**#1 The Perfect Commit.**

1. **Add the *right* changes!**

Our goal here is to create a commit that makes sense, one that only includes changes from a single topic, and in contrast to the easy way when sometimes just cram all our current local changes into the next commit. This is something we should not do.

Being selective and carefully deciding what should go into the next commit is important. The bigger a commit gets and the more topics that are mixed into the commit, the harder it gets to understand both for your colleagues and for yourself in the future.

**Golden rule of version control:** Only combine changes from the same topic in a single commit.

1. **Compose a *good* commit message!**

**The Perfect Commit Message**

1. **Subject = concise summary of what happened.**

Generally, the advice is to write something very concise, less than 80 characters if possible. And the subject should be a very brief summary of what happened.

If you have trouble writing something short and concise, then this might be an indication that you’ve put too many different topics into the commit.

1. **Body = more detailed explanation**

* What is now different than before?
* What’s the reason for the change?
* Is there anything to watch out for / anything particularly remarkable?

**#2 Branching Strategies.**

**Agree on a Branching Workflow in your team.**

1. Git allows you to create branches, but it doesn’t tell you how to use them!
2. You need a written best practice of how work is ideally structured in your team – to avoid mistakes and collisions.
3. It highly depends on your team / team size, on your project, and how you handle releases.
4. It helps to onboard new team member (“this is how we work here”)

**Integrating Changes and Structuring Releases.**

1. **Mainline Development (“Always be integrating”).**

Always be integrating mainline development, always integrate your own work with the work of the team. That’s the motto.

* few branches
* relatively small commits
* high-quality testing and QA standards

1. **State, release and Feature branches.**

Branches enhance structures and workflows.

* different types of branches fulfill different types of jobs

When multiple different types of branches enter the stage.

Here branches are used to fulfill different jobs. New features and experiments are kept in their own branches, releases can be planned and managed in their own branches.

**Low-Running and Short-Lived Branches.**

**Two main types of branches.**

About the long running branches, every Git repository contains at least one long running branch typically something called main or master.

Branches all have something in common; they exist throughout the complete lifecycle of the project.

Another type of long running branches are so called develop branch. Typically, these branches represent states in a project release or deployment process. If your code moves through different states, for example, from development to staging to production, it makes a lot of sense to mirror the structure in your branches too.

And finally, many teams have a convention connected to long-running branches. Typically, commits are never directly added to these branches, commits should only make it to the long-running branch through integration. In other words, through a merge or rebase.

* Exist through the complete lifetime of the project
* Often, they mirror “stages” in your dev life cycle
* Common convention: no direct commits!

The other type of branches are short-lived branches. In contrast to long-running branches, they are created for certain purposes, and then deleted after they have been integrated. There are many different reasons to create live branches. For example, when you start working on a new feature, a bug fix or refactoring or an experiment.

* For new features, bug fixes, refactoring, experiments…
* Will be deleted after integration (merge/rebase)

Choosing these strategies highly depends on your preferences or team size, or type of project.

**Two example branching strategies.**

**GitHub Flow.**

GitHub advocates a workflow that’s extremely lean and simple. It only has a single long-running branch, the default main branch and anything you’re actively working on is done in a separate branch, a short left branch, no matter if that’s a feature, a bug fix or a factoring.

**GitFlow.**

Offers a bit more structure but also more rules to follow. The main branch is a reflection of the current production state. The other long-running branch is typically called develop, which is the starting point for any new releases.

* More structure, more rules.
* Long-running: “main” + “develop”
* Short-lived: features, releases, hotfixes.

**The “best” branching model?**

* Consider your project, release cycle, and team.
* Take inspiration from existing models (like “GitFlow” or “GitHub Flow”)
* Create your own model

If you ask different teams, you will get different answers. There is no perfect branching model that everyone should adopt. It’s more about understanding your project, your release workflow and your team, and then modeling a branching workflow that supports you in the best way possible.

**#3 Pull Requests.**

They are a way to communicate about code and review it. The perfect example is when you’ve finished working on a feature, without a pull request, you’d simply merge your count into main, master, or some other branch. And in some cases, this might be totally fine.

With pull requests you can invite other people to review your work and give you feedback before merging.

**Contributing code to other repositories.**

Think of a popular open source repository, you might have an idea for improving something, but you’re not allowed to push commits. This is another use case for pull requests.

A fork is your personal copy of git repository. You can fork the original repository, make changes in your forked version and open a pull request to include those changes into the original repository. And one of the main contributors can then review your changes and decide to include them or not.

**#4 Merge conflicts.**

**How and when conflicts occur.**

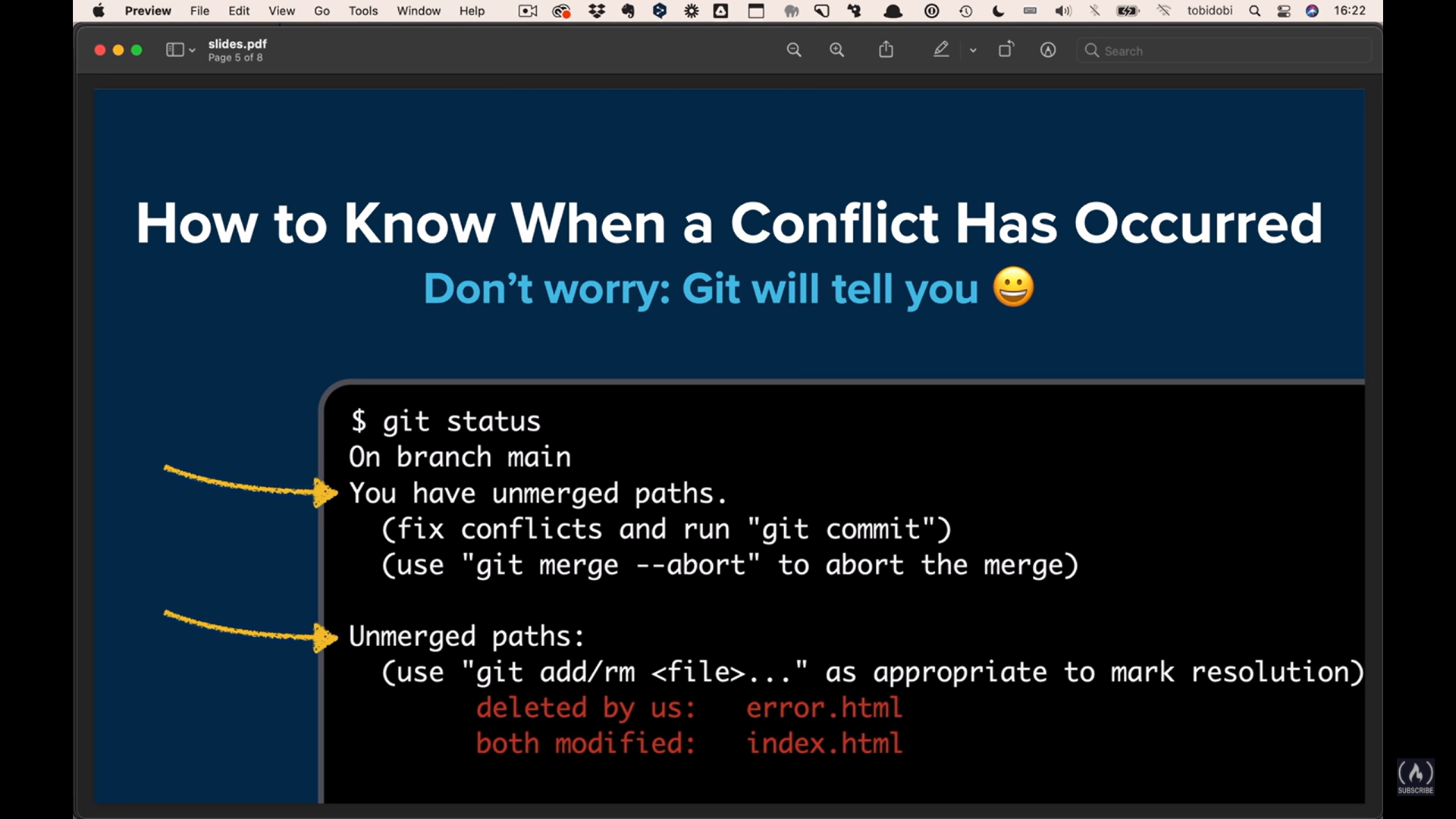
They occur when integrating commits from different sources. Conflicts can also happen when rebasing interactive, when performing a cherry pick or a pull, or even when reapplying a stash. All of these actions perform some kind of integration and that’s when merge conflicts can happen.

**When contradictory changes happen.**

Merging branches works effortlessly most of the time, because Git is usually able to figure things out on its own, but there are situations where contradictory changes where made, and that’s when technology simply cannot decide what’s right or wrong. These situations require a decision from a human.

**How to know when a conflict has occurred.**

Git will tell you.



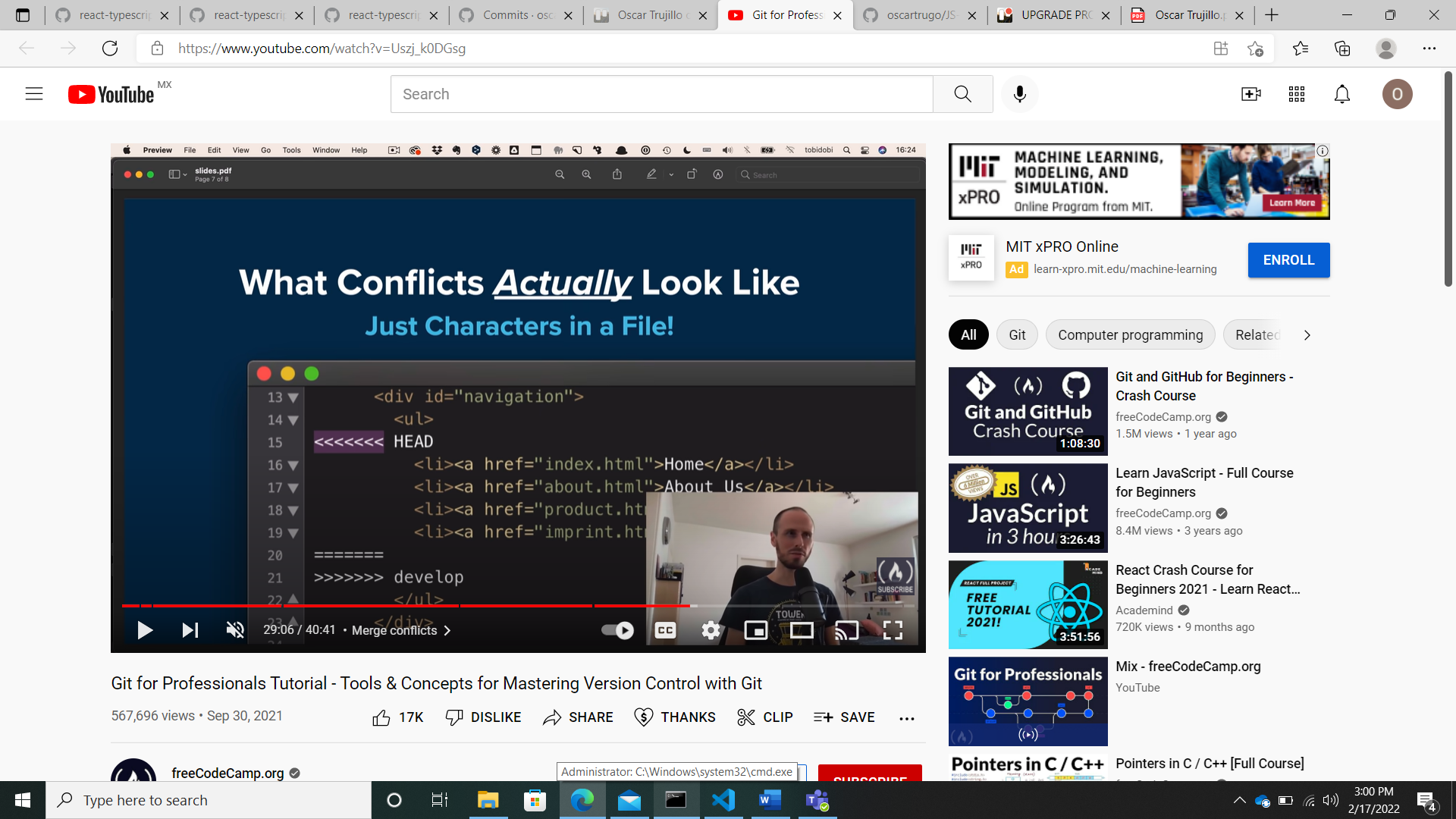
**How to undo a conflict and start over.**

You can always undo and start fresh.



**What conflicts Actually look like.**

Just characters in a file.



**How to solve a conflict.**

Simply clean up the file.

**#5 Merge vs Rebase.**

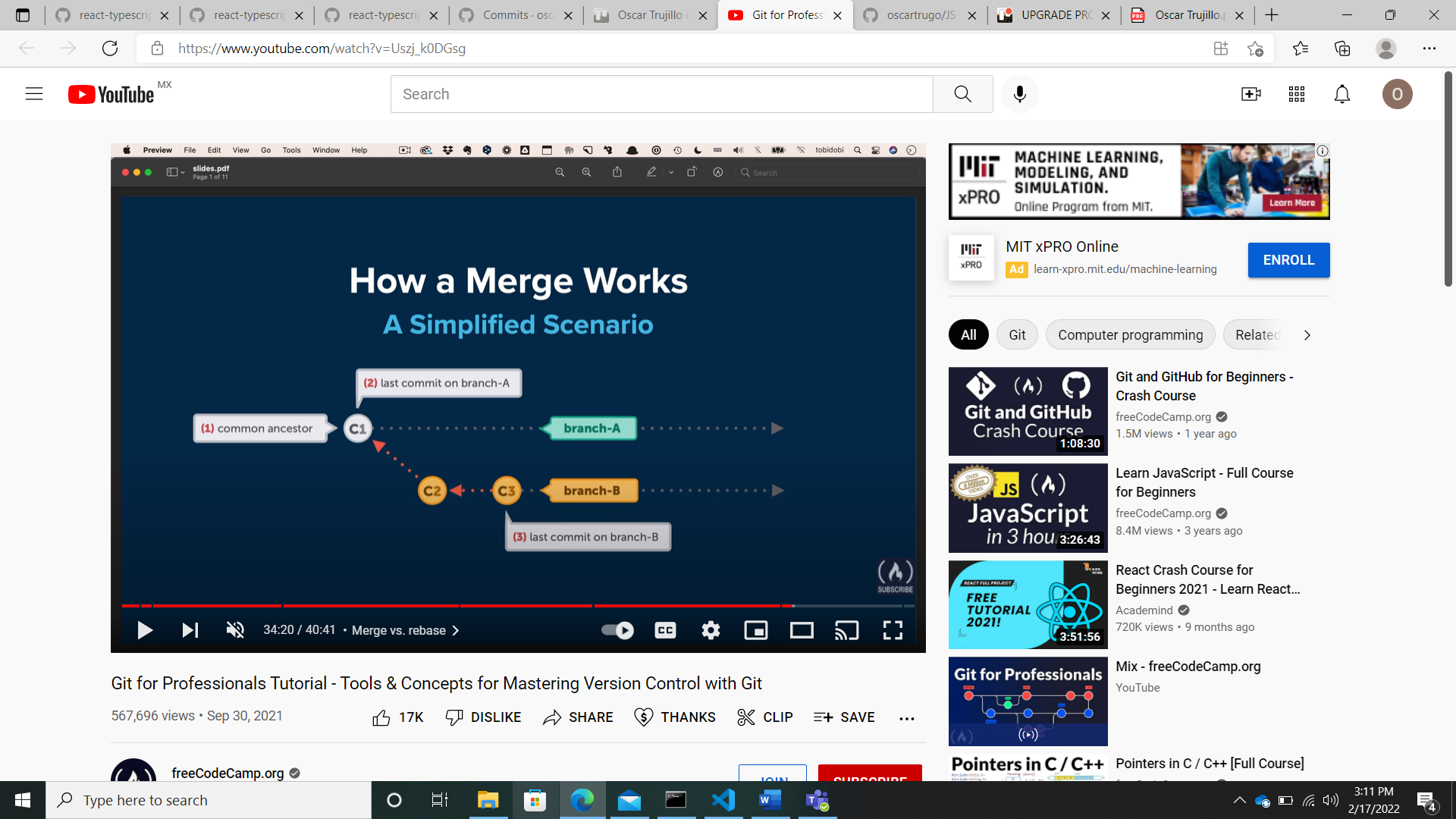
Talking about getting your new code back into an existing branch, there are different ways to do this, and two of the most common ones are merge and rebase.

**Merge.**

When Git performs a merge, it looks for three commits.

* First, the common ancestor commit.
* Second, the last commit on branch -A.
* Third, the last commit on branch-B.

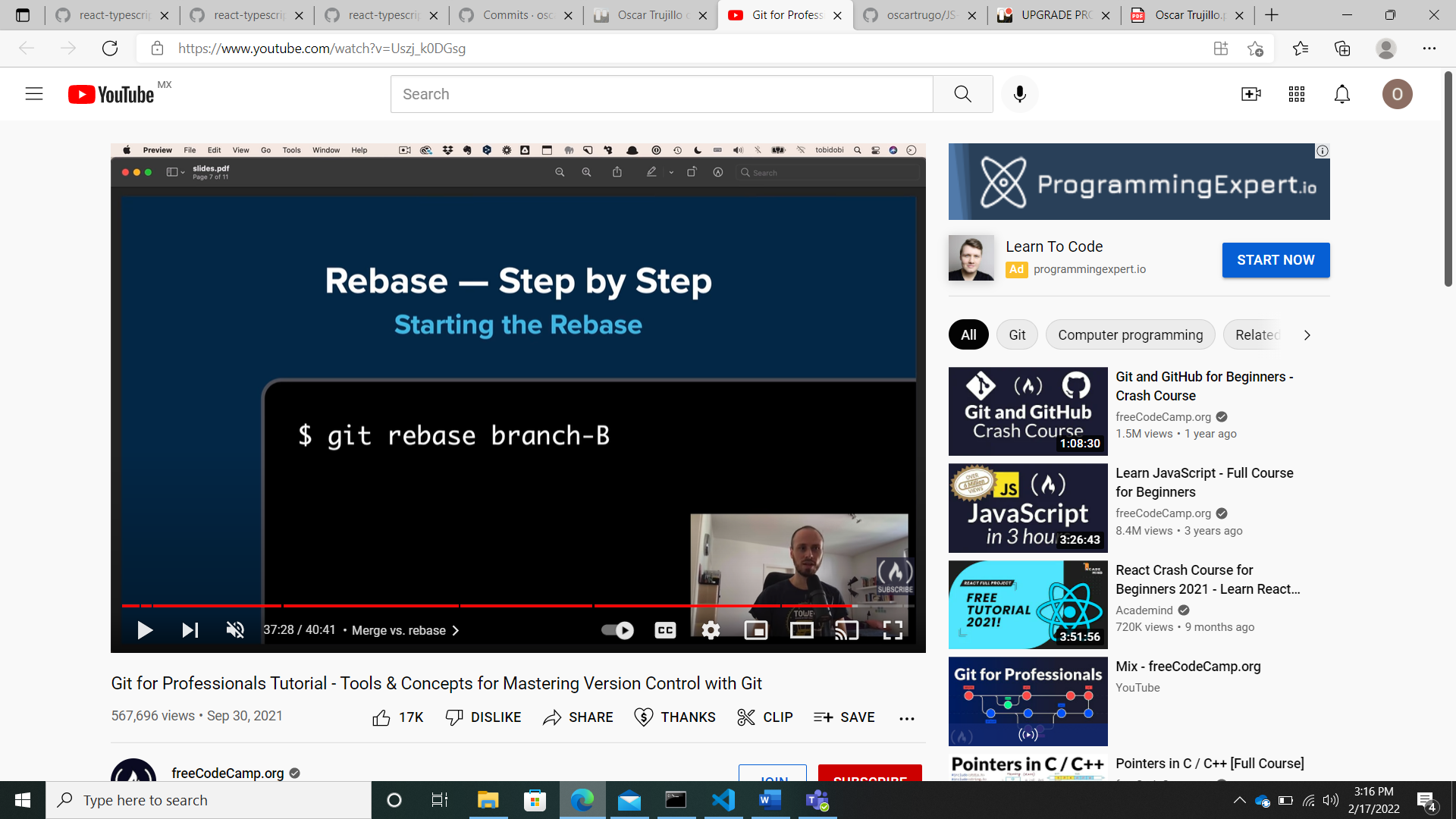
Combining these three commits will perform the integration that we’re aiming for.



**Rebase.**

We want to integrate changes from branch B into branch A, but now using Rebase.

The actual git command to start this is simple, it’s “Git rebase”.



**Step by step.**

First, git will remove all commits on branch A that happened after the common ancestor commit. But it won’t throw them away, they will be saved somewhere temporarily.

Then, Git applies to new commits from branch B, and at this point, temporarily, both branches look exactly the same. But in the final step, those parked commits need to be included.

The new commits from branch A, they’re positioned on top of the integrated commits from branch B, they are rebased, and the result looks like development had happened in a straight line, there is no merge commit that contains all the combined changes.

Rebase also rewrites commit history.