CS378

DR SARAH ABRAHAM

SOURCE CONTROL AND CI

WORKING WITH LARGE SCALE SYSTEMS

- Many things that you can ignore in smaller scale development become essential in large scale projects
  - How do I coordinate code submission with team members?
  - How do I ensure what builds on my system runs for other team members?
  - How do I work with artists, designers, and other non-programmer contributors?
- Game development tends to hit these development challenges earlier than other types of software development

WHAT IS SOURCE CONTROL?

- Allows multiple developers to make changes to a shared codebase
- Relatively straightforward in the serial case:
  - I work on the code, share it with you, then you work on the code
- But becomes more complicated in the concurrent case:
  - We both work on the code then submit it
- Also where is the code?

MASTER VERSUS LOCAL COPIES

- Need for a “definitive” copy of the code that is somewhere safe
  - In-house server or cloud solution
- Need for “working” copies of the code that can safely be tested and modified on a developer’s machine
- Even if a working copy of the code breaks, should not take down the definitive copy
  - …or at the very least we can get the working definitive copy back with as little effort as possible

GIT

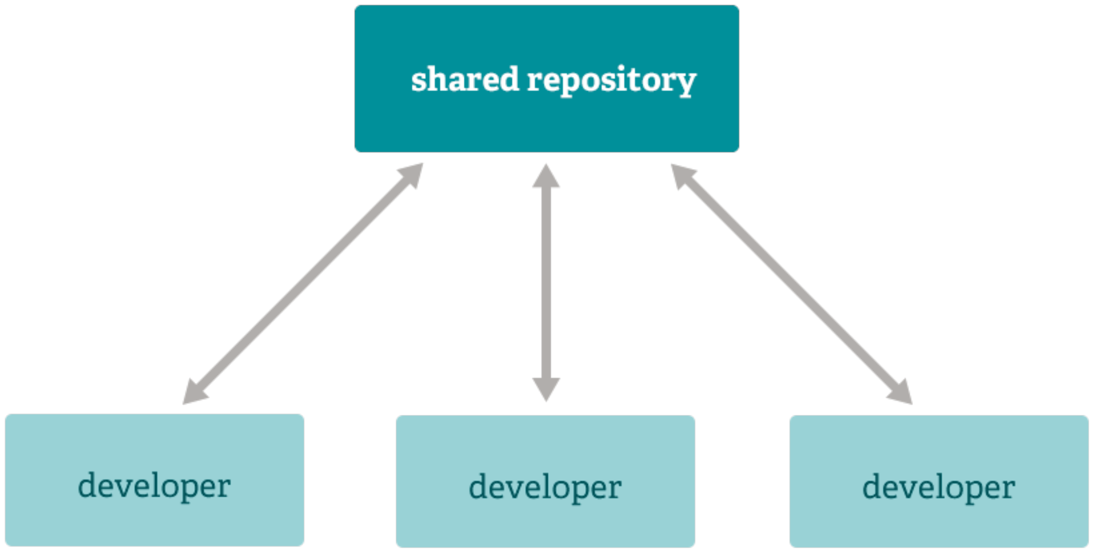
- De facto version control system in software development
  - Has mostly beaten out Subversion in this space
  - Mercurial is another popular choice but this is also a distributed source control manager (DSCM)
- Primary benefits of git are that it is small, fast, and safe

DISTRIBUTED CONTROL

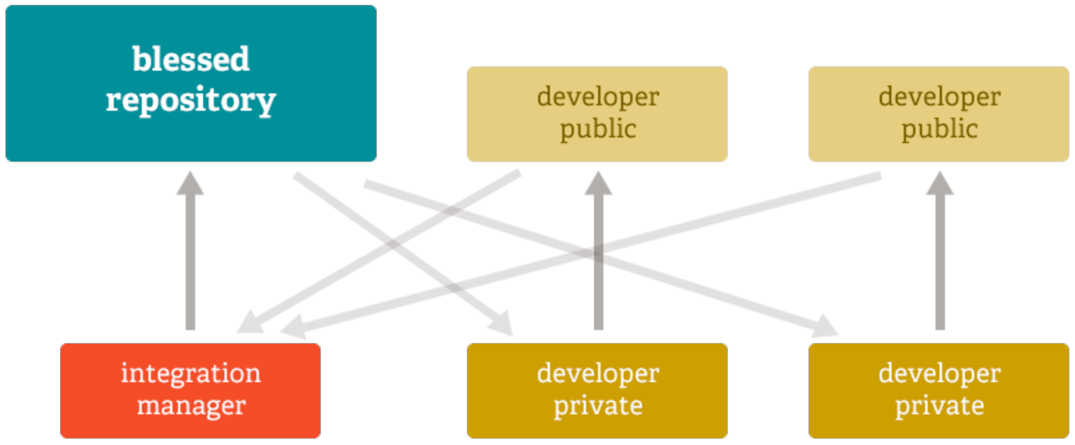
- In a DSCM you access a “clone” of the entire repository rather than “checking out” the latest version
  - Have a full backup at all times
  - Fewer points of failure
  - Easier to fix bad commits
- No notion of a “central” repository
  - Everyone’s working copy is the full repository
- Supports multiple types of workflows

COMMON WORKFLOWS

- Centralized Workflow
  - Developers push changes whenever they complete a task
  - Must integrate other developers' changes before pushing



- Integration Manager Workflow
  - Developers create pull requests for an integration manager to push to the repo
  - Works well with open source projects where anyone can submit



STORAGE FORMAT

- Git stores every commit/file in a hashed document
  - Every commit is a separate entity that is immutable
  - Changes stored in `reflog` as a reference and garbage collected after 30 days
- Files compressed with zlib to reduce storage size for better efficiency

WHAT DO YOU DO IN GIT?

- Basic operations:
  - Initialize
  - Clone
  - Pull
  - Commit
  - Push

INITIALIZATION AND CLONING

- `git init` creates a new git repository in current directory
  - Creates .git subdirectory containing all necessary metadata
  - HEAD file also created to point to current commit
- `git clone` creates a copy of an existing repository
  - Usually how you create a local working copy
  - Creates remote connection called “origin” pointing to original repository

SETTING UP A WORKING DIRECTORY

- Numerous quality of life settings when creating your git repository can be done through git configuration and environment variables
- Also important to set up a `.gitignore` file to prevent including unwanted content
  - Intermediate build data
  - Final builds
  - Project or IDE settings
- Determining what should be included on a `.gitignore` varies from engine to engine
- An example UE4 `.gitignore`: https://gist.github.com/anveo/0d3fef240cb1b46178e6
  - But there are many others!

PULLING AND PUSHING

- `git pull` runs:
  - `git fetch` to download content from the specified remote repository (e.g. origin)
  - `git merge` to merge remote content into local merge commit
- **Must pull before pushing if remote changes do not match local changes**
- `git push` pushes specified branch to specified remote repository
  - Possible to use `force` overriding “upstream” changes but very situational -- do not use unless you understand why you’re doing it!

COMMITS AND LOCAL REPOSITORY MANAGEMENT

- `git commit` is similar to saving
  - Creates actual commit from “staged” files
- `git status` shows current changes to working repository
- `git add` includes requested files to staging
- Staging allows user to select local changes to commit
  - `git reset` can unstage files that should not be staged

BRANCHING VERSUS FORKING

- Branching allows for multiple “working copies” of the same repository
  - Powerful tool that allows for multiple types of work flows and efficient, clear ticket management
  - `git branch` can create, rename and delete branches
- Forking gives every developer their own server-side repository
  - Developers push to their own server-side repository and project maintainer can integrate changes as necessary
  - Useful on large, open source projects with lots of contributors

MERGE CONFLICTS

- Occur when git cannot resolve the “correct” way to integrate changes
  - Multiple people changed the same line of code
  - A file was deleted but is being modified locally
- Note that a conflict is never on the remote side -- only the local side
  - As frustrating as it may be in the moment, it can always be solved!

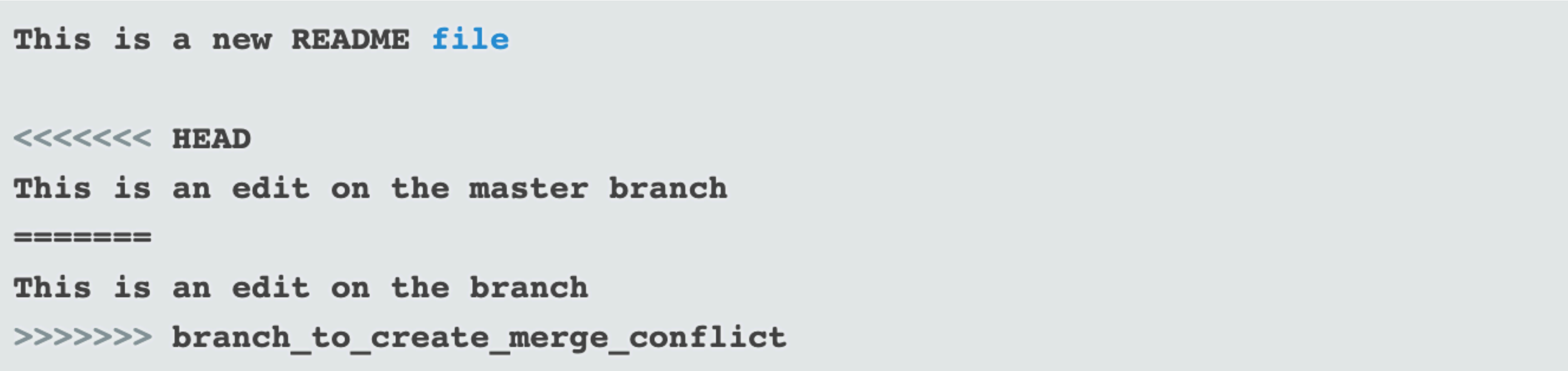
FAILURE TO START MERGE

- Cannot initiate merge if there are changes in the working area or stages
  - Local changes can be committed
  - Local changes can be “stashed” away (`git stash`)
  - Can switch, or create branches, or undo changes using `checkout`

FAILURE DURING MERGE

- Cannot complete a merge due to a conflict between the local branch and the branch being merged
    - Conflict must be resolved by looking through the offending file and manually fixing
    - Must compare <<< current-branch to >>> content-to-merge and select correct content to keep
    - Can also abort the merge attempt using `abort` flag

GIT MERGE EXAMPLE



```
This is a new README file

<<<<<<< HEAD
This is an edit on the master branch
=======
This is an edit on the branch
>>>>>>> branch_to_create_merge_conflict
```

https://opensource.com/article/20/4/git-merge-conflict

- Top <<< section is current branch (HEAD)
- Bottom >>> section is what is being merged
- === separates the conflicting segments of code (only one segment is valid)
- Text is generated by git within the file

WHAT ABOUT BINARY DATA?

- Git needs to clone every version of every file due to its distributed nature
  - Works well generally
  - Not so great for large assets
- How can we handle this problem?

GIT LFS

- Git Large File System
- Replaces large, binary files in the repository with pointers to assets in an LFS cache
    - Handled automatically so no need to understand how the pointers work
- Essential for working with game engines and other creative projects
    - Numerous binaries for artists and designers
    - Levels and other assets are almost always binary data!
- Need to install LFS **once** on the working machine to track all file types that are binary data:
    - https://git-lfs.github.com/

LOCKING FILES

- Possible to lock a file meaning on the user holding the files lock can modify it
  - Prevents distributed work on a given file
  - More useful for binaries than code
- Git LFS allows for locking binary files using `--lockable` flag when first tracking the data type
  - Must use `git lfs` on the command line to lock it before modifying and unlock it so others can access it
- Can also handle file locking through GitLab UI
- More info on both here: https://docs.gitlab.com/ee/user/project/file_lock.html

IS THIS ALL THERE IS TO GIT?

- My goodness, no!
- Git is…very complex
  - Many other available commands and flags
  - All of these are highly situational but if you have a problem, likely git has a solution
  - Best to learn through doing, so don't be afraid to break things!

PERFORCE

- Industry standard for version control in game industry
  - Preferred because of its native handling of large binary assets
- Perforce is centralized rather than distributed
  - Notion of one master version copied to individual workspaces
  - Same idea as git's Centralized Workflow but some implementation differences
- Scales well with large databases and cross repository dependencies

CHECK OUT AND CHECK IN

- Developers pick out specific files to checkout, modify, and submit back to the repository
  - Exclusive checkouts ensure only one developer can access a given file at a time
  - Permissions system ensures developers can only access certain files
- Exclusive checkouts solve problems related to merging binary files such as levels when it is difficult or impossible to merge conflicts
  - But makes workflow sequential so not always ideal

STREAMS

- Perforce uses “streams” for branching and merging
  - Developers can switch between them as with branches
  - Can have merge conflicts when submitting changes but gives notice before merge starts
- Streams can define rules for how changes can be merged and from which streams
- Stream type examples:
  - Release streams are designed to be more stable than its parent
  - Task streams are lightweight, short-term branches

UNREAL AND SOURCE CONTROL

- UE4 has built in support for source control
  - Perforce and SVN supported by default but git works as well
- Activate source control via editor preferences
  - Allows for better check in and out of modified/added assets
  - Allows hot reloading of changes
- Editor-based source control can be used in conjunction with command line (or GUI) source control commands

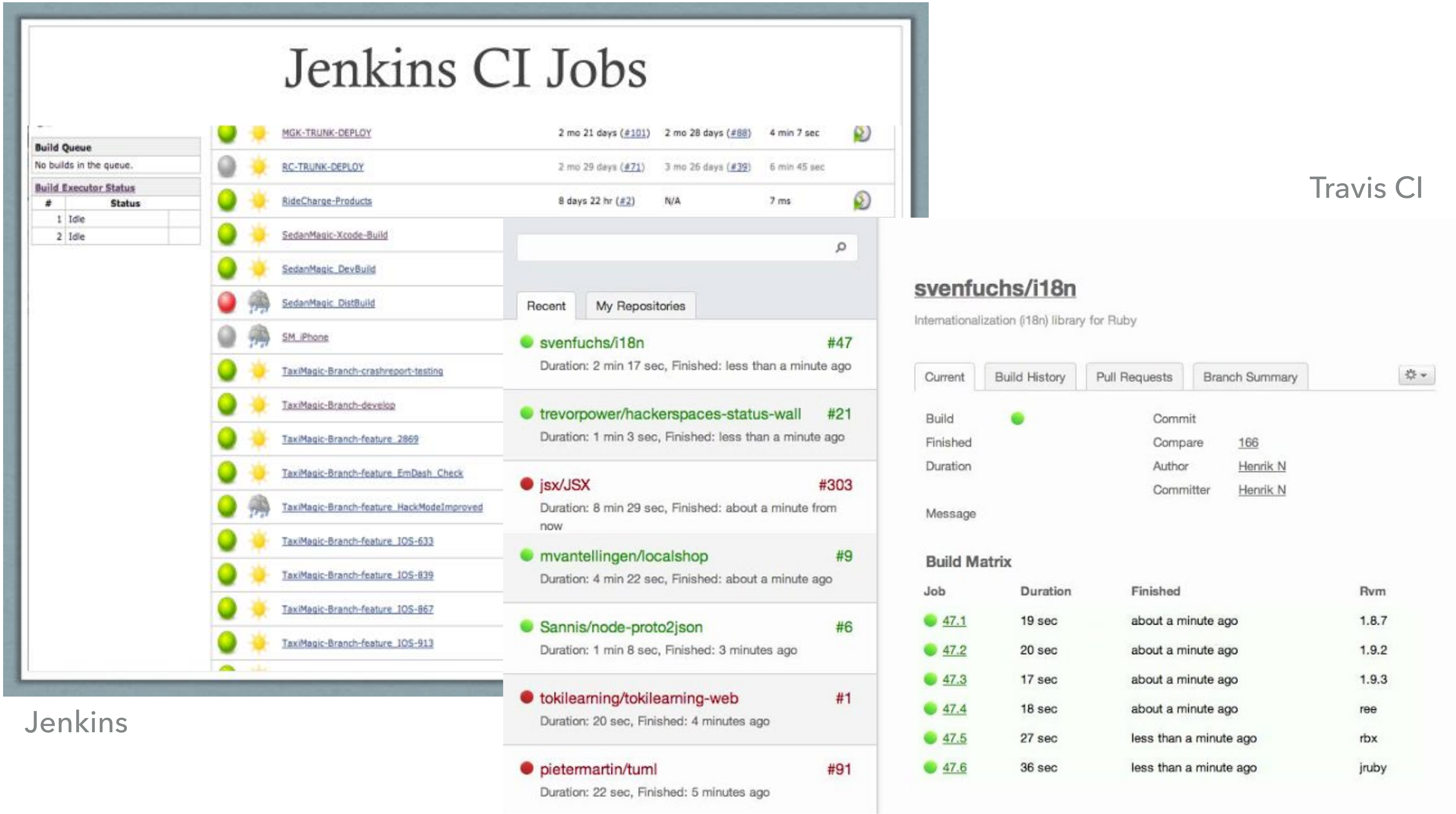
WHAT IS CONTINUOUS INTEGRATION?

- Process of automatically building and testing code every time changes are committed
  - Use of unit tests to ensure some degree of correctness
  - Constant, validated builds helps minimize merge conflicts and unexpected behaviors
- Helps organize builds at different stages of development
- Prevents late-stage issues and keeps pipeline flowing

USING CONTINUOUS INTEGRATION

- When code is frequently committed to a shared repository
- Requires:
    - Well-established work flow
    - Automatic build scheduling
    - Relatively fast builds
    - Unit tests to prevent erroneous code (in theory)

CI SYSTEMS



Jenkins CI Jobs

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

●	☀	MGK-TRUNK-DEPLOY	2 mo 21 days (#101)	2 mo 28 days (#88)	4 min 7 sec	
●	☀	RC-TRUNK-DEPLOY	2 mo 29 days (#71)	3 mo 26 days (#39)	6 min 45 sec	
●	☀	RideCharge-Products	8 days 22 hr (#2)	N/A	7 ms	
●	☀	SedanMagic-Xcode-Build				
●	☀	SedanMagic_DevBuild				
●	☁	SedanMagic_DistBuild				
●	☁	SM_iPhone				
●	☀	TaxiMagic-Branch-crashreport-testing				
●	☀	TaxiMagic-Branch-develop				
●	☀	TaxiMagic-Branch-feature_2869				
●	☀	TaxiMagic-Branch-feature_EmDash_Check				
●	☁	TaxiMagic-Branch-feature_HackModeImproved				
●	☀	TaxiMagic-Branch-feature_IOS-633				
●	☀	TaxiMagic-Branch-feature_IOS-839				
●	☀	TaxiMagic-Branch-feature_IOS-867				
●	☀	TaxiMagic-Branch-feature_IOS-913				

● svenfuchs/i18n #47
Duration: 2 min 17 sec, Finished: less than a minute ago

● trevorpower/hackerspaces-status-wall #21
Duration: 1 min 3 sec, Finished: less than a minute ago

● jsx/JSX #303
Duration: 8 min 29 sec, Finished: about a minute from now

● mvantellingen/localshop #9
Duration: 4 min 22 sec, Finished: about a minute ago

● Sannis/node-proto2json #6
Duration: 1 min 8 sec, Finished: 3 minutes ago

● tokilearning/tokilearning-web #1
Duration: 20 sec, Finished: 4 minutes ago

● pietermartin/tuml #91
Duration: 22 sec, Finished: 5 minutes ago

svenfuchs/i18n

Internationalization (i18n) library for Ruby

Build	●	Commit	
Finished		Compare	166
Duration		Author	Henrik N
		Committer	Henrik N
Message			

Build Matrix

Job	Duration	Finished	Rvm
● 47.1	19 sec	about a minute ago	1.8.7
● 47.2	20 sec	about a minute ago	1.9.2
● 47.3	17 sec	about a minute ago	1.9.3
● 47.4	18 sec	about a minute ago	ree
● 47.5	27 sec	less than a minute ago	rbx
● 47.6	36 sec	less than a minute ago	jruby

Jenkins

Travis CI

GITLAB CI

- GitLab has CI/CD build in
- Set up runners with jobs configured in `.gitlab-ci.yml` file
- Set up pipeline for building and deploying code
    - Include all essential stages and scripts those stages will execute in the runner
- We won't be working directly with CI/CD in UE4, because it has too much overhead, but we'll discuss this topic to better understand how large-scale software build systems work