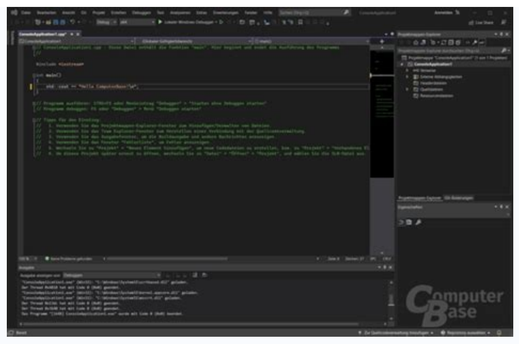
Android studio git blame

Continue



Modern software development relies on Version Control Systems, as many developers manage their codebase and track changes with them. Being the de facto standard of VCSs, Git is used everywhere across technologies. Not surprisingly, Android Studio (and IntelliJ IDEA under the hood) has excellent integration with Git, however this time we are going to focus on a different kind of version control, the version control of your local changes. But wait a minute, the local repo is a full-fledged Git repository, so why we need to talk about local changes? Well, if you have ever ... committed a half-finished solution to Git just to have a safe-state to return to when making changes pushed not-yet-finished work or completely reverted just to merge a different branch made commits like Fixing bugs #12, #22, #7, #149 .... just because you have solved them in one breath ... then I think there is room for improvement with your local change management. Don't get me wrong, I am also guilty of all the things above. This is why I search for better approaches, and I would like to share some with you! From local changes to remote changes Although Git is a distributed VCS, in most cases there is a sort of central remote Git repository, often referred as origin (or the blessed repository). This is the single source of truth, as in most cases this is used by a CI tool, and all developers working on the codebase have a local copy of that repository (or a significant part of it). When you make changes on your local code, these changes affect your working directory. This is not your local repository, just a working copy. Upon commit, these changes are saved into your local repository, only visible to you. When you push your local commits, these changes are uploaded to the remote Git repository. So technically your local repository is also a VCS on its own. Constraints of a commit Commits are changes in our codebase, and a commit is considered to be the smallest unit of work we produce, an atomic transformation of the source code. It transforms our code from one state to another, basically adds and removes lines. One release of the software is an ordered chain of commits, that transforms the initial state to the end product step by step. In most workflows, code that is not committed or not pushed to the blessed repository is not part of the codebase, as it will never get released. Commits also have constraints, as they must transform a correct state of your app to another one correct state. Correct state here means that your code at least compiles and all the tests are passing - you may also have a CI to ensure this. However, as I mentioned in the introduction, there are couple of scenarios that may look like a commit, but they cannot satisfy the above constraints. These should be managed locally by using local change management solutions. Local history Intellj IDEA and Android Studio have a neat feature called Local history. It tracks changes on your working directory, so it's technically not the part of Git. It also enables you to revert any changes in between commits. It even works with lines, files and folders, so it is possible to revert even the whole codebase to a specific point in time. As its name says, it is local, so this history is only visible to you. Therefore, you no longer need to commit a safe state, that for instance compiles, but isn't a complete solution yet. To use Local history on any folder or file press right click, then Local history > Show history. Here you see your local changes on the selected files or folders. On a previous entry, you can hit right click and Revert to revert to that state. Git commit(s) affecting the selected file(s) are also displayed on that timeline, making it easier to navigate through it. You can also label the current state to annotate specific versions in your local history, by right clicking on your source choosing Local History > Put label. These labels are then displayed on the timeline similarly to commits (without the Commit Changes: prefix), so they'll help you find your way back to the marked state later. Changelists One underutilized feature of Android Studio, and also one of my favorites is Changelists. By default, all the changes on your local working copy are part of the Default changelist, which you see in the Version Control panel(⌘+9 or ALT+9 on PC), under the Local changes tab. A changelist is a group of local changes, and it is up to you to split your changes into more changelists, the way you want to. I recommend switching on Group by directory and Expand all on the toolbar on the left for easier navigation, but these are just my preferences - use the configuration that suits you the most. You can create a changelist by right clicking inside Local Changes and selecting New Changelist. Every changelist must have a name, and I suggest using something that describes it well, because if this technique clicks for you - and I hope it will - you may end up using many changelists parallel, and inactive changelists are often in a collapsed state. Also, I suggest that you use a name that is easy to address. Later we will talk about moving changes between changelists, and then you may end up typing in the name of the changelist. If you include a bug tracker number or an issue ticket ID in this name, it will make your life easier. Changes on your local working copy are relative to your current Git HEAD,and changes being made will always be part of the currently active changelist. By default, the Default changelist is active, however when you create a new changelist, you can set it to active. Only one changelist can be active at a given time, which makes sense. There is a very neat feature called Track context, which means that the changelist will be linked to the open editors you're using when working on that changelist. It lets you continue your work where you had left that, with the exact same opened files and cursor(s). Although Track context tracks your open editors, it does not track recent files separately, that feature is global. As I use Android Studio without tabs as Hadi suggests, I navigate with Recent files (⌘+E, or Ctrl+E on PC) all the time, so that would be useful, but others may use it in different ways... You can set any changelist to be the active one by right clicking on that and choosing Set to active, or by pressing Ctrl+Space (Both MAC and PC) when it's selected. Moving changes It's easy to move files changes between changelists: just right click on a file in the local changes tab and select Move to Another Changelist or use ⌘+⇧+M (CTRL+SHIFT+M) and select a changelist. You can also type in the changelist's name, that is why I suggest memorable names. You can also create a new changelist if one with the name you've entered doesn't exists yet. A shiny New! badge will indicate this, which also helps you make sure you didn't just make a typo :D. That is very neat, but it gets more exciting when you start to move changes line by line, not file by file. To manage changes line by line, open the diff of your file from your local changes tab, by right clicking and choosing Show Diff or ⌘+D (CTRL+D). This will show you the changes in the file. After right clicking on a changed line - on the content itself, and not the line numbers - you can select Move to another changelist or press ⌘+⇧+M (CTRL+SHIFT+M), and move these changes, just like you can with entire files. One thing to keep in mind is that the IDE tracks contiguous blocks of changes (hunks) instead of individual lines, so technically, these are what you can move around. This can be confusing if two unrelated changes are next to each other, but it is more likely that changes affecting multiple lines next to each other are part of the same logical change (this is why versionCode and versionNumber changes are considered to be only one change on the screenshot). Utilizing changelists Committing changelists When we have organized our changes into changelists there are multiple things we can do with them. We can of course commit changelists one-by-one. To do that, select a changelist and press ⌘+K (CTRL+K). It will pop up the commit window with the changelist selected, and by default the commit message will be the name of the change list. However, if there is a comment added to the changelist, that will be the commit message instead. An inherent effect of thoughtful naming of changelists is that you will no longer write messages upon commit, as you have defined the scope of your commit when you named your changelist. For me, this led to much better commit messages overall. Shelving and stashing changes When you are in the middle of something and need to switch branches, you may want to just put away your current work for later, save that work-in-progress state somehow. As it does not feel like a real commit, and since it's inconvenient to revert back and forth repeatedly in local history, you must use something different. We have two options to handle that situation: shelving and stashing. Shelving Shelving lets you save changelists into a separate local storage. It is a feature of IDEA and independent of Git. You can select your changelist and hit Shelve changes, and then track shelved changes under the Shelf tab. When a changelist is shelved its contents are not just saved, but also detached from your current work, so if you continue to work on that changelist, it won’t be tracked by that shelved saved state anymore. However, you can shelve the same changelist twice, with a different name and state. When you want to continue your work, just right click a shelved changelist under the Shelf tab, and select Unshelve... or hit ⌘+⇧+U (CTRL+SHIFT+U) when it's selected. This is the recommended approach to handle the switch branches situation, when you need to save in-progress work. Stashing Stashing, on the other hand, is a feature of Git. Stashing is similar to shelving, however currently IDEA only supports stashing the whole working copy, so stashes cannot benefit from changelists. Another major difference is that Git stash metadata is saved in the .git folder, which is not tracked by Git, while IDEA’s shelf is saved under .idea/shelf/ in a .patch format. This makes it much easier to distribute shelved changes. For switching machines or passing a work-in-progress solution to someone else, a distributed shelf could be a considerable option. In general, I would recommend using shelf over stash! Unless you use another IDE in tandem with Android Studio, which does not support IDEA’s shelf, I cannot think of another use case where I would prefer the latter. Pros and cons Following the above-mentioned techniques will result in many benefits such as: Having a much cleaner Git repository in general. Reverting changes is easier. Cherry-picking features is easier. No more work-in-progress commits. Smaller chucks of code produced, which are much easier to code review. However, it will have some drawbacks to keep in mind: An inevitable increase in the number of commits - Automate your CI to run on every push (with multiple commits possibly) instead of every commit. Work in progress changes are only local - in case of a hardware failure or damage they are lost. Changelists are not isolated - meaning that when your current local code compiles and works as expected, the individual changelists may not. To overcome this, I suggest shelving all non-active changelists before commiting to test your changelist in isolation. Although I have to admit that this often requires too much effort, and is therefore skipped sometimes. Shelved changes are not relative to a specific commit, so when you unshelve them, you may face conflicts. Conclusions After all, I believe these techniques will improve your code quality and team work. I have used these techniques in the past couple years for many projects, and they've proven to be very useful, so I hope you will also benefit from them! Thanks for my coworkers for their review of this article, and your feedback is also welcome! You can reach me on Twitter at @itsbata. Follow us on social media