Ci/Cd Course

May 2022

First Meeting

WORKSPACE DIRECTORY -- add --> STAGING AREA -- commit --> LOCAL REPOSITORY -- push --> REMOTE REPOSITORY

WORKSPACE DIRECTORY can also CLONE and PULL from LOCAL REPOSITORY or REMOTE REPOSITORY

LOCAL REPOSITORY can also CLONE and PULL from REMOTE REPOSITORY

basic git commands:

- git init --> initialize a local git repository, sets up git's internal structure
- git add --> stages file versions (not files!) from local workspace to staging AREA
- git commit --> save a snapshot of staged versions (with reference to the parent version)
- git push --> upload the local repository to the remote one
- git fetch --> refreshes the local information about the remote repository
- git pull --> updates the local repository with changes done on the remote one

branch : a sequence of commits, the last of which is identified by a pointer. technically, just a pointer to a commit some branches (origin/) point to the remote repository.

HEAD : the pointer to the commit representing the current version of the project

- git merge --> when the two branches being merged don't diverge, a fast-forward merge, otherwise, it needs to perform a three-way merge. In this type of merge, git tries to automatically keep all the changes that have been made on any one of the two branches

only (by file and by hunk), but when it fiends a conflict it needs the help of a human.

hunk : a group of line changes clustered together .gitignore file: git will not track the changes made to this files (mostly used for node modules)

Practical examples----------

- clone repository from github (is SSH is chosen, u'll need the secure key): git clone githubLink.git
- create new branch (locally with vscode or remotely with github)
- if i created the branch remotely, i have to fetch it in my local repository
- checkout from main branch to new branch --> git checkout branchName
- get updates from remote repository: checkout to the main branch --> git pull
- i go back to my branch, i do a merge request verso main --> git merge main

Second Meeting

Every salesforce server runs many instances, each instance servers a **tenant**.

Salesforce exposes the retrieve() API, that retrieves XML files representations of components in an organization. SFDX uses this API for it's commands.

Other API exposed by salesforce are:

- deploy() that brings the local defined metadata to the salesforce org
- ...

Deployments can fail for different reasons:

- ill-formed metadata files
- code not compiling
- apex tests failing

- inconsistencies between different metadata files or between versioned metadata and manual changes to the org

Apex testing considerations

- salesforce's way of calculating code coverage is less than ideal
- you get coverage even without assertions
- 100% coverage doesn't mean that you're doing testing right

Move information from one org to another:

- Change Sets: completely manual. Create an outbound message set on the source org picking the components you want to bring over, then accept or change on the target org. It's a manual operations. Since the metadata do pass through your local workspace, you can version the changes
- ...

All metadata sit in the **/src** folder. src/package.xml is your projec's471927Manifest. it lists all the components your project is made of. 'types' defines which type of components, 'members' define which individual components within that type (you can set it to '*' to select all the members of that particular type). package.xml tells deploy() and retrieve() which files to get/put on the org.

In the package.xml, all the object, standard and custom, are under 'custom objects'.

If u edit the package.xml, you can only ADD or UPDATE elements on the org, you cannot DELETE an object and then deploy, that doesn't affect the object in the org. Instead, you can use a 'destructive package' (deploy with the option '--destructive')

Third Meeting

Delta deployments: *sfdy deploy -diff* determine what metadata have been changed, then prepares a temporary .pkg containing only those metadata, than deploys using that .pkg

branch permissions: define who can commit/push to which branches

Pull-request approval: force changes to branches to be approved via pull-request.

Trigger on push: as soon as a new commit is pushed or merged on a remote branch, trigger the pipeline and launch a deploy to the corresponding environment. if the proper branch ermissions are set, pushes to the branch can only be done via pull-request and therefore approved by a trusted user. If the deployment fails, it is necessary to revert

Trigger on pull-request: as soon as a new pull-request towards the branch is opened, launch the deployment to the corresponding environment.