Lab: Debugging with Git (bisect)

Software bugs have been a problem for as long as software has existed. As Git records all the commits to the repository then it becomes a great source of information and diagnostic tool when identifying how issues were introduced.

Tutorial Overview

In this scenario we'll explore the different ways you can find which commit introduced the problem. The environment has been initialised with a Git repository containing a single HTML file which renders a list of items.

Pre-reqs:

- Google Chrome (Recommended)

Lab Environment

There is no requirement for any setup.

Run the following command in **terminal**: `cd ~/Desktop/gitlab-intro/lab9/ && mv git .git`

Git Diff Two Commits

The git diff command is the simplest to compare what's changed between commits. It will output the differences between the two commits.

Example

You can visually any two commits by providing the two commits hash-ids or pointers (blobs)

`git diff HEAD~2 HEAD`

Git Log

While git log helps you see the commit messages but by default it does not output what actually changed. Thankfully the command is extremely flexible and the additional options provide useful insights into the history of the repository.

Examples

To see the overview of the commits in a short view use the command `git log --oneline`

To output the commit information with the differences of what changed you need to include the -p prompt such as `git log -p`

This will output the entire history. You can filter it with a number of different options. The -n specifies a limit of commits to display from the HEAD. For example `git log -p -n 2` displays HEAD and HEAD~1.

If you know the time period then you can use a time period to between commits before a particular date using --since="2 weeks ago" and _--until="1 day ago".

As with most Git commands, we can output a range of commits using **HEAD...HEAD~1** as shown in the terminal.

Use the command `git log --grep="Initial"` will output all the commits which include the word "Initial" in their commit message. This is useful if you tag commits with bug-tracking numbers.

Protip

Your commit history can become noisy due to use merge notification commits. To remove them provide the argument -m with git log.

Git Bisect

The **git bisect** commands allows you to do a binary search of the repository looking for which commit introduced the problem and the regression. In this step we'll find the commit which forgot HTML tags in list.html.

Git bisect takes a number of steps, execute the steps in order to see the results.

Steps

1. To enter into bisect mode you use the command `git bisect start`.
2. Once in bisect mode you define your current checkout as bad using `git bisect bad`. This indicates that it contains the problem your searching to see when it was introduced.
3. We've defined where a bad commit happened, we now need to define when the last known good commit was using `git bisect good HEAD~5` . In this case it was five commits ago.
4. Step 3 will checkout the commit in-between bad and good commits. You can then check the commit, run tests etc to see if the bug exists. In this example you can check the contents using `cat list.html`
5. This commit looks good as everything has correct HTML tags. We tell Git we're happy using `git bisect good`. This will automatically check out the commit in the middle of the last known good commit, as defined in step 5 and our bad commit.
6. As we did before we need to check to see if the commit is good or bad. `cat list.html`
7. This commit has missing HTML tags. Using `git bisect bad` will end the search and output the the related commit id.

The result is that instead of searching five commits, we only searched two. On a much larger timescale bisect can save you signifant time.

Git Blame

While having a "blame" culture isn't desirable, it can be useful to know who worked on certain sections of the file to help with improvements in future. This is where git blame can help.

`git blame <file>` shows the revision and author who last modified each line of a file.

Example

Running blame on a file will output who last touched each line.

```
git blame list.html
```

If we know the lines which we're concerned with then we can use the -L parameter to provide a range of lines to output.

```
git blame -L 6,8 list.html
```