Git, GitHub & Maven

Distributed version control system

Git Overview

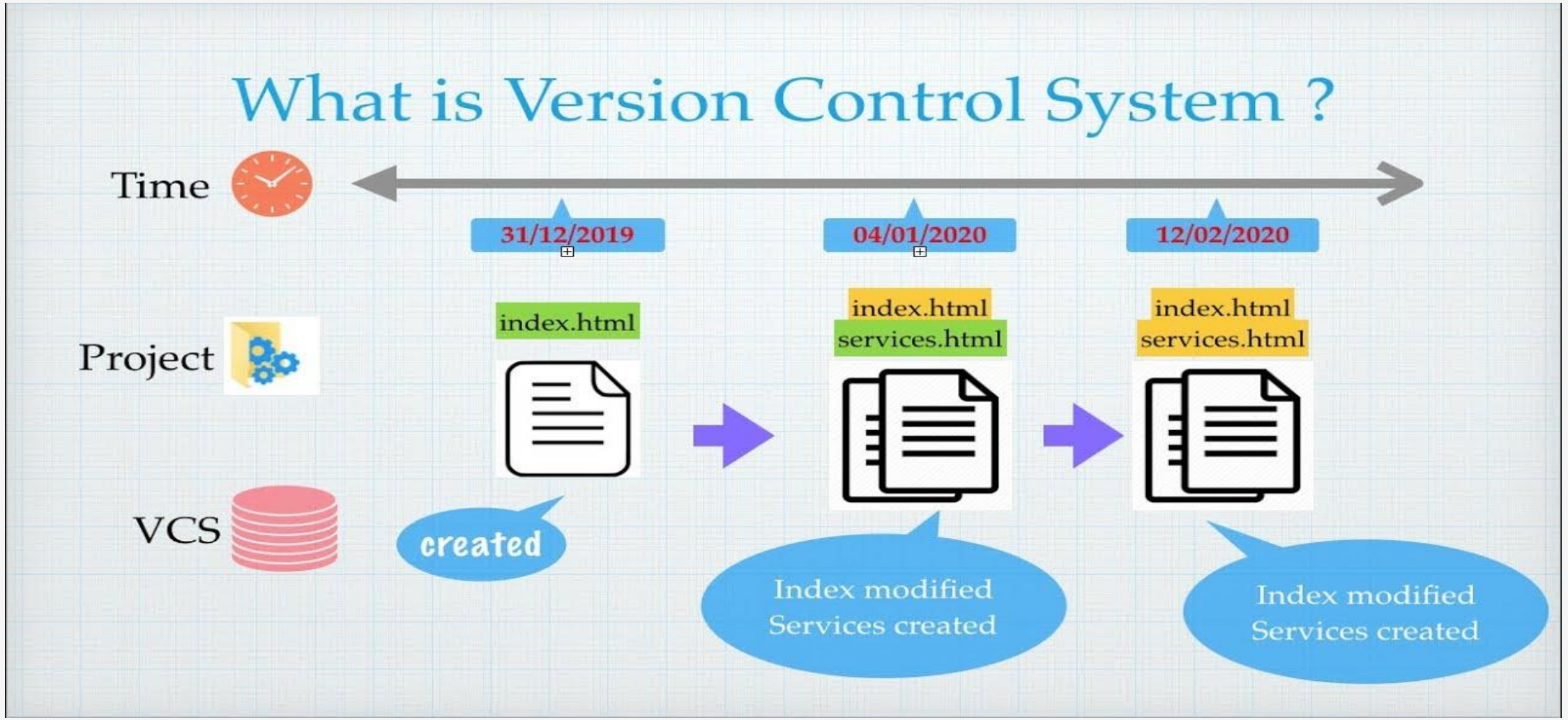
In this session, you will learn:

- Distributed Version Control basics
- Basic Git commands
- Core concepts of Git like:
  - Creating new features without affecting the current working version (Branching)
  - Collaborating with other developers (clone, push, rebase and merge conflicts)
  - Saving new changes temporarily (Stashing)
  - Selecting particular changes from others

By the end of this session, you will be able to use Git.

Let’s get started!

What is Version Control System?



What is Version Control ?

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

By using Version control in your project, you can easily find out:

- **What** are the changes made
- **Which** files are changed
- **When** were the changes made
- **Who** has made the changes

Why Use a Version Control System?

A Version Control System (VCS) gives you many super-powers that help you track changes and secure your source code such as,

- **Collaborating with big teams** smoothly to develop code
- **Reverting the code** to the bug-free version in case any minor changes in the new version introduces a bug.
- Providing **backup** when the central server crashes
- Rendering an option to **time-travel your project** and go to a specific version

Happy Learning ☺

Types of Version Control Systems

- Local: It allows you to copy files into another directory and rename it (for example, project1.1). This method is error-prone and introduces redundancy.
- Centralized: All version files are present in a single central server. For example, CVS, SVN and Perforce.
- Distributed: All changes are available in the server as well as in local machines. For example Git and Mercurial.

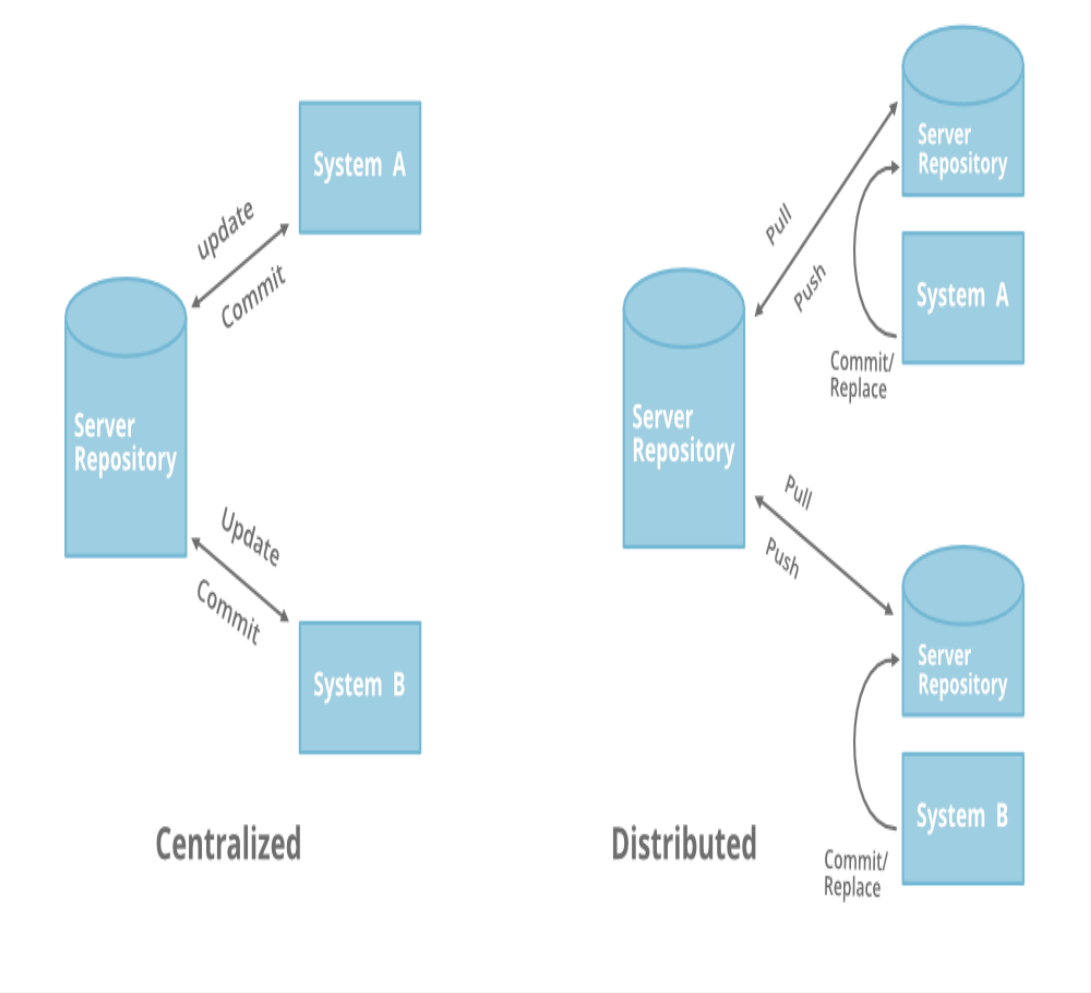
Happy Learning ☺

Centralized vs Distributed Version Control Systems

Centralized	Distributed
You can keep changes only in the server	You can keep changes locally (commit) as well.
Changes can be merged in the server(remote) alone	Changes can be merged locally as well as remotely
Work gets interrupted if the server is down.	The project is present with all the team members to work locally.

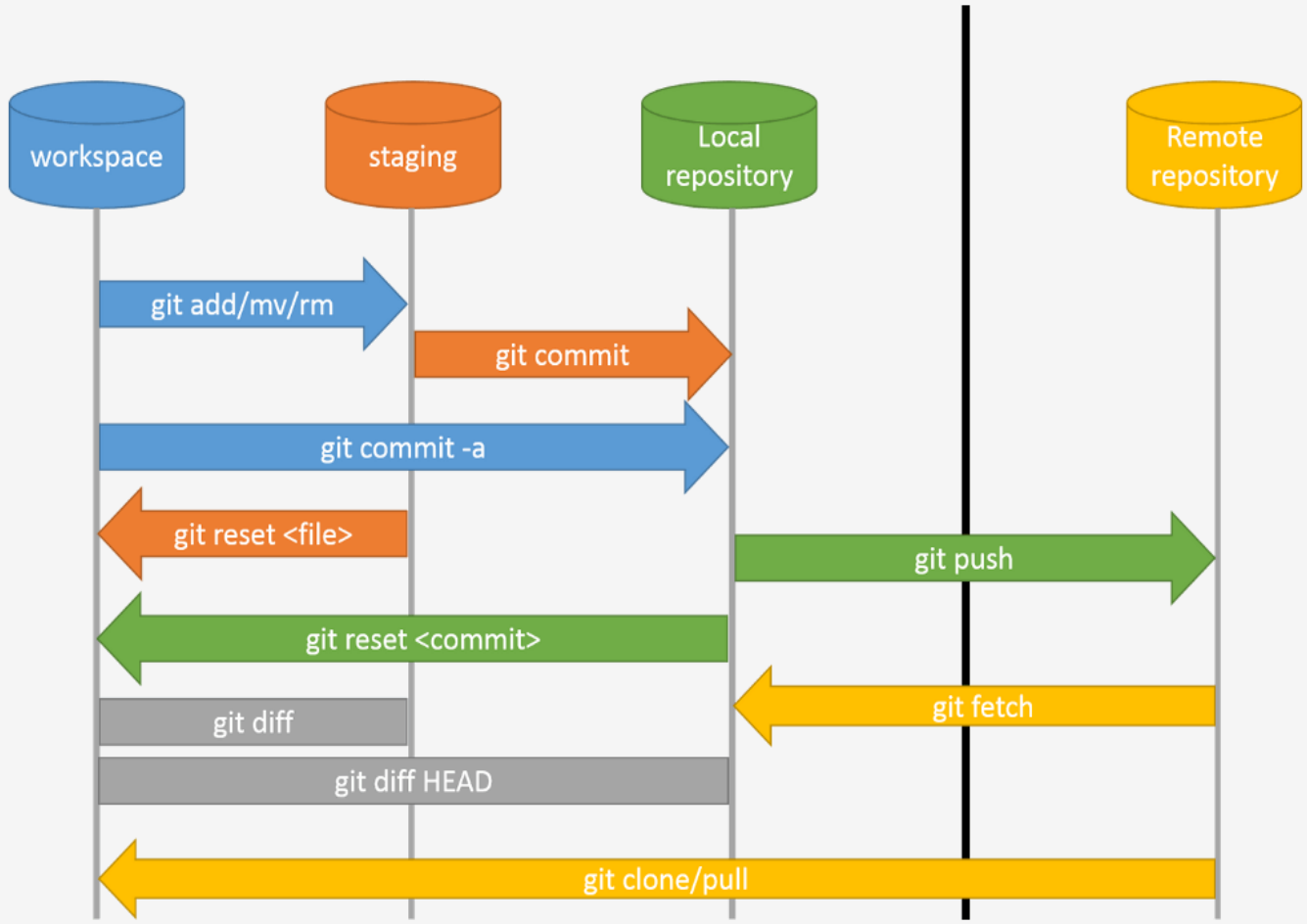
Git is the widely popular Distributed Version Control System which you will be learning in the session.

Are you ready?



Cheat sheet

- git clone : Get the complete project from remote to your local machine.
- git pull origin <branch-name> : Get the new changes from remote branch to local branch.
- git push origin <branch-name> : Send your local branch changes to the remote branch.
- git remote add <name> <url> : Add a new remote repo link to your local repo.
- git remote –v : list all the remote repo URLs linked to your local repo



Git Basics

1. Create new GitHub repo
2. Push code to this repo using git
3. Go over these git commands,
    1. git add
    2. git commit
    3. git push
    4. git difftool

How to Install Git?

For Windows

https://git-scm.com/download/win

For Mac

https://git-scm.com/download/mac

For Linux

Type the following commands from the Shell.

```
$ sudo apt-get update

$ sudo apt-get install git
```

After successful installation of Git, you can configure your user name and e-mail ID using the following commands,

```
$ git config --global user.name “Firstname Lastname”

$ git config --global user.email “myemail@gmail.com”
```

Git Terminologies

Before starting with the basics, let's explore a few terminologies:

- Git Repository: A directory with .git *folder*, where all the contents are tracked for changes.
- Remote: It refers to a server where the project code is present. For example, GitHub and Gitlab.
- Commit: It is similar to version. When you make changes in a file, you need to *commit the changes* in order *to save and create its new version*, which will create a unique commit hash. (like version number)
- Origin: It is a variable where Git stores the URL of your remote repository. For example, origin => *https://www.github.com/username/myrepo*

Git Basic Commands

Before starting with the basics, let's explore a few terminologies:

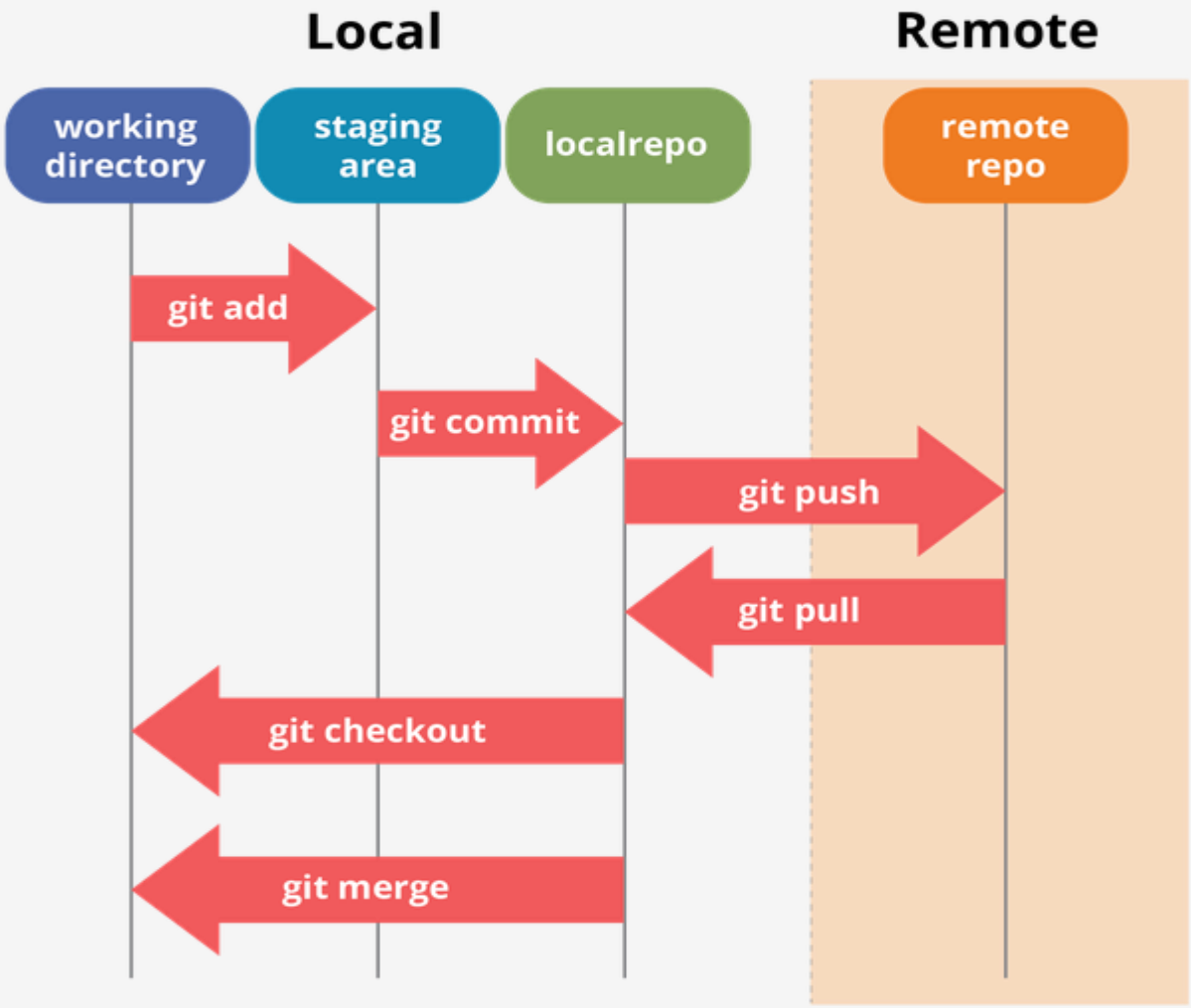
- *git init* adds .git *folder*, and initializes the current folder to track its changes.
- *git status* displays the current state of the staging area and the working directory, that is which files are added/removed/modified.
- *git diff* shows the exact changes with line and column number
- *git add* adds the changes to the staging area. If you have added a new file, this command starts tracking the file for modifications.
- *git commit* will save all the changes with a unique hash number in the local repository
- *git push* sends the changes to the remote repository (server)

Stages in Git

The following are the three stages in the Git workflow:

- *working area* you can **edit *files*** using your favorite editor/integrated Development Environment.
- *staging area* you have made the changes and added the changes to Git. You can still make changes here. (From the analogy explained in the next card). It is like taking an item out of the box, where the box is the staging area. (*git add*)
- *local repository* You have finalized the changes and committed them with a new hash and proper message. (*git commit*)

Remote Repository you can now push the changes to online platforms like Github or Gitlab from where others can collaborate.(*git push*)



Git Stages: Analogy

Git works in three stages known as The Three Trees: Working Area, Staging Area and Local Repository which means Git maintains three states of a file. To understand this better, let us take an analogy of a box.

Assume you are packing items in the house in different boxes and labeling them to identify it later.

- Get a table and keep all the items than you want to pack underneath.(*git init*)
- Now you might select specific kitchen items, dust them and club similar items (like spoons) together. (doing changes – working area)
- Add the items that are ready to the box. (*git add* - Staged)
- Seal the box and add a label – "Kitchen Items". (*git commit* - Committed)

After *git commit* a unique hash is created and the changes are saved.

Git Basic Commands: add, commit, push

Git works in three stages known as The Three Trees: Working Area, Staging Area and Local Repository which means Git maintains three states of a file. To understand this better, let us take an analogy of a box.

Assume you are packing items in the house in different boxes and labeling them to identify it later.

- Get a table and keep all the items than you want to pack underneath.(*git init*)
- Now you might select specific kitchen items, dust them and club similar items (like spoons) together. (doing changes – working area)
- Add the items that are ready to the box. (*git add* - Staged)
- Seal the box and add a label – "Kitchen Items". (*git commit* - Committed)

After *git commit* a unique hash is created and the changes are saved.

Git Ignore or Keep

If you **do not want Git to any file/directory**, you can add the file/directory to .gitignore file.

To track an empty directory, you need to add .gitkeep to that empty directory, as Git ignores them by default.

Git Basic Commands: add, commit, push

1. Create new github repository.
2. Push code to this repository using git
3. Go over these git commands,
    1. git add
    2. git commit
    3. git log
    4. git push
    5. git difftool

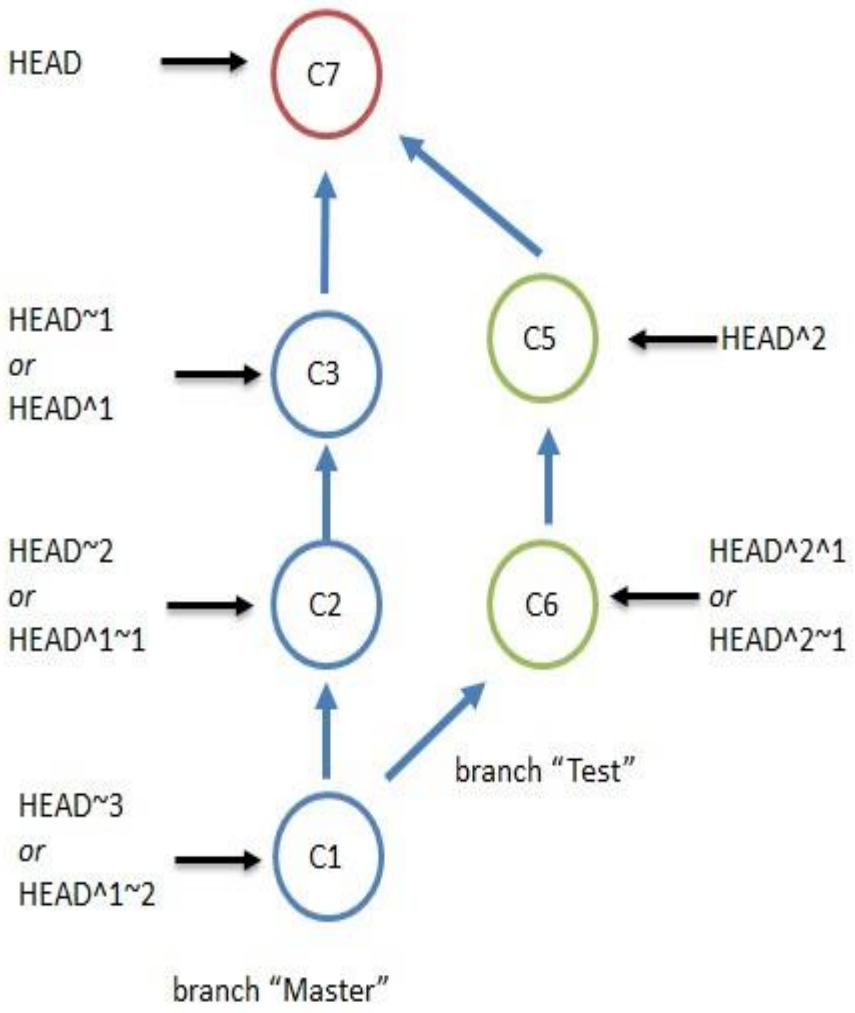
Git History

Git provides the following commands, which allows you to **read and review your repo's history**:

- git log
- git show
- git diff

Before learning more about these tools, you must first understand the HEAD and Dot operators.

What is HEAD in Git?

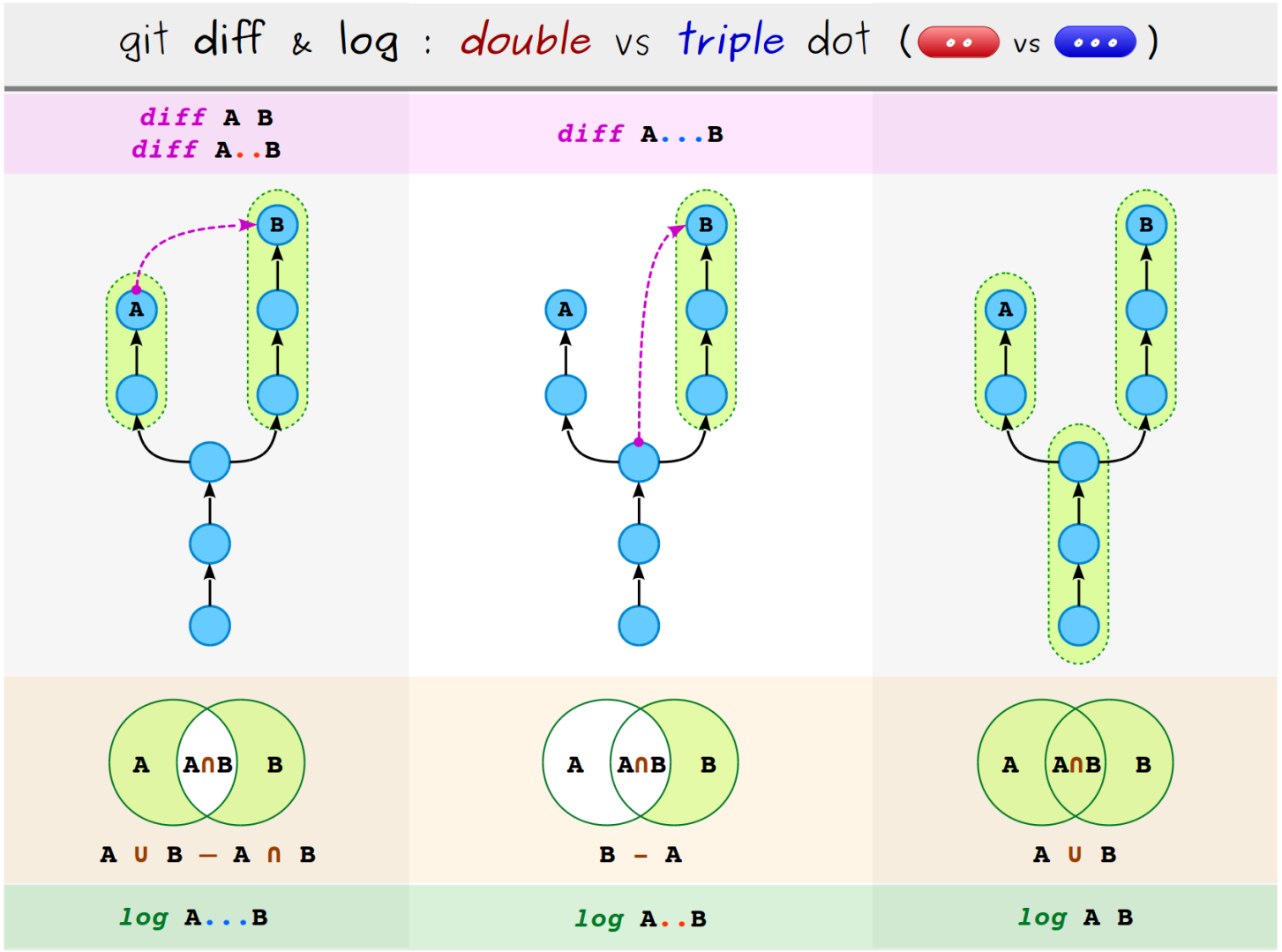


HEAD is a reference variable that always points to the tip of your current branch, that is recent commit of your current branch.

HEAD can be used with the following symbols to refer to other commits:

- Tilde symbol (~): used to point to the previous commits from base HEAD
- Caret symbol (^): used to point to the immediate parent commit from the current referenced commit.
- HEAD means the recent commit of current branch -> C7

Dot Operators



Double Dot Operators

- It is the default operator in git diff
- git diff a..b or git diff a b command will display all the differences from A to C

Trible Dot Operators

- It shows the differences between master and feature branch starting at the last common commit.
- git diff A...B commands output would be the difference in B branch.

Git Log

Git log command shows the list of commits in the current branch. You can use it in the following ways:

- *git log -2* displays the history of last two commits
- *git log commit_id* shows the history starting from commit_id
- *git log filename* displays the list of commits for the file.

Flags

You can enhance the output of git log command using these optional flags:

- *git log --oneline :* Fits the log output to a single line
- *git log --decorate :* Adds a symbolic pointer to the output
- *git log --graph :* Gives a graphical representation to the log output.

Collaborate Using Git

Local to Remote

To collaborate with other developers, you need to push your work to the remote repository, and vice-versa you need to pull others work from remote to contribute your work to the project.

In Git, remote is a repository on a server where all your team members can place the code to collaborate.

Remote URL Types

You can keep your project code in remote servers like GitHub, GitLab or on a self-hosted servers. Git sets a default name for your remote URL as origin

Remote URL can be one of the two types:

- HTTPS URL like https://github.com/user/reps.git : you can clone/push using your username and password.
- SSH URL, like git@github.com:user/repo.git : you need to configure SSH keys in Github/Gitlab account and local machine.

Git Clone

To get source code of an already existing project from remote repo (For example, GitHub) you can use

git clone <url> command

For example, git clone https://github.com/kannanfsd/accdevb1.git

This command downloads the complete project, all branches, commits and logs from the given remote URL (react repo here) to your local machine.

Git Pull

Your teammate hash pushed the changes to the project's remote repo where you are also working. You can now pull the changes to your local machine using any one of the following commands.

- git pull is the convenient shortcut key to fetch and merge the content
  - git pull <remote_name> <branch_name>
- git fetch command downloads the remote content to your local repo, without changing your code changes.
  - git fetch <remote_name> <branch_name> fetches the content from that specific branch in remote to your current working area.
- git merge command merges the fetched remote content to the local working tree.
  - git merge <remote_name>/<branch_name> merges the content to the specified branch.

Git Push

To keep your changes and work in remote repo, you need to push the branch using the command
git push <remote_name> <branch_name>

Git push takes two arguments, namely:

- <remote_name>
- <branch_name>

For example, git push origin master

- origin will contain the remote URL
- master is the branch that is pushed.

Working with Existing Projects

Mr. Max is new to the project. For a particular task, he created ten new files in his local machine. His technical lead said there is a common repo where all the team members place their code. He asked to Max to push his files to the same repo. What should Max do?

In such a scenario, you can connect your local repo with an existing remote repo using git remote add command.

Git Remote

The syntax to link your local repo with remote repo is

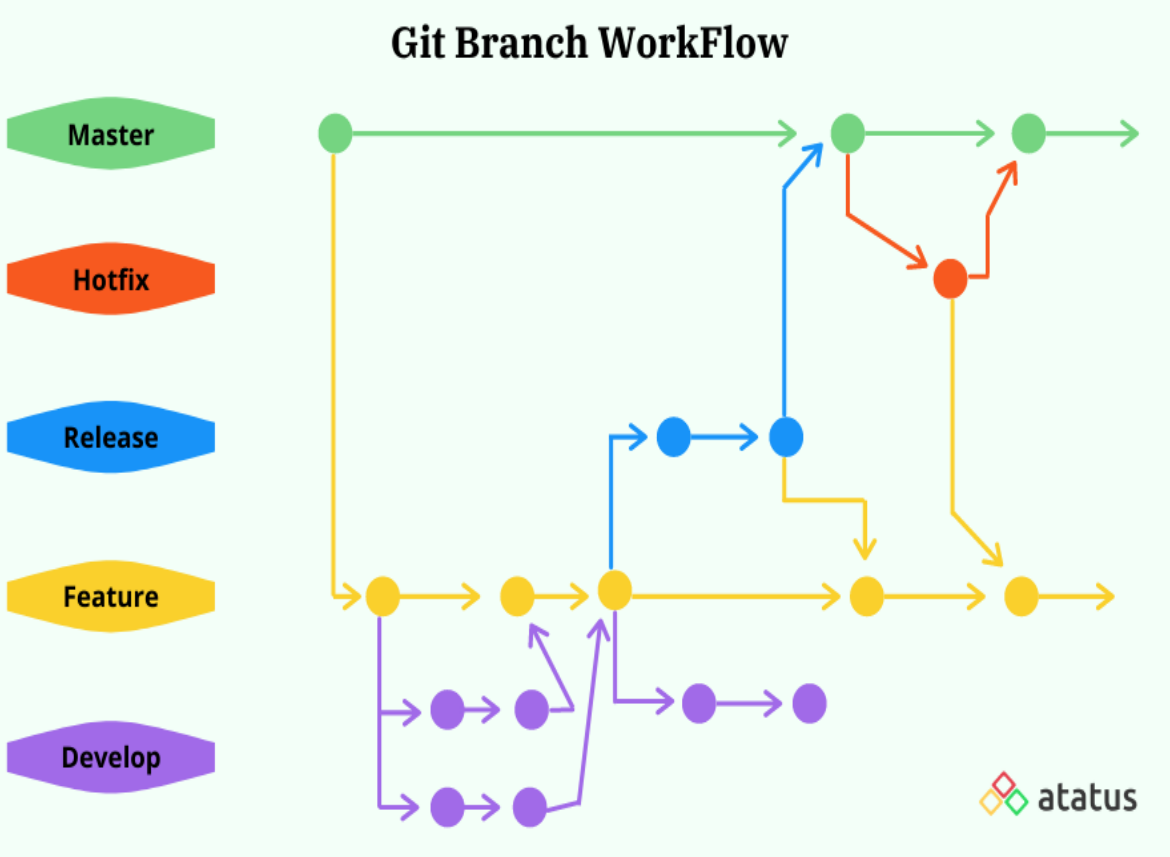
git remote add <remote_name> <remote_url>

It takes two arguments, namely:

- <remote_name>, let us take default name origin
- <remote_url>, let us take https://github.com/kannanfsd/repo.git

For example: git remote add origin https://github.com/kannanfsd/repo.git

Branches



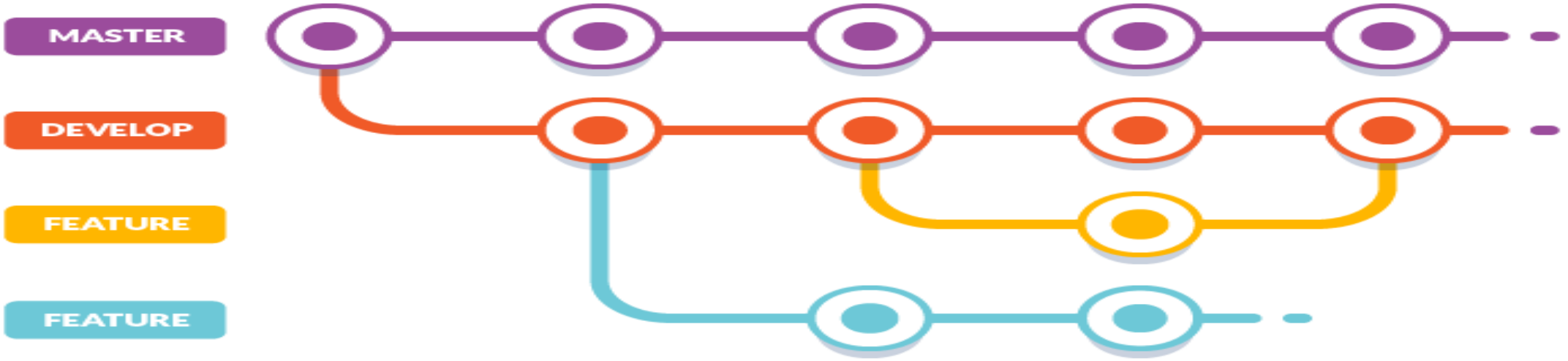
A branch is a **copy of your complete project**, where you can add new changes and develop new features. Technically, it is a collection of commits.

When you create **a new Git project**, it has a branch called master by default.

How to Build Features?

To build new features without affecting the current working code, you need to:

- Create new branch from the master git branch <branchname>. Here you will create code for the new feature.
- merge the feature branch with the master (or other branch where you want it to be). You can merge two branches locally or in remote.



Branch Operations

You can do the following with branch:

- Creating new branch: git checkout –b <branch-name>
- Pushing branch from local to remote repo: git push origin <branch-name>
- Renaming branch:
  - Renaming local branch: git branch –m old-name new-name
  - Renaming remote branch: git push origin :old-name new-name
- Deleting branch
  - Deleting local branch: git branch –d <branch-name>
  - Deleting remote branch: git push origin –d <branch-name>

New branches for New Features

In a project, two developers need to build sign-in and sign-up pages at the same time. How can they do that without affecting the existing application?

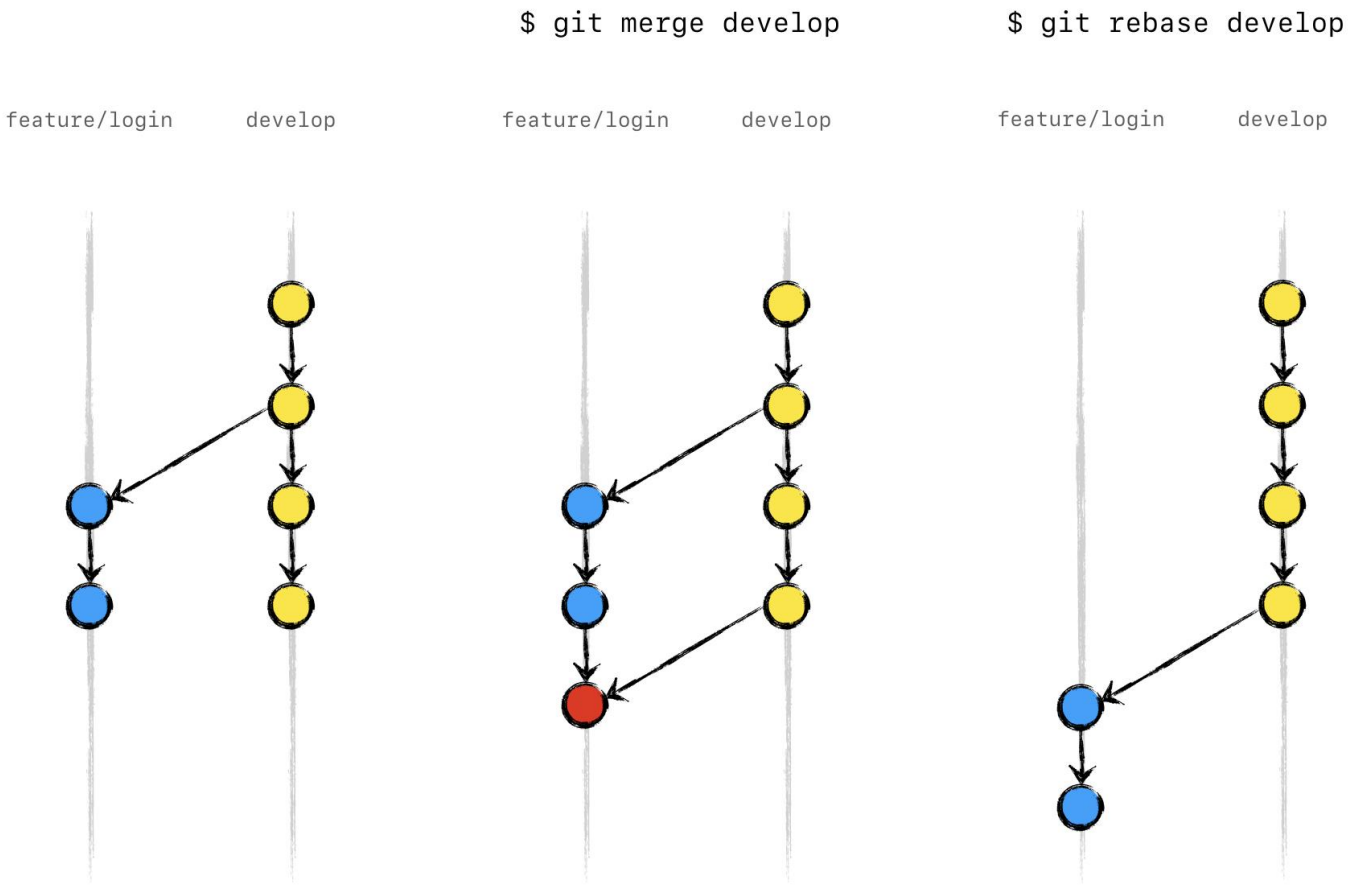
They can create new branches for each new feature, such as signin and signup branches and work on the features parallelly.

Building New Feature: signin

For a new feature to be built:

- Create a new branch from the master branch: git checkout –b signin
- Add your new code in the new feature branch: git add <filename>
- Commit your changes once done: git commit –m “add signin page”
- Push your changes to the remote: git push origin signin

Merging

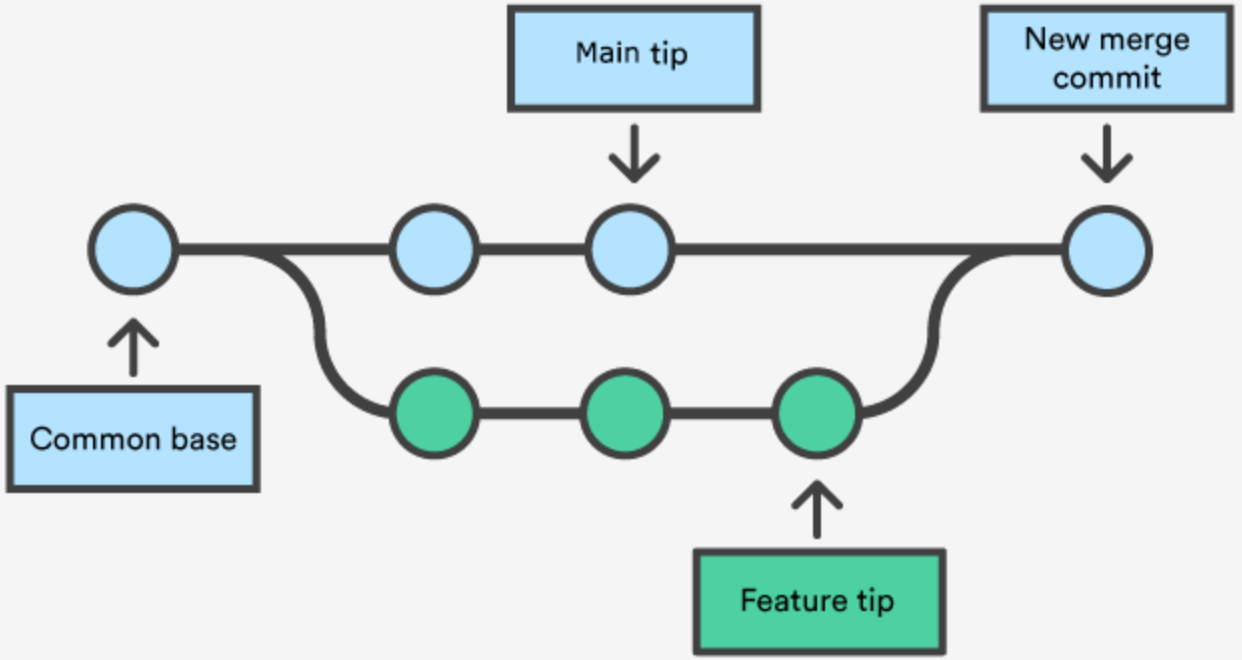


Integrating Changes

Once you have developed your feature in a separate branch, you need to integrate the feature and master branch. You can do that using one of the following two commands:

- merge
- rebase

What is Git Merge?



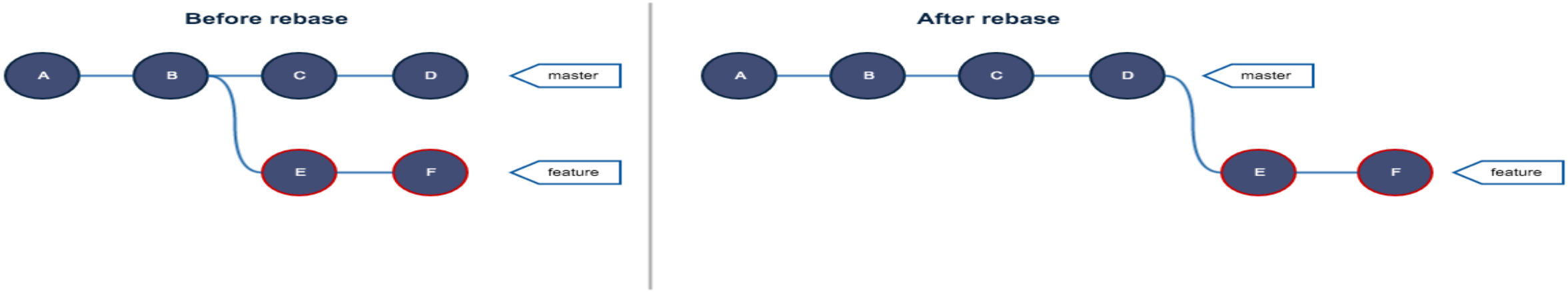
Merge is an operation to integrate changes from one branch to another branch by adding a new commit

Merging Feature Branch: sign-in to master

You can merge with sign-in branch with the master in two ways:

- In your local and push master: git merge master sign-in && git push origin master
- In remote server: you need to create a new pull request (or merge request) from sign-in branch to master branch in GitHub portal.

What is Git Rebase



Rebase is an operation to integrate changes from one branch to another. It is an alternative to the merge command.

If you are re-bashing feature branch with master, the command will remove the original commits on feature, bring all the commits from master to feature and add the original commits on top of it.

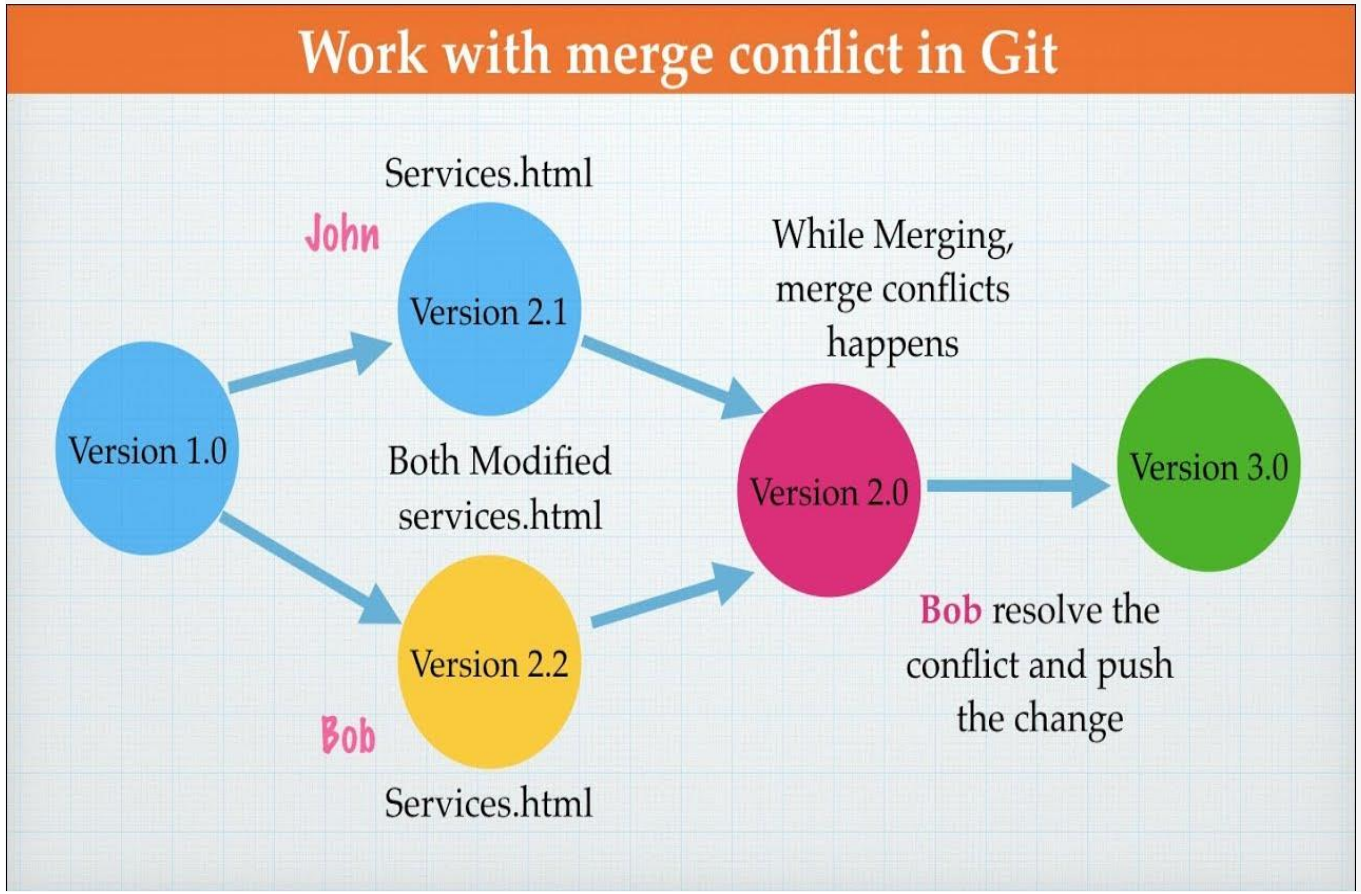
- This changes the commit hashes and re-writes the history.
- This gives linear flow of development when you look at the history of repo.

Merge vs Rebase

While merge and rebase are both used to integrate changes of two branches, let us see how they differ and which one you can use.

	Merge	Rebase
What it does?	Creates new commit while integrating changes from one branch to another.	Does not create any new commit while integrating changes from one branch to another.
Safe or not?	It is safe in nature, as it does not re-write the history of feature branch.	It is destructive in nature, if you rebase feature branch with latest changes in master, your feature branch commit hashes changes.
When to use?	When your feature branch is in remote and someone else is creating new branch on top of your feature branch. Use git merge to get the changes from master to feature.	When your feature branch is local and changing history will not affect others. Use git rebase to get new changes from master to your feature branch.

Conflicts



Suppose you have planned to go for a movie with your family at 8PM tomorrow. And at the same time, your best friend has asked you to join for dinner.

What will you do? Confused?
Well, this is what happens to Git, conflict arises when two developers change the same line of code. Git fails to understand which line to keep and which one to delete. So it asks you to resolve the conflict.

Commands that can cause conflicts

- git merge
- git rebase
- git cherry-pick
- git stash pop
- git pull

Managing Changes

When you make a mistake in word, what do you do ? CTRL + Z, right?

You cannot undo your mistakes in Git as you do in word, but don't worry, Git gives you the following.

Set of commands to undo your changes:

- *amend*
- *checkout*
- *reset*
- *revert*

Set of commands to undo/remove your files:

- *rm*
- *clean*

Git Amend

git commit --amend command is used to fix your previous commit where you do not want to add a new commit. You can:

- *Correct your typos* in commit messages
- Add small changes in the file that you missed adding in the last commit

Git Checkout

Checkout is used to switch. You can switch between branches, commits and files. Here you will learn how to use this command to undo changes.

You can apply checkout command on:

- Working file
- Commit

Checkout Files

If this is what you feel, you can use checkout command to discard all your changes from your working area and switch to the last committed version of your file. This command will help you restart your work as if nothing happened.

Example git checkout filename

Note: This command will only discard the changes in the working area. If you have already staged the changes you need to use reset command first..

Checkout Commits

While debugging the application, if this thought crossed your mind, then you can use git checkout command to switch to a specific commit.

For example: git checkout commitSHA

- commitSHA is the commit has to identify it uniquely. You can find this using git log
- This command **creates a detached head**, meaning, this will **give you a temporary branch** to work and debug.
- Avoid creating any new commits here, as this is a temporary branch.

Git Reset

Reset, as the name says, is used to reset your work to a specific point in time (a commit). It is a powerful and versatile command, using which you can even **undo the changes that you have already committed.**

Imagine your git history as a timeline, then you can quickly jump to a specific time in the past and reset your work as if nothing happened.

To apply **reset** command on:

- staged files
- commits

Reset – Staged files

If you have staged your changes and forgot to add something, you can reset the file, so the file is moved from staging to working area where you can make the required changes.

For example: ***git reset filename***

Reset – Commit Files

You can move to the previous commit where the app is working fine. Did you wonder what will happen to the changes in those two commits that you are skipping? You can use one of the following **three flags,** which decides in which stage the changes should move:

- *--soft* : this moves your commit changes into staging area and does not affect your current working area.
- *--hard* **:** this deletes all the commit changes. Be cautious with this flag. You might lose your changes as this flag resets both staging area and working directory to match the <commit>.
- *--mixed :* this is the default operating mode, where your commit changes are moved to working area.

For example, *git reset --mixed <commitSHA>* , by default moving all the changes to the working directory.

Git Revert

You tracked a bug in your project which was due to a rogue commit.

git revert is another command to undo changes from an old commit, similar to reset. However, git revert inverses the changes from that old commit and creates a new revert commit, instead of deleting the old commit.

This helps prevent Git from losing history.

For example,

git revert commitSHA

git revert –n commitSHA

Revert vs. Reset Commits

While both reset and revert can undo your commit changes, the following are a few differences.

Revert	Reset
Does not delete commits and preserves history.	Deletes commit and changes history.
If other developers are using your branch to build new features, prefer revert to undo changes on that branch as it might prevent any conflicts.	Prefer reset if the branch is only in local

Delete Files

Just like changes (in a file), you might also want to undo/remove files. Depending on its status, whether Git is tracking the file or not, you can use the following commands to remove them:

- *git rm*
- *git clean*

Git rm

git rm is used to delete any tracked file from your repository. Files from both the staging area and the working directory can be removed using the same.

These changes will not persist until a new commit is added, which in turn creates a new commit history.

You can get back your deleted files using git reset and git checkout

Restore Deleted Files

How to restore a deleted file?

- First, find the commit ID where the file was deleted: **git rev-list –n 1 HEAD -- filename**
- Then checkout to that commit ID to get back the file **git checkout deletingCommitId^ -- filename**

Git clean

git clean command undoes files from your repo. However it stands unique from other undo operations like checkout, reset and revert as it primarily focuses on untracked files.

- git clean is undoable
- Git clean makes hard file deletion possible, similar to the Unix rm command.
- It is good to execute git clean –n command to perform a dry run, which helps to know the list of files to be removed.

Golden Rules of Git

- Create a new repository for every new project
- Create a new branch for every new feature.
- Let branches be temporary, that is delete the merged branch.
- Write a good descriptive commit message, where:
  - The first line should not be more than 75 characters
  - Fix, feature or update is present at the start of your commit message

Use Pull Request (PR) to merge your code with master.

Course Summary

In this session, we have learned about Git. We have understood the following conceps,

- Get a table and keep all the items than you want to pack underneath.(*git init*)
- Now you might select specific kitchen items, dust them and club similar items (like spoons) together. (doing changes – working area)
- Add the items that are ready to the box. (*git add* - Staged)
- Seal the box and add a label – “Kitchen Items”. (*git commit* - Committed)

After *git commit* a unique hash is created and the changes are saved.

Creating SSH Keys

Generate new SSH key, ssh-keygen –t rsa –b 4029 –C “youremail@gmail.com”

Assume you are packing items in the house in different boxes and labeling them to identify it later.

- Get a table and keep all the items than you want to pack underneath.(*git init*)
- Now you might select specific kitchen items, dust them and club similar items (like spoons) together. (doing changes – working area)
- Add the items that are ready to the box. (*git add* - Staged)
- Seal the box and add a label – “Kitchen Items”. (*git commit* - Committed)

After *git commit* a unique hash is created and the changes are saved.

How to use PowerPoint Designer

How it works:

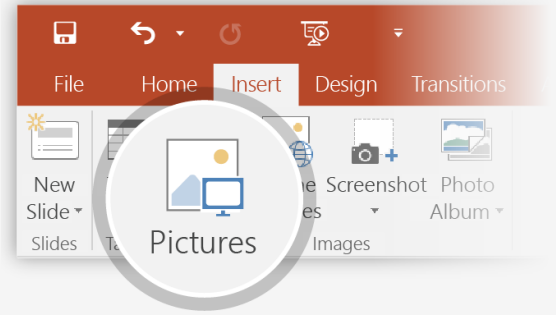
1. Start a new presentation by going to **File** > **New** > **Blank Presentation**.

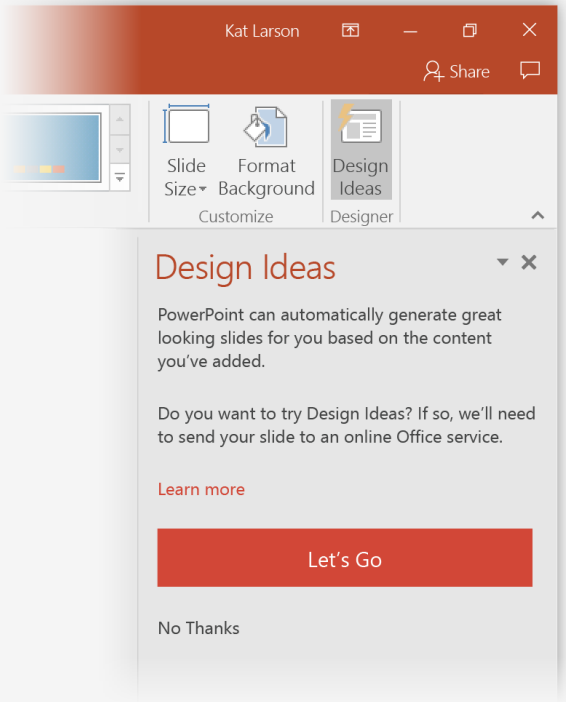
2. On the very first slide, add a picture: Go to **Insert** > **Pictures** or **Insert** > **Online Pictures** and choose the picture.

   Hint: You need to be online when you add the picture.

3. When PowerPoint asks your permission to get design ideas, select **Let's Go**.

4. Choose a design you like from the **Design Ideas** task pane.



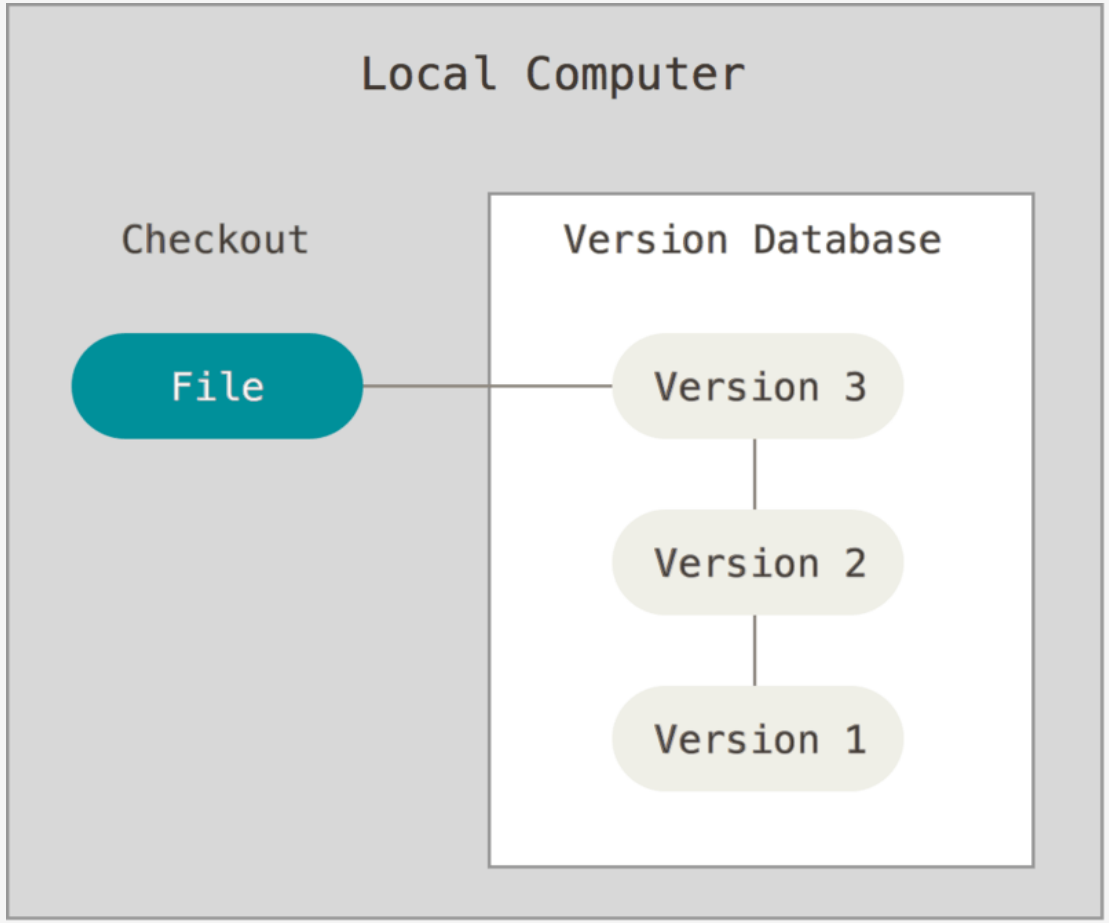


Morph

Morph makes smooth animations and object movements in your presentation. You use two similar slides to perform the animation, but it looks to your audience like the action happens on one slide.

Play the video on the right to see a quick example.

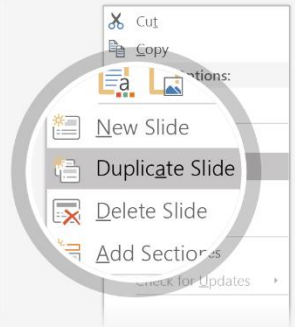
Morph is a subscription-only feature. If you have an Office 365 subscription, you can try it yourself with the steps on the next slide.



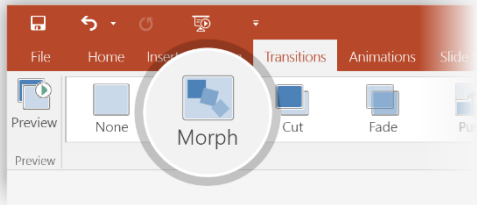
Setting up Morph

Try it yourself with these two simple “planets”:

1. Duplicate this slide: Right-click the slide thumbnail and select **Duplicate Slide**.



2. In the second of these two identical slides, change the shapes on the right in some way (move, resize, change color), then go to **Transitions** > **Morph**.



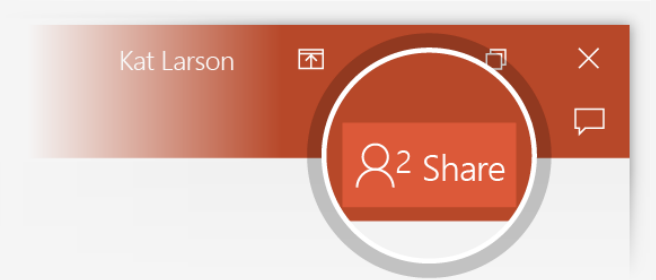
3. Return to the first of the two slides and press **Slide Show** button and then select **Play** to see your circle morph!

Hint: **Effect Options** gives you even more options for **Morph**.

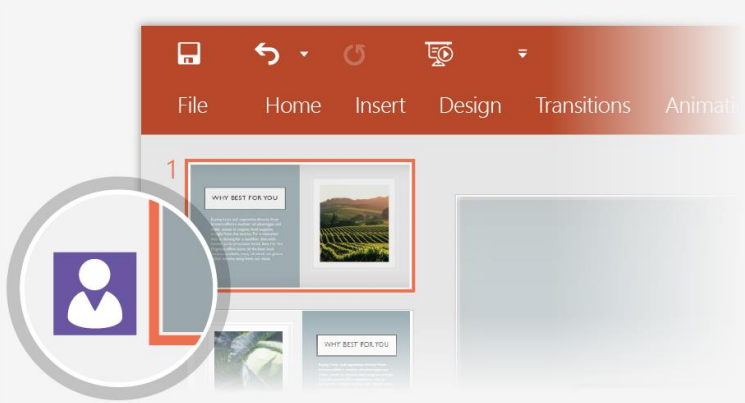
Working together in real time

When you share your presentation with others, you'll see them working with you at the same time.

How it works:



1. Select Share from above the ribbon, or by using short-key Alt-ZS, to invite people to work with you (You can save to the cloud at this point.)



2. When other people are in the presentation, a marker shows who is on which slide…

3. …and the part of the slide they're editing.

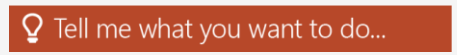
You’re an expert with Tell Me

The Tell Me box finds the right command when you need it, so you can save time and focus on your work.

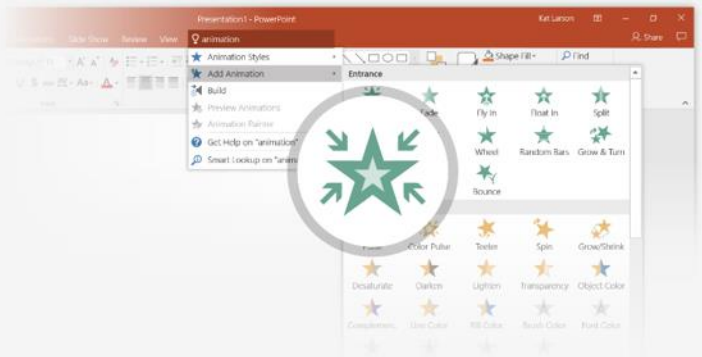
Try it:

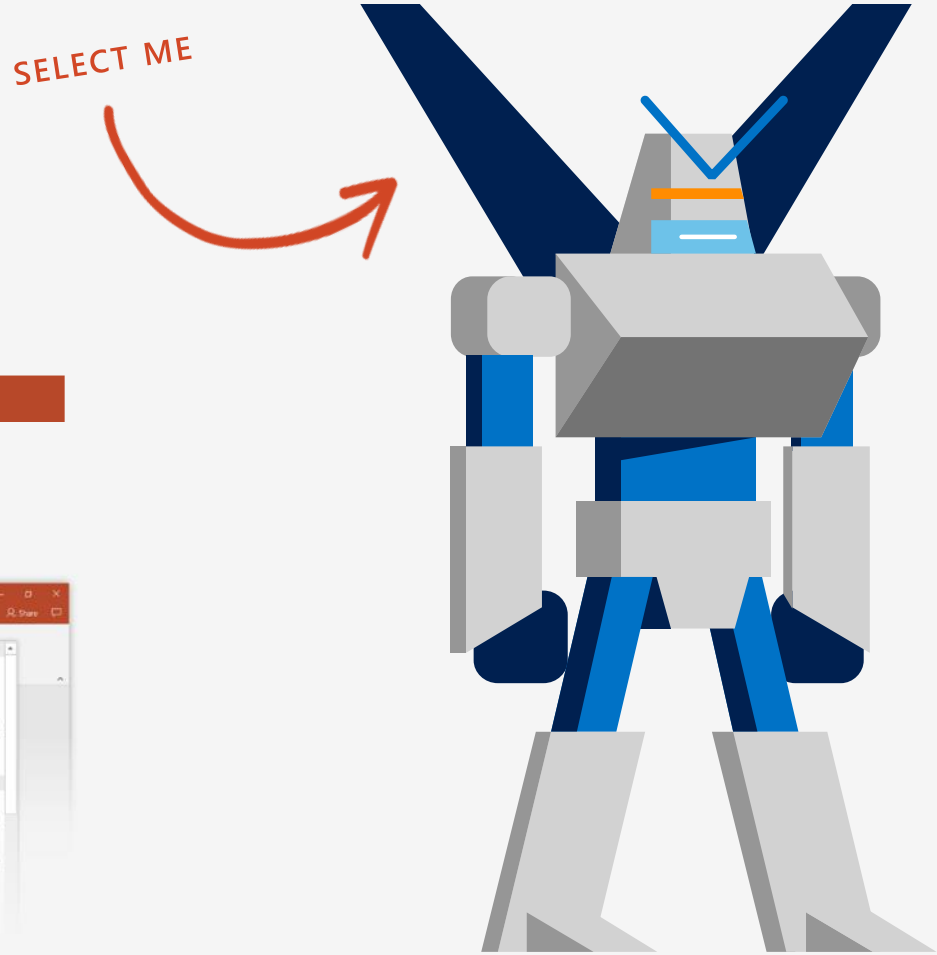
1. Select the Robot picture on the right.

2. Type *animation* in the **Tell Me** box, and then choose **Add Animation**.



3. Choose an animation effect, like **Zoom**, and watch what happens.

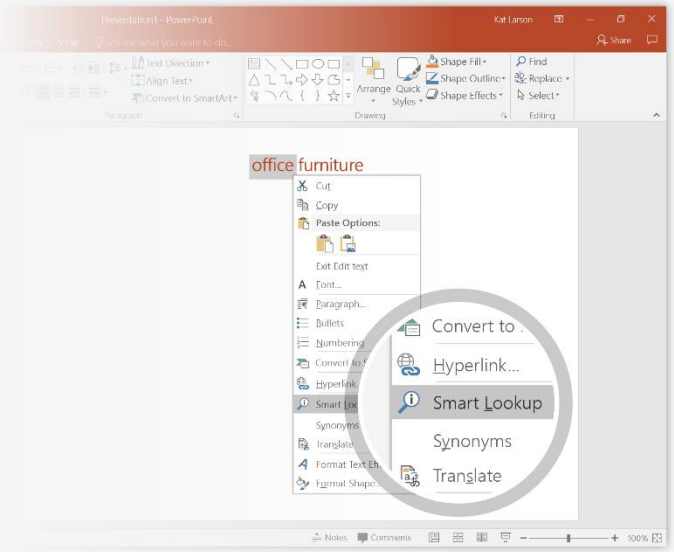




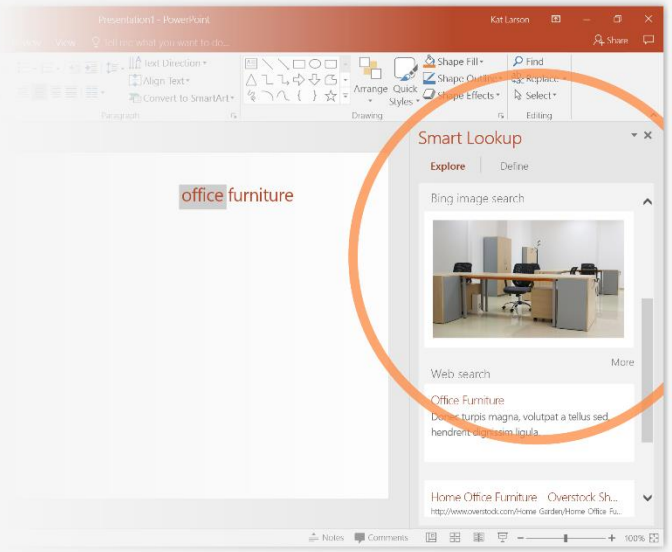
Explore without leaving your slides

Smart Lookup brings research directly in to PowerPoint.

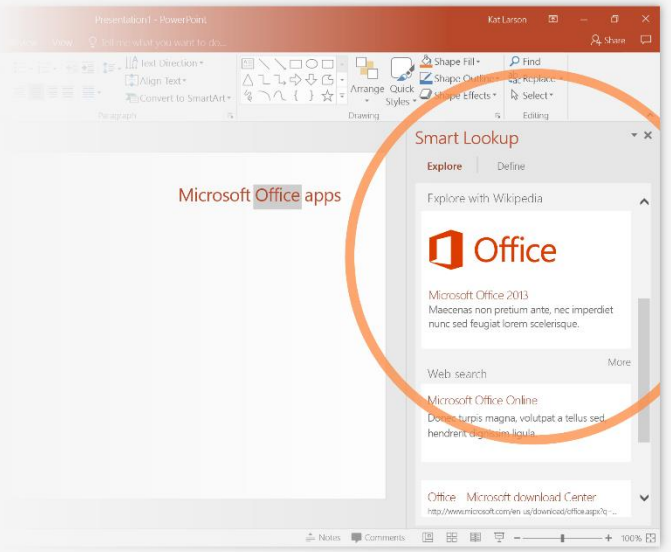
Try it:



1 Right-click in the word *office* in the following phrase: office furniture



2 Choose **Smart Lookup**, and notice that results are contextual for that phrase, not Microsoft Office apps.



3 Just for fun, try Smart Lookup again by right-clicking in the word *Office* in Step 2.

Maven

Maven is a very popular **build automation tool** used for **Java** projects. In this session, you will learn:

- Core concepts of Maven
- Significance of POM file
- Build lifecycle of Maven
- Phases of Maven
- Goals of Maven
- Usage of various Maven commands
- Integration of Maven with Jenkins.

What is Maven?

Maven is a comprehensive build management tool that helps developers in performing the following tasks related to any project.

- Compiling the source code
- Running the test
- Packaging the code into jar files.
- Performing related activities such as,

Create websites -> upload build result -> Generate reports

- Deploying the final product

Why Maven?

Often while working on a Java application, you might be handling numerous jar files.

You will have to handle proper dependency, version inclusions, building, publishing as well as deploying the app.

One of the efficient and hassle-free ways to deal with these activities is to use a automated build tool like **Maven.**

Setup Maven environment

Verify Maven Installation using,

```
$ mvn –version or $mvn -v
```

Maven Core Concepts

Maven works around the following core concepts:

- ***Project Object Model*** **(POM)** is the XML representation of the project where all the dependencies and configuration details are stored. POM plays a major role in ensuring that all the project resource references are maintained.

- **Build Life Cycles, Phases, Goals** – Maven build process is composed of many build life cycles. Each life cycle has one or more phases. Each phase has one or many associated goals.

Maven Core Concepts (continued)

- ***Dependencies and Repositories –*** Dependencies are external JAR files required for the project to work. Maven downloads these dependencies into the local, central or remote repository.

- **Build Plugin** – Adding plugins to the POM file allows us to add new custom actions to be done during the build process.

- **Build Profile** – Projects can be built differently by using different build profiles.

POM File

Each project has a corresponding POM file that is located in the root directory

Whenever a goal (a specific build task) has to be executed, Maven looks for the configuration details in POM

- Project details can be represented in the form of XML file called **pom.xml**
- It is the fundamental unit that contains information about the project.
- It holds all the resources required for a build such as **source code location**, **test source**, **dependency details** such as external or internal dependency etc.

Unique Identifier for POM file

pom.xml takes minimal coordinate attributes as

Inputs for the project as groupId : artifactId : version (alias GAV)

POM stores the information such as the location of the source code and records any external dependencies. It describes what needs to be built as part of the project.

Sample POM file

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.devops.nglabs</groupId>
    <artifactId>my-devops-project</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

archetype:generate

- **Archetype**
- **Group ID**
- **Artifact ID**
- **Version**
- **package**

Dependency Management

One of the key benefits of using Maven is its effective dependency management mechanism.

As the project complexity grows, the number of jar files and other external APIs that might be needed for the project to consume/interact might grow. This might lead to difficulties in managing such dependencies along with appropriate versions.

With Maven, all dependencies of your project are maintained in a single pom.xml file. Maven takes care of downloading these dependencies into the local repositories and makes them available for the project.

Maven Repositories

All the dependencies are held and maintained in a repository in Maven. These repositories are the directories of packaged jar files.

Whenever Maven searches for dependency addition, it looks at local repository followed by central repository then the remote repository.

- A ***local repository*** refers to the repository on a developer's computer.
- A ***central repository*** is the one that Maven community provides
- A **remote repository** could be one on the web server from where the dependencies can be downloaded.

Maven Repositories

All the dependencies are held and maintained in a repository in Maven. These repositories are the directories of packaged jar files.

Whenever Maven searches for dependency addition, it looks at local repository followed by central repository then the remote repository.

- A ***local repository*** refers to the repository on a developer's computer.
- A ***central repository*** is the one that Maven community provides
- A **remote repository** could be one on the web server from where the dependencies can be downloaded.

Get a New Dependencies

There are many repositories that hold some of the commonly used Maven dependencies.

https://mvnrepository.com/ is the recommended site for getting the needed ones.

Let us search for **spring-core** maven dependency in https://mvnrepository.com/ . You could click on the needed version to get the dependency information as shown below. The same can be copied to the dependencies element in pom.xml

```
<dependency>
    <groupId>core-ibank-project</groupId>
    <artifactId>core-ibank-project</artifactId>
    <scope>system</scope>
    <version>1.0</version>
    <systemPath>${project.basedir}/lib/core-ibank-project-0.0.1-SNAPSHOT.jar</systemPath>
</dependency>
```

The dependency will be downloaded and kept at ***MAVEN_REPOSITORY_ROOT=.m2/repository/ in Linux***

Dependency Ranges

Till now, we were defining the version for dependency uniquely

Let us see how we can specify dependency version for

- As greater than range
- In between range

Dependency Ranges

If user depends on any version of Junit, which is higher than 4.8

- Dependency range for Junit will be specified as Junit > 4.8
- You can define it using exclusive quantifies boundary denoted as ***(,]***

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>(4.8, ]</version>
    <scope>test</scope>
</dependency>
```

Dependency version in between range

If user depends on any version of Junit, which is greater than or equal to 4.8 but less than 4.11

- Dependency range for Junit will be specified as Junit 4.8 to Junit 4.11
- You can define it using exclusive quantifies boundary denoted as ***[,)***

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>[4.8, 4.11)</version>
    <scope>test</scope>
</dependency>
```