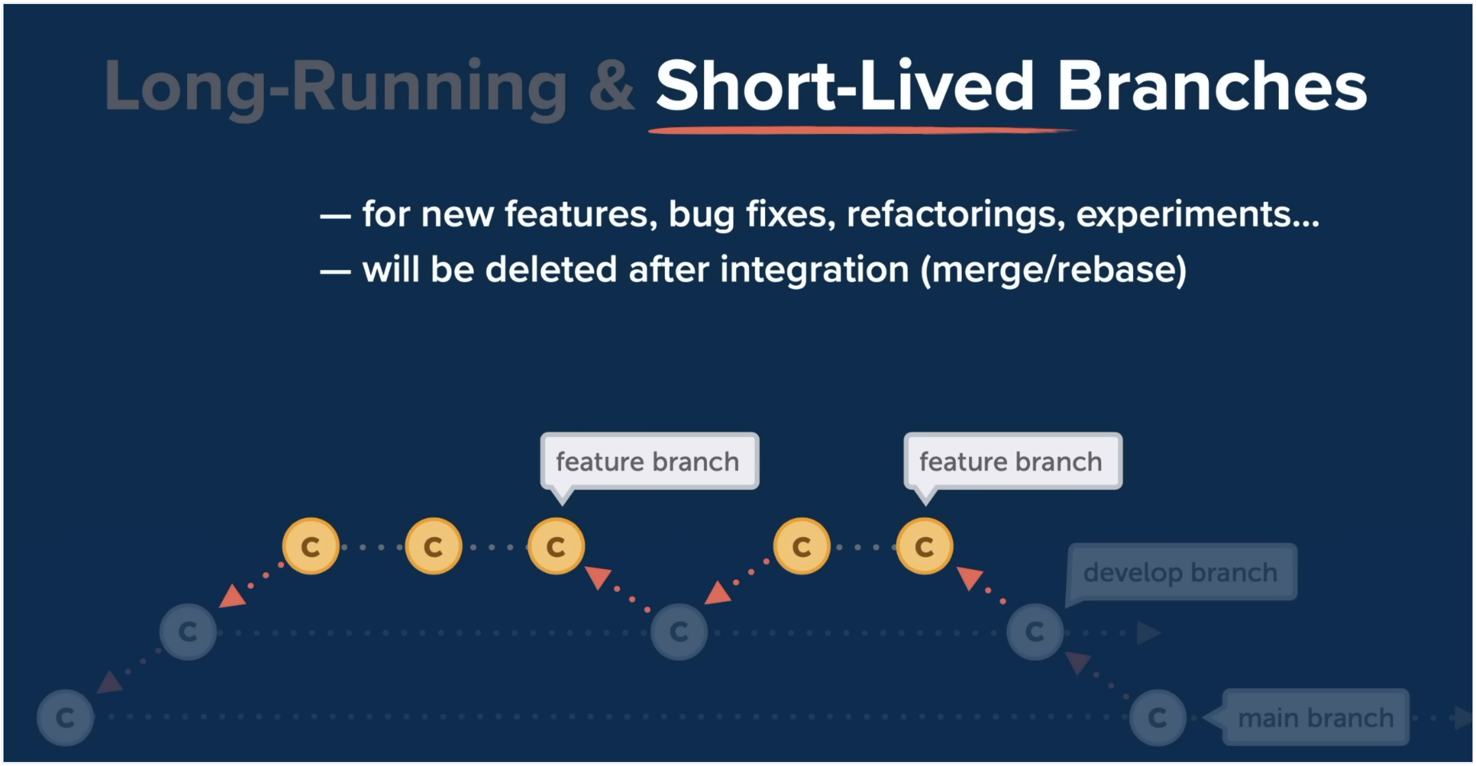
1.2.3 For PR description:

Example:

```
FUSECMP-713 Allow client users to invite people into the project

Create an invitation component inside the ‘Team Members’ component which
is viewable only by those users with client role in the project by
typing an email address and clicking the dropdown after it is shown.
```

2. Branching Strategies

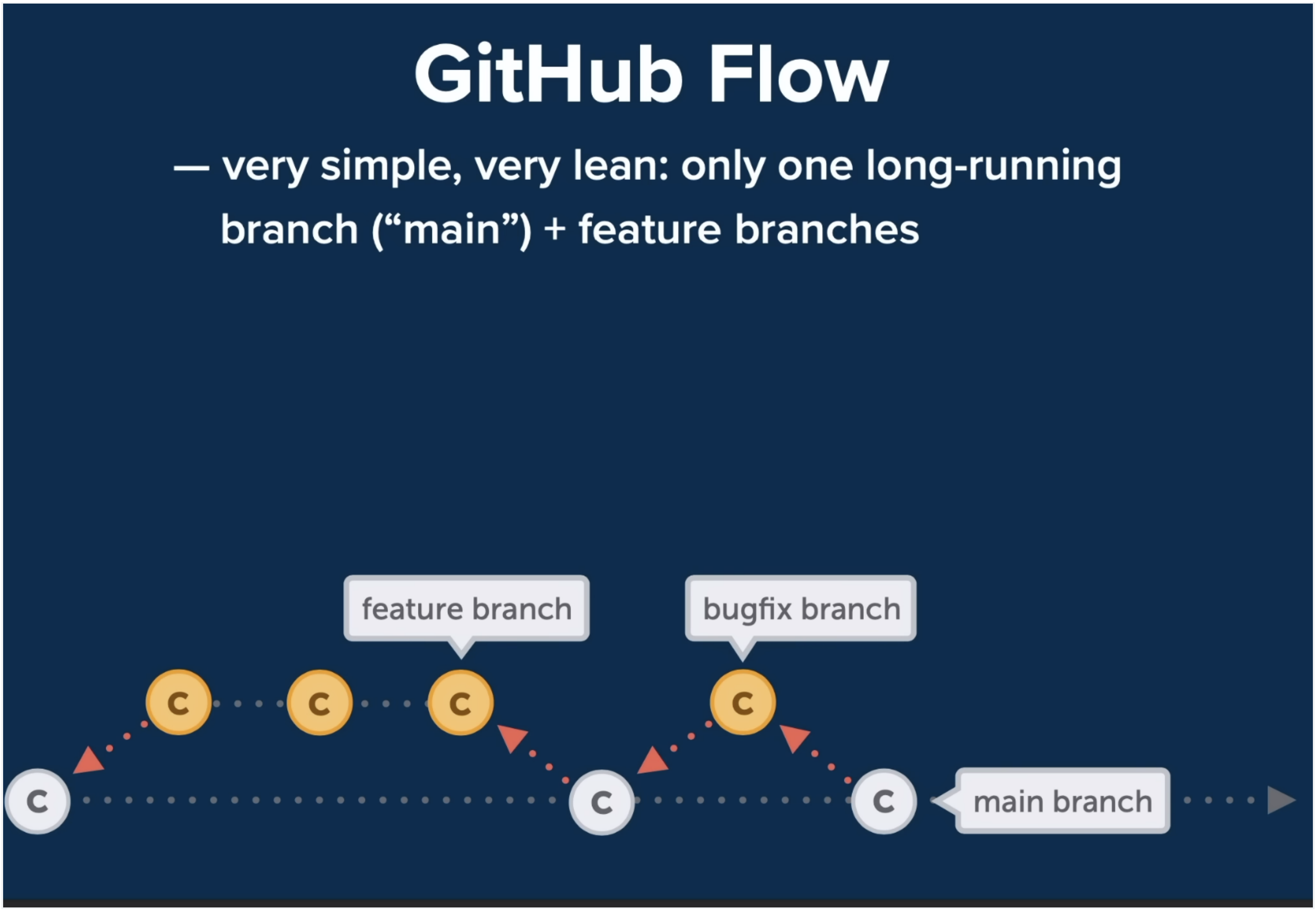
2.1 Long Running & Short-lived Branches



Long-Running & Short-Lived Branches

— for new features, bug fixes, refactorings, experiments…
— will be deleted after integration (merge/rebase)

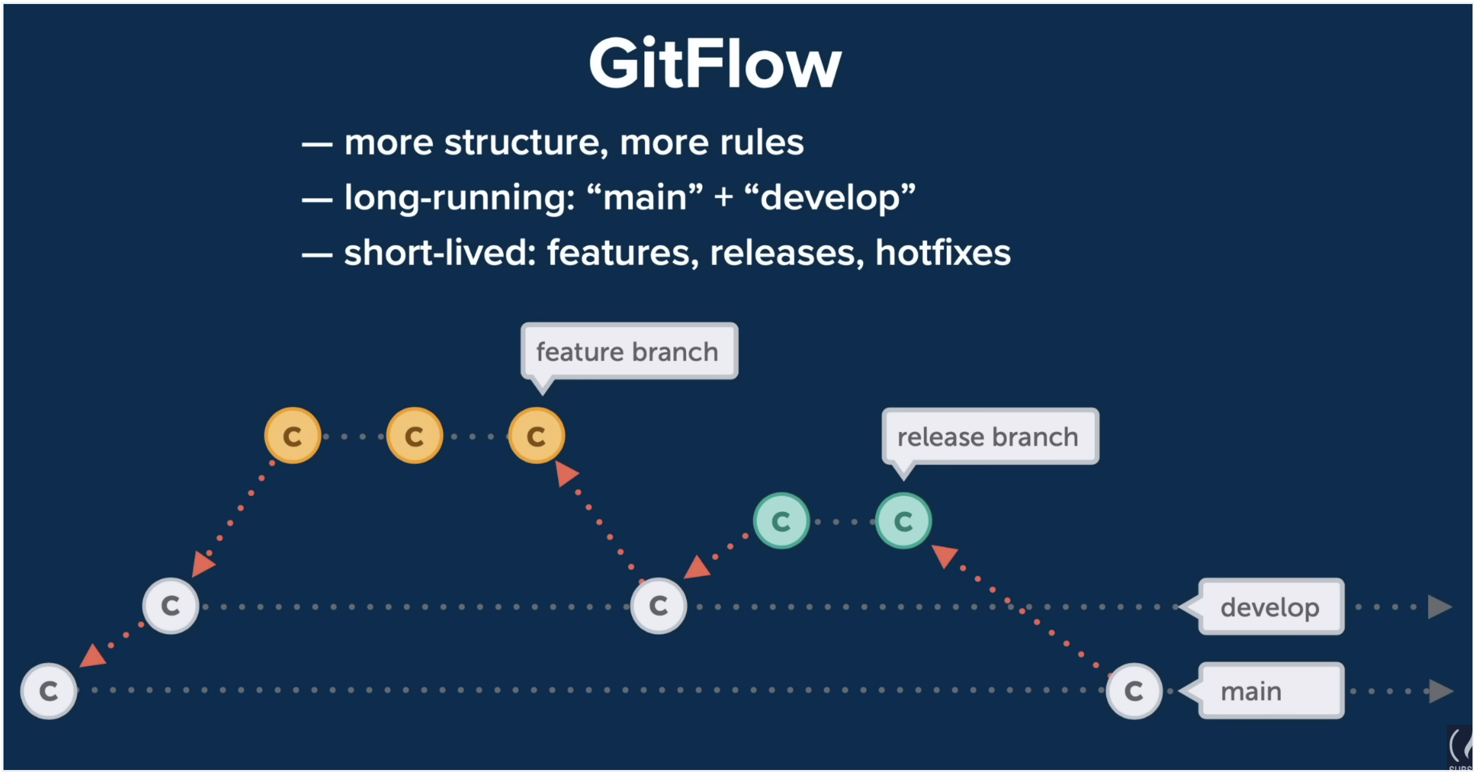
2.1.1 Github Flow



GitHub Flow

— very simple, very lean: only one long-running branch (“main”) + feature branches

2.1.2 GitFlow



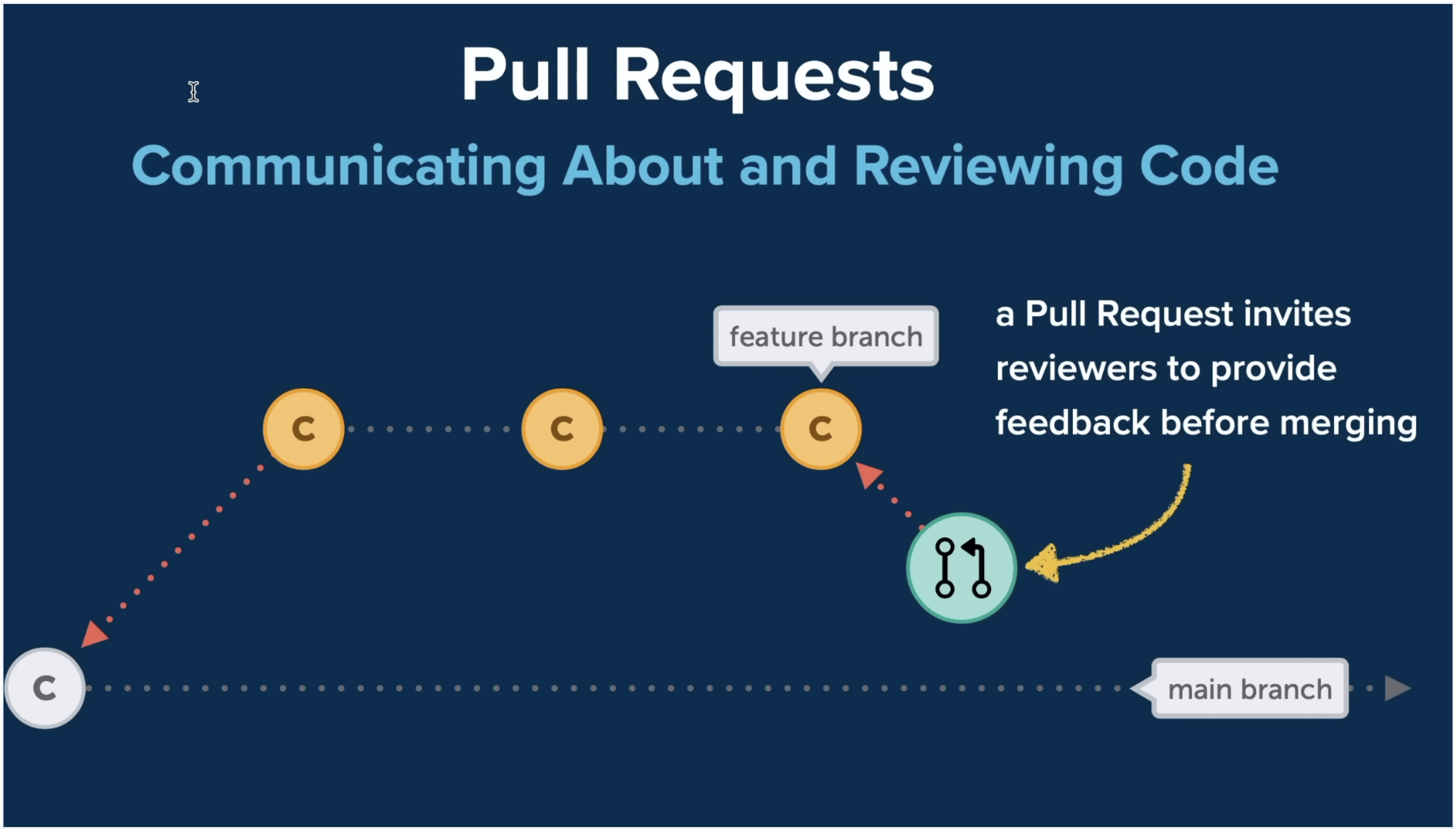
GitFlow

— more structure, more rules

— long-running: “main” + “develop”

— short-lived: features, releases, hotfixes

3. Pull requests

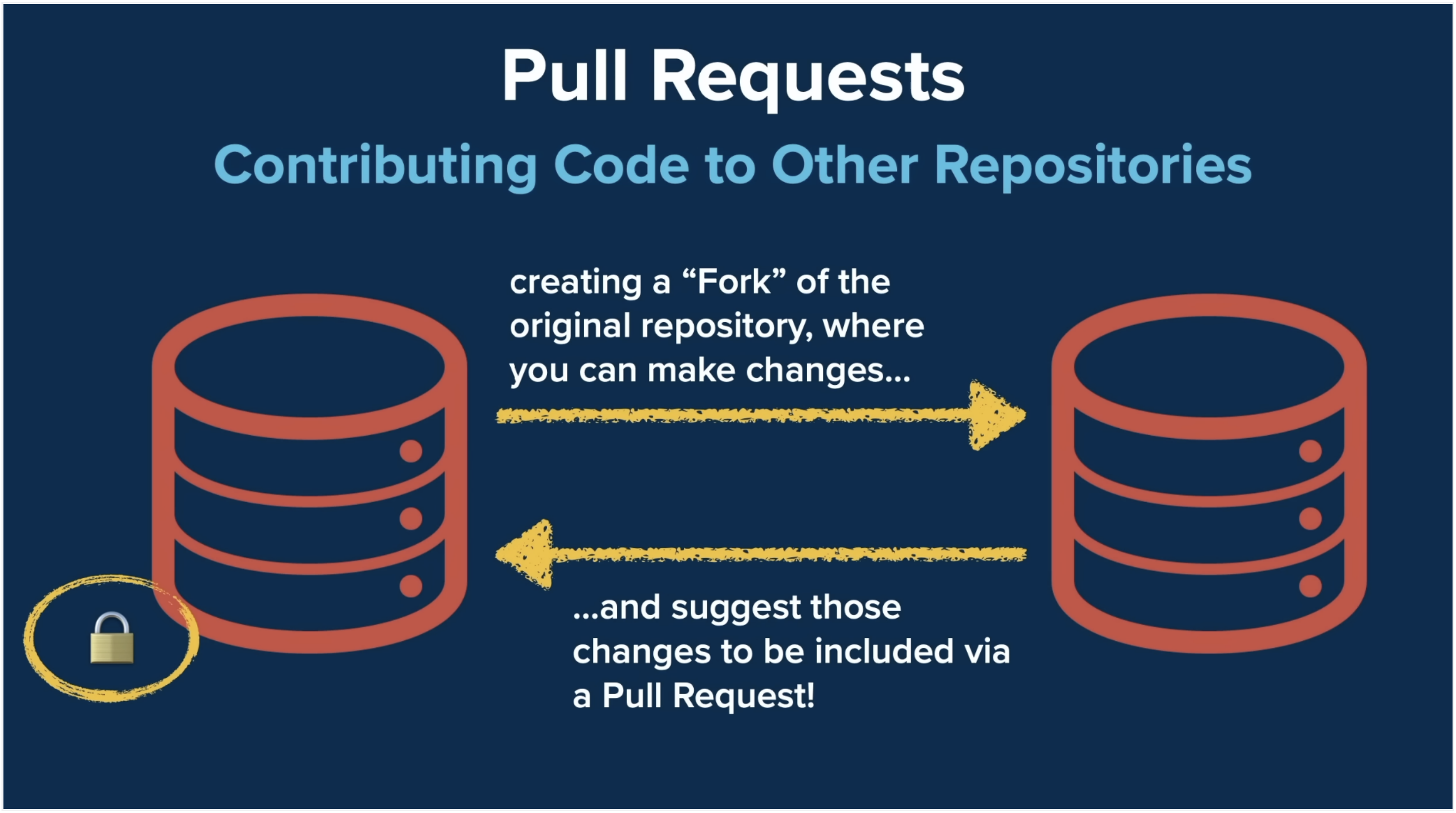


Pull Requests

Communicating About and Reviewing Code

a Pull Request invites reviewers to provide feedback before merging

3.1 Contributing Code to Other Respositories



Pull Requests

Contributing Code to Other Repositories

creating a “Fork” of the original repository, where you can make changes...

...and suggest those changes to be included via a Pull Request!

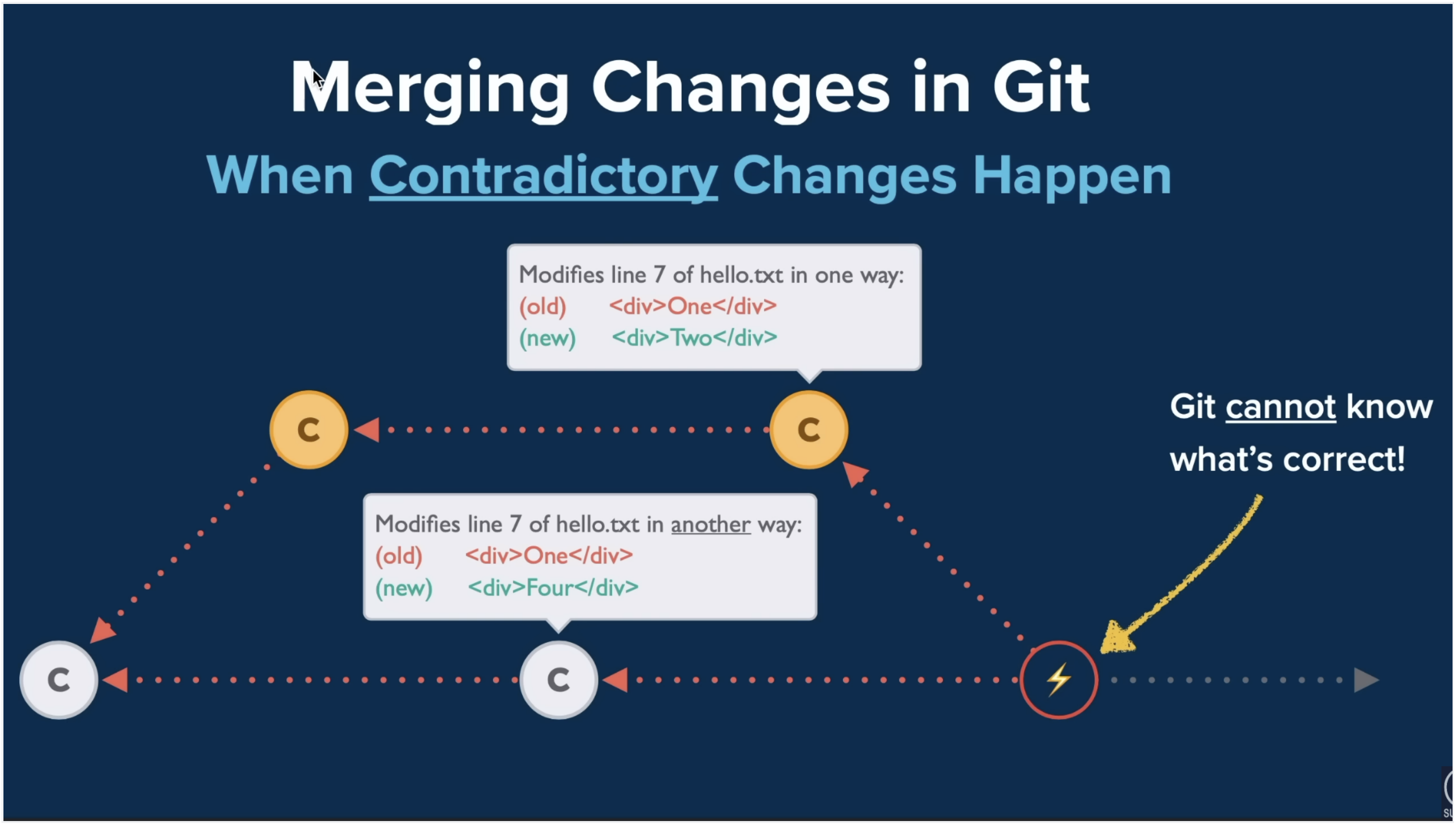
- We create a branch which we later request to be included.

Here is an example Bash script to clone a repo, create a new branch, and create pull requests

```
#!/bin/bash

# Clone the Ruby on Rails repository
git clone https://github.com/gntr/rails.git
cd rails

# Create a new branch and switch to it
git checkout -b feature/my-feature-branch

# Make changes to the code (e.g., add new functionality)

# Stage and commit the changes
git add .
git commit -m "[GLAD-123] Allow client users to invite people into the project

Create an invitation component inside the ‘Team Members’ component which
is viewable only by those users with the client role in the project by
typing an email address and clicking the dropdown after it is shown."

# Push the branch to the remote repository
git push origin feature/my-feature-branch

# Create a pull request using GitHub CLI (gh)
```

```
gh pr create --title "Feature: Add new functionality" --body "This pull request adds a new feature to the project." --base main --head feature/my-feature-branch
```

4. Merge Conflicts



Merging Changes in Git

When Contradictory Changes Happen

Modifies line 7 of hello.txt in one way:
(old) <div>One</div>
(new) <div>Two</div>

Modifies line 7 of hello.txt in another way:
(old) <div>One</div>
(new) <div>Four</div>

Git cannot know what's correct!

4.1 Terminologies and Commands

Git Merge

Definition: Combines changes from different branches into the current branch.

Usage: `git merge <branch-name>`

Example: Merge changes from the "feature" branch into the current branch.

```
git merge feature
```

Git Rebase

Definition: Incorporates changes from one branch by placing commits on top of another branch.

Usage: `git rebase <base-branch>`

Example: Rebase the "feature" branch on top of the "main" branch.

```
git checkout feature
git rebase main
```

Git Pull

Definition: Fetches changes from a remote repository and merges them into the current branch.
Usage: `git pull [<remote>] [<branch>]`
Example: Pull changes from the "origin/main" branch into the current branch.

```
git pull origin main
```

Git Stash Apply

Definition: Applies previously stashed changes back to the working directory.
Usage: `git stash apply [<stash-id>]`
Example: Apply the most recent stashed changes.

```
git stash apply
```

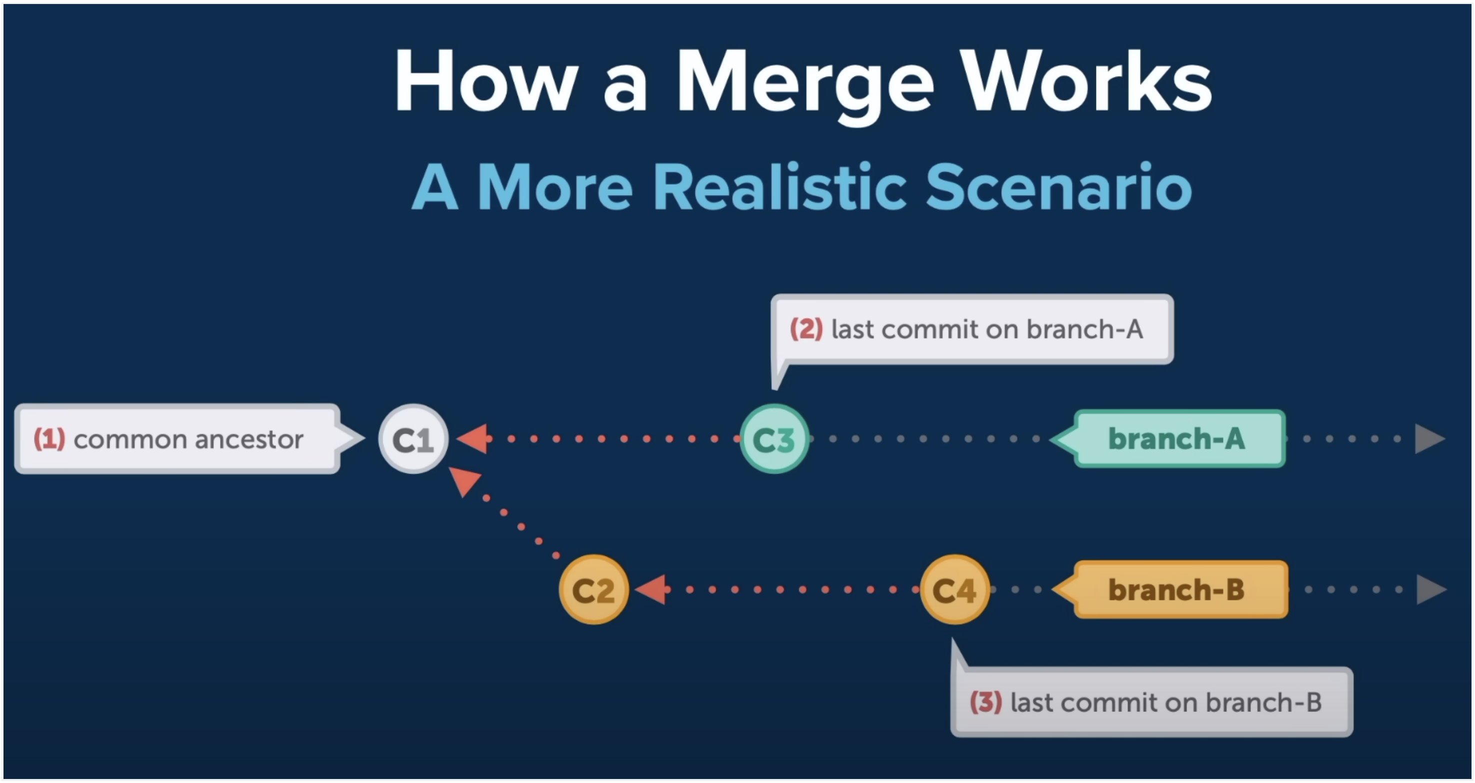
Git Cherry-pick

Definition: Applies specific commits from one branch to another.
Usage: `git cherry-pick <commit-hash>`
Example: Apply the changes from commit "abc123" to the current branch.

```
git cherry-pick abc123
```

Assume we have two branches: `main` and `feature`, and both branches have made changes to the same file `example.txt`. We'll create a merge conflict deliberately to showcase how to resolve it.

4.2 Merge Conflict with `git merge`:


How a Merge Works
A More Realistic Scenario
(2) last commit on branch-A
(1) common ancestor
C1
C3
branch-A
C2
C4
branch-B
(3) last commit on branch-B

1. Switch to the `main` branch and make a change to `example.txt`:

```
git checkout main # Make some changes to example.txt and commit the changes
```

2. Switch to the `feature` branch and make a conflicting change to the same lines in `example.txt`:

```
git checkout feature # Make different changes to the same lines in example.txt and
commit the changes
```

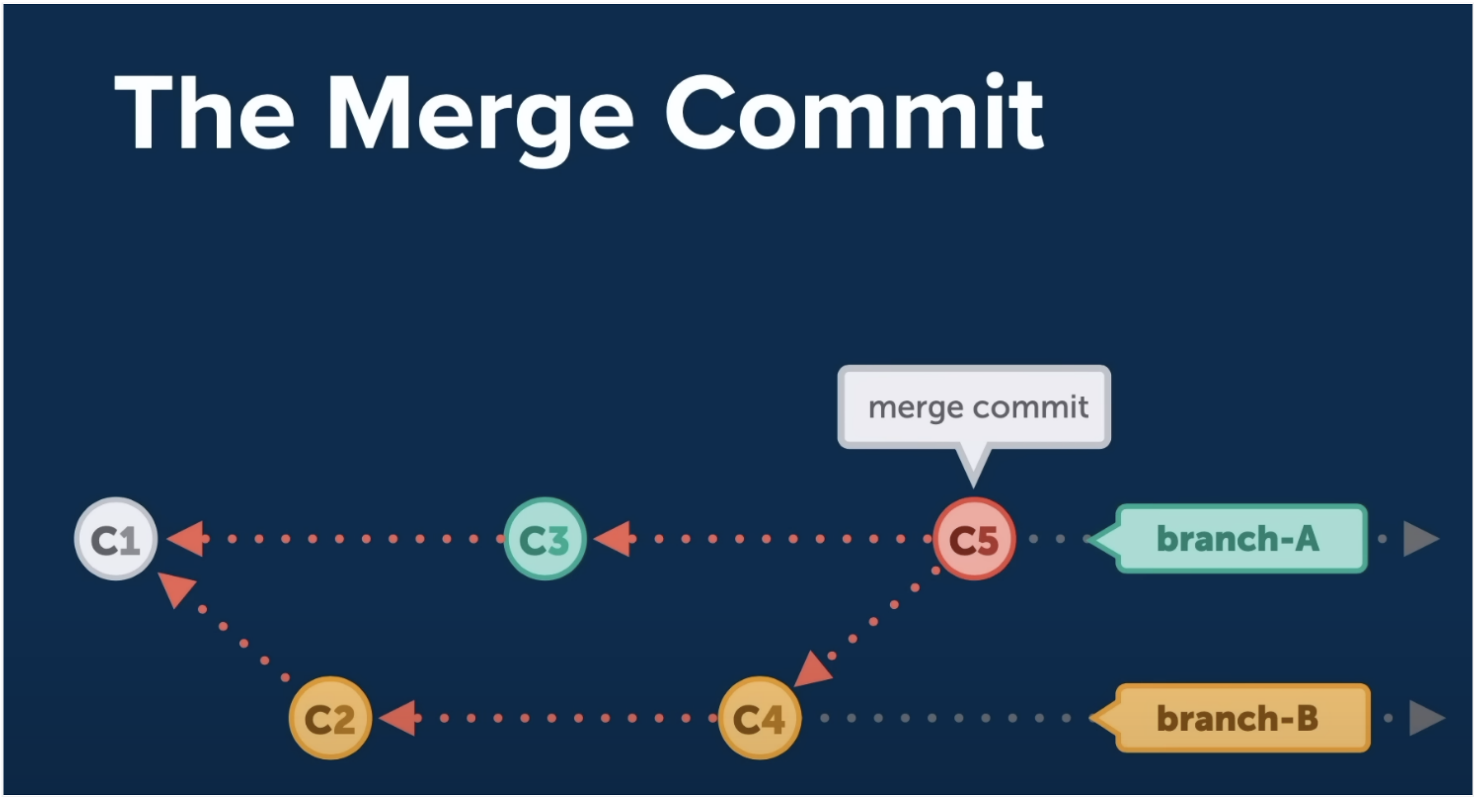
3. Merge the `feature` branch into `main`:

```
git merge feature
```

   This will result in a merge conflict, and Git will notify you of the conflict in `example.txt`.

4. Open `example.txt` and resolve the conflict by manually editing the file to keep the desired changes. Save the file.
5. Add and commit the resolved changes:

```
git add example.txt git commit -m "Merge conflict resolved"
```



The Merge Commit

4.3 Merge Conflict with `git rebase`:
Rebase is an alternative way of merging where you may want the history to look like a straight line.

1. Switch to the `feature` branch and make a change to `example.txt`:

```
git checkout feature # Make some changes to example.txt and commit the changes
```

2. Switch back to the `main` branch and make a conflicting change to the same lines in `example.txt`:

```
git checkout main # Make different changes to the same lines in example.txt and commit
the changes
```

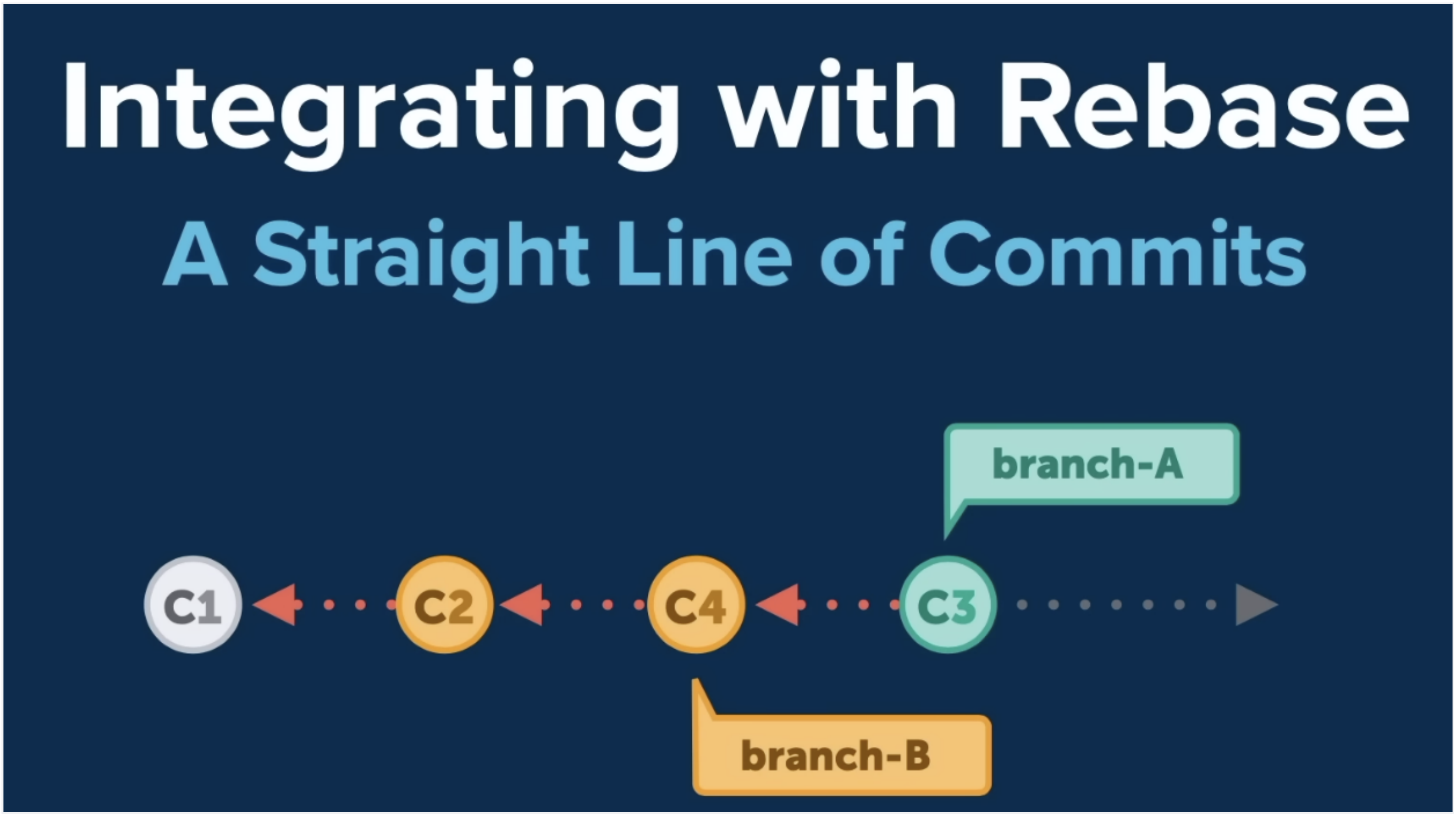
3. Rebase the `feature` branch on top of `main`:

```
git rebase main
```

   This will result in a rebase conflict, and Git will notify you of the conflict in `example.txt`.

4. Open `example.txt` and resolve the conflict by manually editing the file to keep the desired changes. Save the file.
5. Continue the rebase with:

```
git rebase --continue
```


Integrating with Rebase
A Straight Line of Commits
branch-A
C1
C2
C4
C3
branch-B