

Gitting Further

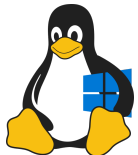
More complex operations with Git

K. Müller¹ J. Verzijden²

¹MasterCLASS
E.T.S.V. Scintilla

²Scintilla Operator Team
E.T.S.V. Scintilla

October 6, 2026



About us



Kasper Müller

kasperm@scintilla.utwente.nl



Johan Verzijden

johanv@scintilla.utwente.nl

Contents

① Configuration

- SSH Keys
- GPG Keys
- Aliases
- Bonus: Handling multiple identities

② Complex Operations

- Reset
- Rebase
- Cherry-picking

③ Various topics

- Forking
- Stashing
- Worktrees
- Bisect

Contents

① Configuration

- SSH Keys
- GPG Keys
- Aliases
- Bonus: Handling multiple identities

② Complex Operations

- Reset
- Rebase
- Cherry-picking

③ Various topics

- Forking
- Stashing
- Worktrees
- Bisect

Why SSH Keys?

SSH Keys

Why SSH Keys?

- Git uses HTTP Basic Auth by default

Why SSH Keys?

- Git uses HTTP Basic Auth by default
 - Has become cumbersome/impossible with 2FA

Why SSH Keys?

- Git uses HTTP Basic Auth by default
 - Has become cumbersome/impossible with 2FA
- Git can also connect over SSH

Why SSH Keys?

- Git uses HTTP Basic Auth by default
 - Has become cumbersome/impossible with 2FA
- Git can also connect over SSH
 - With username/password (still cumbersome)
 - With SSH keys (easy once configured)

SSH Keys - Generate a keypair and upload

- `$ ssh-keygen -t ed25519`
- Specify the location and name, e.g.
`/home/user/.ssh/utwente_id_ed25519`
- Set a passphrase and confirm (remember it!)
- Upload your key to
 - [Gitlab](#)
 - [Github](#)

SSH Keys - Configuration

- Configure SSH

- Open ~/.ssh/config and add the following:

```
Host gitlab.utwente.nl
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/utwente_id_ed25519
```

- Verify that it works

```
$ ssh -T git@gitlab.utwente.nl
```

- Configure Git

```
$ git remote origin set-url git@gitlab.utwente.nl:<number>/<your-project>.git
```

GPG Keys - Overview

What are GPG Keys?

¹Pretty Good Privacy

²GNU Privacy Guard

GPG Keys - Overview

What are GPG Keys?

- PGP¹/GPG²
- Used for cryptographic encryption and signing of data
- In this case: signing of commits
- Proves that the Git identity belongs to you

¹Pretty Good Privacy

²GNU Privacy Guard

GPG Keys - Overview

What are GPG Keys?

- PGP¹/GPG²
 - Used for cryptographic encryption and signing of data
 - In this case: signing of commits
 - Proves that the Git identity belongs to you
-

How to set one up?

¹Pretty Good Privacy

²GNU Privacy Guard

GPG Keys - Overview

What are GPG Keys?

- PGP¹/GPG²
 - Used for cryptographic encryption and signing of data
 - In this case: signing of commits
 - Proves that the Git identity belongs to you
-

How to set one up?

- 1 Installation
- 2 Generate a keypair
- 3 Upload your public key
- 4 Configure Git

¹Pretty Good Privacy

²GNU Privacy Guard

GPG Keys - Installation

- Run `gpg --version` to see if installed
- Not installed? See <https://gnupg.org/download/#binary>
- Our advice:
 - Windows
 - Install Gpg4win
 - Linux
 - You already have it (-:
 - Install the RPM package or ApplImage
 - MacOS
 - Install Mac GPG from [gpgtools](#)

GPG Keys - Generate a keypair

Run `gpg --full-generate-key`. Some remarks:

GPG Keys - Generate a keypair

Run `gpg --full-generate-key`. Some remarks:

- The default key type and size are good
- Set an expiry date
 - In one year is a good default
- Set a name and an email address
 - Useful if same as your Git identity
- Set a passphrase **and store it!**
 - Prevents others from using the key
 - **Keep the passphrase strong and secure!**

GPG Keys - Upload your public key

- 1 Export your **public key**: `gpg --armor --export <keyid>`
 - <keyid> is from the output of `gpg -K --keyid-format=long:`
`sec rsa3072/<keyid> 2024-06-01 [SC] ...`
- 2 Copy everything including BEGIN PGP PUBLIC KEY BLOCK and END PGP PUBLIC KEY BLOCK
- 3 Add it to your account on Github/Gitlab. See the manual for
 - Github
 - Gitlab

GPG Keys - Configure Git

- 1 Tell Git to always sign your commits:

```
git config [--global] commit.gpgsign true
```

- 2 Tell Git which GPG key to use for signing:

```
git config [--global] user.signingkey <keyid>
```

- <keyid> is from the output of `gpg -K --keyid-format=long:`
sec rsa3072/<keyid> 2024-06-01 [SC] ...

- Shortcuts for existing commands (aliases)
 - `git gr` instead of `git log --oneline --graph`
 - `git sw other-branch` instead of `git switch other-branch`
 - `git ust` instead of `git stash pop`
- Completely custom commands
 - Get `git ignore <file>` with `git config --global alias.ignore '! echo $1 >> .gitignore; tail .gitignore; :`

Handling multiple identities

- You probably have multiple identities
 - Personal work
 - University projects
 - Study association
 - etc.

³Source: <https://www.micah.soy/posts/setting-up-git-identities/>

Handling multiple identities

- You probably have multiple identities
 - Personal work
 - University projects
 - Study association
 - etc.
- Managing these is difficult
 - You forget to change it
 - There is no easy way to change it

³Source: <https://www.micah.soy/posts/setting-up-git-identities/>

Handling multiple identities

- You probably have multiple identities
 - Personal work
 - University projects
 - Study association
 - etc.
- Managing these is difficult
 - You forget to change it
 - There is no easy way to change it
- Solution: combine custom configuration and aliases!³

³Source: <https://www.micah.soy/posts/setting-up-git-identities/>

Handling multiple identities

1 Create the alias `git identity <id>`

```
git config alias.identity '!  
git config user.name "${git config user.$1.name}";  
git config user.email "${git config user.$1.email}";  
git config user.signingkey "${git config user.$1.signingkey}"; :'
```

Handling multiple identities

❶ Create the alias `git identity <id>`

```
git config alias.identity '!  
git config user.name "$(git config user.$1.name)";  
git config user.email "$(git config user.$1.email)";  
git config user.signingkey "$(git config user.$1.signingkey)"; :'
```

❷ Let Git force you to choose an identity

- ❶ `git config --global --unset user.name`
- ❷ `git config --global --unset user.email`
- ❸ `git config --global --unset user.signingkey`
- ❹ `git config --global user.useConfigOnly true`

Handling multiple identities

1 Create the alias git identity <id>

```
git config alias.identity '!  
git config user.name "$(git config user.$1.name)";  
git config user.email "$(git config user.$1.email)";  
git config user.signingkey "$(git config user.$1.signingkey)"; :'
```

2 Let Git force you to choose an identity

- 1 git config --global --unset user.name
- 2 git config --global --unset user.email
- 3 git config --global --unset user.signingkey
- 4 git config --global user.useConfigOnly true

3 Create your different identities

- git config --global user.github.name 'Bob'
- git config --global user.github.email 'bob@example.com'
- git config --global user.ut.name 'Bob (BSc)'
- git config --global user.ut.email 'bob@utwente.nl'
- git config --global user.ut.signingkey A6A6A6A6A6A6A6A6

Handling multiple identities

1 Create the alias git identity <id>

```
git config alias.identity '!  
git config user.name "$(git config user.$1.name)";  
git config user.email "$(git config user.$1.email)";  
git config user.signingkey "$(git config user.$1.signingkey)"; :'
```

2 Let Git force you to choose an identity

- 1 git config --global --unset user.name
- 2 git config --global --unset user.email
- 3 git config --global --unset user.signingkey
- 4 git config --global user.useConfigOnly true

3 Create your different identities

- git config --global user.github.name 'Bob'
- git config --global user.github.email 'bob@example.com'
- git config --global user.ut.name 'Bob (BSsc)'
- git config --global user.ut.email 'bob@utwente.nl'
- git config --global user.ut.signingkey A6A6A6A6A6A6A6A6

4 Choose your identity in a repository

```
git identity github or git identity ut
```

Contents

1 Configuration

- SSH Keys
- GPG Keys
- Aliases
- Bonus: Handling multiple identities

2 Complex Operations

- Reset
- Rebase
- Cherry-picking

3 Various topics

- Forking
- Stashing
- Worktrees
- Bisect

Complex Operations

We will look at three commands:

- 1 Reset
- 2 Rebase
- 3 Cherry-pick

Important remarks:

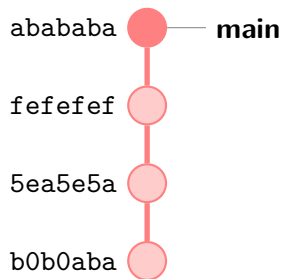
- ! These commands can rewrite history
- ! Should only be used on *private*⁴ branches

⁴branches on which only you are working

```
git reset [<mode>] <hash>
```

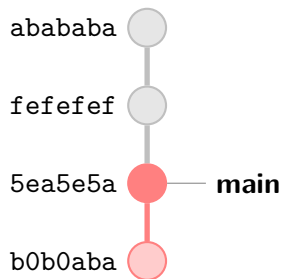
- Always undoes 1 or more commits
- Moves back to the commit hash provided
- Multiple modes
 - ❶ soft - Doesn't touch the index or working tree
 - ❷ mixed - Resets the index (default)
 - ❸ hard - Cleans the index and working tree. Untracked files are deleted

Reset - Example



```
git reset --hard 5ea5e5a
```

Reset - Example

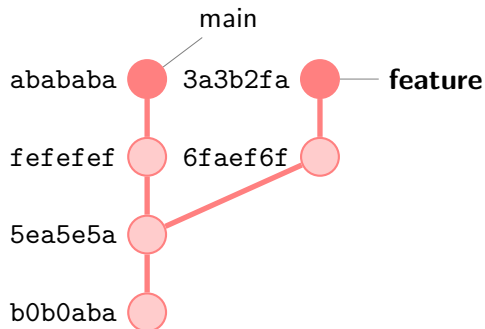


```
git reset --hard 5ea5e5a
```

```
git rebase [--interactive] [--onto <branch>] <branch>
```

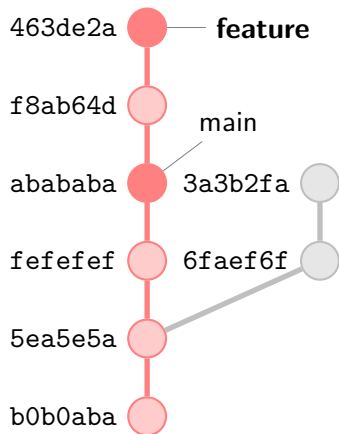
- Gives you a lot of power, so be careful!
- Allows you to
 - rebase a branch (duh)
 - edit commits and their order
 - a lot more

Rebase - Example



Rebase (move the base of) a branch:
`git rebase main feature`

Rebase - Example



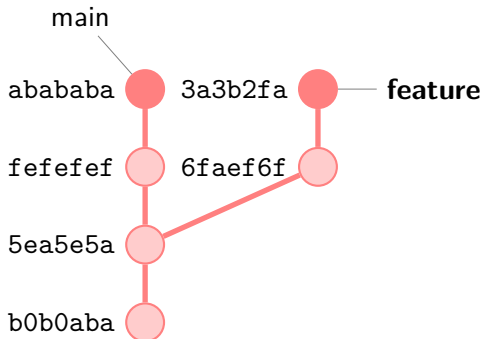
Rebase (move the base of) a
branch:
`git rebase main feature`

Cherry-picking

```
git cherry-pick [--edit] [-x] <hash>
```

- Reapply changes by the specified commits onto the head
- `--edit` allows you to edit the commit message
- `-x` adds a reference ("cherry picked from commit") to the commit message

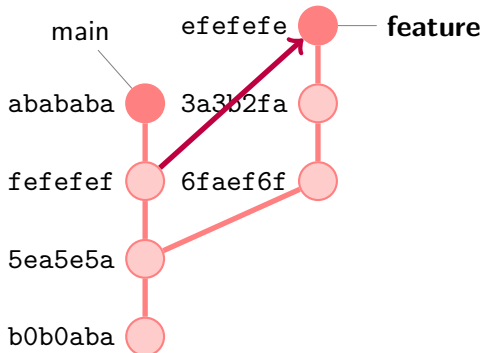
Cherry-pick - Example



Cherry-pick commit fefefef:

- 1 `git switch feature`
- 2 `git cherry-pick fefefef`

Cherry-pick - Example



Cherry-pick commit fefefef:

- 1 `git switch feature`
- 2 `git cherry-pick fefefef`

Contents

① Configuration

- SSH Keys
- GPG Keys
- Aliases
- Bonus: Handling multiple identities

② Complex Operations

- Reset
- Rebase
- Cherry-picking

③ Various topics

- Forking
- Stashing
- Worktrees
- Bisect

- When contributing to an open-source project, you often don't get permissions to create branches and push code to the repository directly.

- When contributing to an open-source project, you often don't get permissions to create branches and push code to the repository directly.
- Instead, you should fork the repository, push your changes there and then create a Pull Request.

- When contributing to an open-source project, you often don't get permissions to create branches and push code to the repository directly.
- Instead, you should fork the repository, push your changes there and then create a Pull Request.
- Another reason to fork a repository is when you want to make changes for yourself and publish it separately.

Stashing

- Saves your uncommitted changes to the side

Stashing

- Saves your uncommitted changes to the side
- Works like a stack (First In; First Out)

Stashing

- Saves your uncommitted changes to the side
- Works like a stack (First In; First Out)
- Commands:
 - `git stash [-m <message>]` - Saves the changes (optionally with a description)
 - `git stash pop` - Moves changes back to your working space *and deletes them from the stash*
 - `git stash drop` - Deletes changes from the stash (*Note: Can cause loss of changes*)

Stashing

- Saves your uncommitted changes to the side
- Works like a stack (First In; First Out)
- Commands:
 - `git stash [-m <message>]` - Saves the changes (optionally with a description)
 - `git stash pop` - Moves changes back to your working space *and deletes them from the stash*
 - `git stash drop` - Deletes changes from the stash (*Note: Can cause loss of changes*)
- Useful when switching between branches while you have uncommitted changes:
 - `git stash` - Save the changes temporarily
 - `git switch <branch>` - Switch to a different branch
 - `git stash pop` - Recover your changes from the stash

Stashing

- Saves your uncommitted changes to the side
- Works like a stack (First In; First Out)
- Commands:
 - `git stash [-m <message>]` - Saves the changes (optionally with a description)
 - `git stash pop` - Moves changes back to your working space *and deletes them from the stash*
 - `git stash drop` - Deletes changes from the stash (*Note: Can cause loss of changes*)
- Useful when switching between branches while you have uncommitted changes:
 - `git stash` - Save the changes temporarily
 - `git switch <branch>` - Switch to a different branch
 - `git stash pop` - Recover your changes from the stash
- If the stashed changes conflict with the state of the other branch you get a merge conflict.
- In this case `git stash pop` will not delete the stash entry to prevent loss of data.

Multiple stash entries

Stash

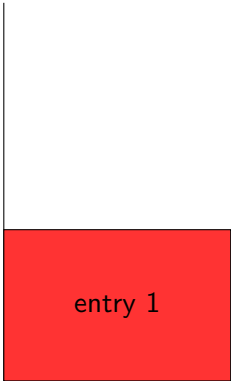
- Let's save something to the stash:



Multiple stash entries

Stash

- Let's save something to the stash:
- `$ git stash`

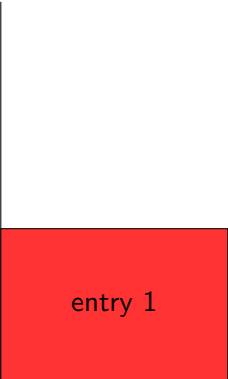


entry 1

Multiple stash entries

Stash

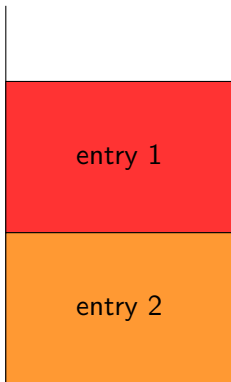
- Let's save something to the stash:
- `$ git stash`
- Let's save a second entry:



entry 1

Multiple stash entries

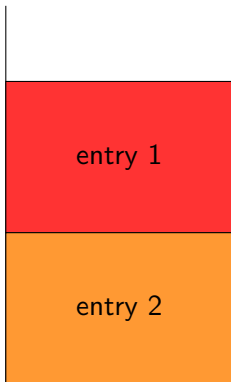
Stash



- Let's save something to the stash:
- `$ git stash`
- Let's save a second entry:
- `$ git stash`

Multiple stash entries

Stash

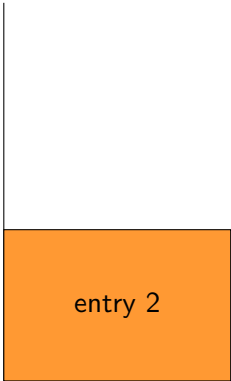


- Let's save something to the stash:
- `$ git stash`
- Let's save a second entry:
- `$ git stash`
- If we now pop something off, we can explicitly choose an entry:

Multiple stash entries

Stash

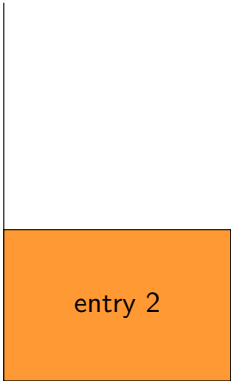
- Let's save something to the stash:
- `$ git stash`
- Let's save a second entry:
- `$ git stash`
- If we now pop something off, we can explicitly choose an entry:
- `$ git stash pop 1 # returns entry 1`



entry 2

Multiple stash entries

Stash



entry 2

- Let's save something to the stash:
- `$ git stash`
- Let's save a second entry:
- `$ git stash`
- If we now pop something off, we can explicitly choose an entry:
- `$ git stash pop 1 # returns entry 1`
- Otherwise we get the entry which was added last (FIFO):

Multiple stash entries

Stash

- Let's save something to the stash:
- `$ git stash`
- Let's save a second entry:
- `$ git stash`
- If we now pop something off, we can explicitly choose an entry:
- `$ git stash pop 1 # returns entry 1`
- Otherwise we get the entry which was added last (FIFO):
- `$ git stash pop # returns entry 2`

- Mounts a branch of your repository in a different directory

- Mounts a branch of your repository in a different directory
- Useful when you need to compare functionality in multiple branches, or work on multiple branches

Worktrees

- Mounts a branch of your repository in a different directory
- Useful when you need to compare functionality in multiple branches, or work on multiple branches
- Commands:
 - `$ git worktree add [--reason <string>] <path> <branch>`
Adds a new worktree at *path* of *branch*.
 - `$ git worktree list`
Lists all current worktrees (including the main worktree).
 - `$ git worktree remove <path>`
Removes the worktree at *path*.
 - `$ git worktree move <path> <new-path>`
Moves the worktree at *path* to *new-path*.

- Helps to find which commit introduced a bug

- Helps to find which commit introduced a bug
- Based on binary search principle

- Helps to find which commit introduced a bug
- Based on binary search principle
- Start by marking one commit
 - bad - contains the bug
 - good - doesn't contain the bug

- Helps to find which commit introduced a bug
- Based on binary search principle
- Start by marking one commit
 - bad - contains the bug
 - good - doesn't contain the bug
- Git will automatically determine the commit halfway and move there

- Helps to find which commit introduced a bug
- Based on binary search principle
- Start by marking one commit
 - bad - contains the bug
 - good - doesn't contain the bug
- Git will automatically determine the commit halfway and move there
- Mark it and Git will move to the next commit accordingly

- Helps to find which commit introduced a bug
- Based on binary search principle
- Start by marking one commit
 - bad - contains the bug
 - good - doesn't contain the bug
- Git will automatically determine the commit halfway and move there
- Mark it and Git will move to the next commit accordingly
- You end up at the first bad commit (introduced the bug)

Bisect

- Helps to find which commit introduced a bug
- Based on binary search principle
- Start by marking one commit
 - bad - contains the bug
 - good - doesn't contain the bug
- Git will automatically determine the commit halfway and move there
- Mark it and Git will move to the next commit accordingly
- You end up at the first bad commit (introduced the bug)
- Stop bisect with `$ git bisect reset`

Bisect - Simple example

HEAD

abababa

bdac853

cafe7e7

ab56da3

fefefef

5ea5e5a

b0b0aba



Bisect - Simple example

HEAD

abababa

bdac853

cafe7e7

ab56da3

fefefef

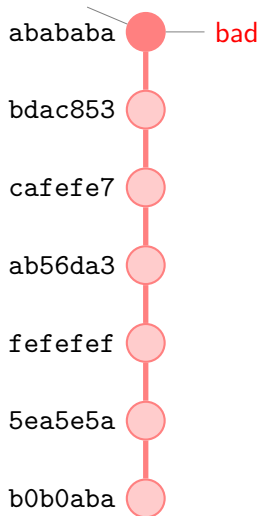
5ea5e5a

b0b0aba

1 \$ git bisect start

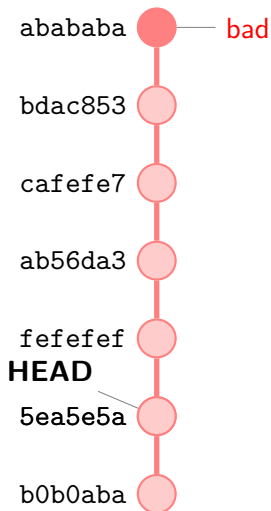
Bisect - Simple example

HEAD



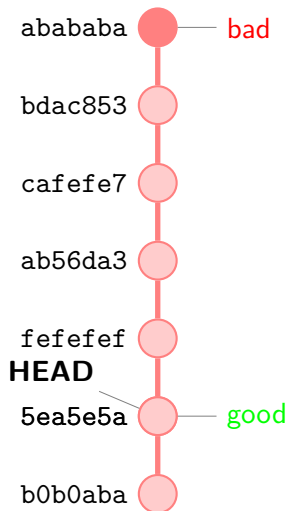
- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`

Bisect - Simple example



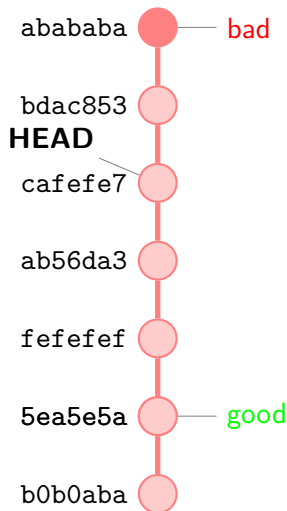
- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`

Bisect - Simple example



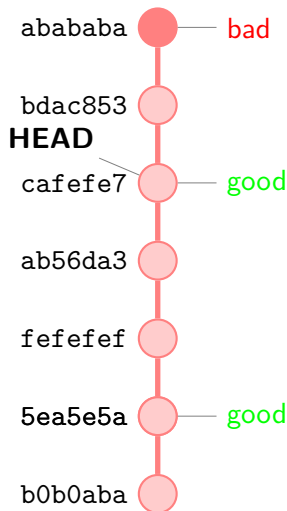
- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`
- 4 Mark the commit as good:
`$ git bisect good`

Bisect - Simple example



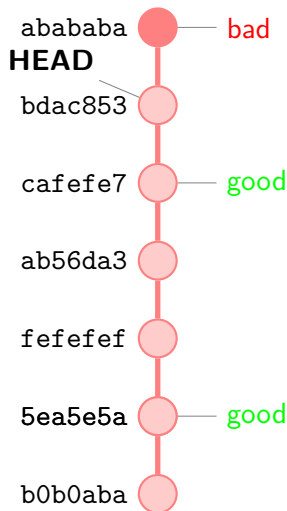
- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`
- 4 Mark the commit as good:
`$ git bisect good`
- 5 Git moves to the halfway point

Bisect - Simple example



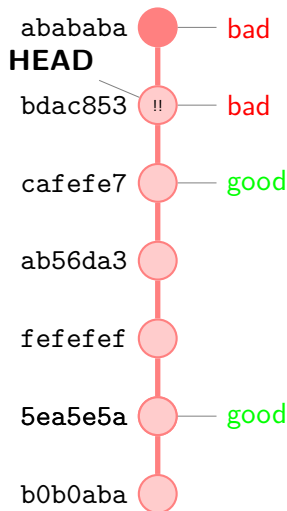
- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`
- 4 Mark the commit as good:
`$ git bisect good`
- 5 Git moves to the halfway point
- 6 Mark it accordingly

Bisect - Simple example



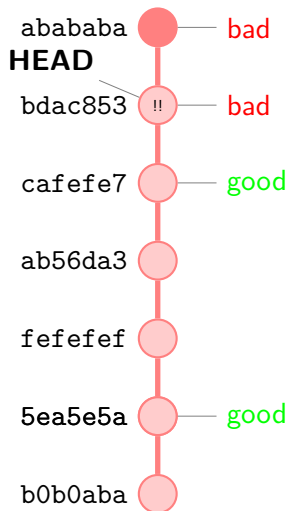
- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`
- 4 Mark the commit as good:
`$ git bisect good`
- 5 Git moves to the halfway point
- 6 Mark it accordingly
- 7 Repeat last two steps until you've found the faulty commit

Bisect - Simple example



- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`
- 4 Mark the commit as good:
`$ git bisect good`
- 5 Git moves to the halfway point
- 6 Mark it accordingly
- 7 Repeat last two steps until you've found the faulty commit

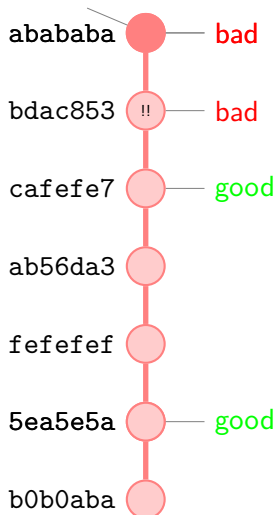
Bisect - Simple example



- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`
- 4 Mark the commit as good:
`$ git bisect good`
- 5 Git moves to the halfway point
- 6 Mark it accordingly
- 7 Repeat last two steps until you've found the faulty commit
- 8 Once done, stop the bisect:
`$ git bisect reset`

Bisect - Simple example

HEAD



- 1 `$ git bisect start`
- 2 Mark the current commit as bad:
`$ git bisect bad`
- 3 Find a commit which doesn't contain the bug, e.g. `5ea5e5a`:
`$ git checkout 5ea5e5a`
- 4 Mark the commit as good:
`$ git bisect good`
- 5 Git moves to the halfway point
- 6 Mark it accordingly
- 7 Repeat last two steps until you've found the faulty commit
- 8 Once done, stop the bisect:
`$ git bisect reset`

Sources for future reference

- `git help <command>`
- gitimmersion.com
- Rebase deep-dive: [Conference talk about rebase \(30m\)](#)

Think of a question later on? Feel free to reach out to us!

MasterCLASS

masterclass@scintilla.utwente.nl

Kasper Müller

kasperm@scintilla.utwente.nl

Johan Verzijden

johanv@scintilla.utwente.nl