Quick2Cloud for Cloud Systems

Built on IBM Cloud

CLOUDFOUNDRY

Quick2Cloud College - Course 103

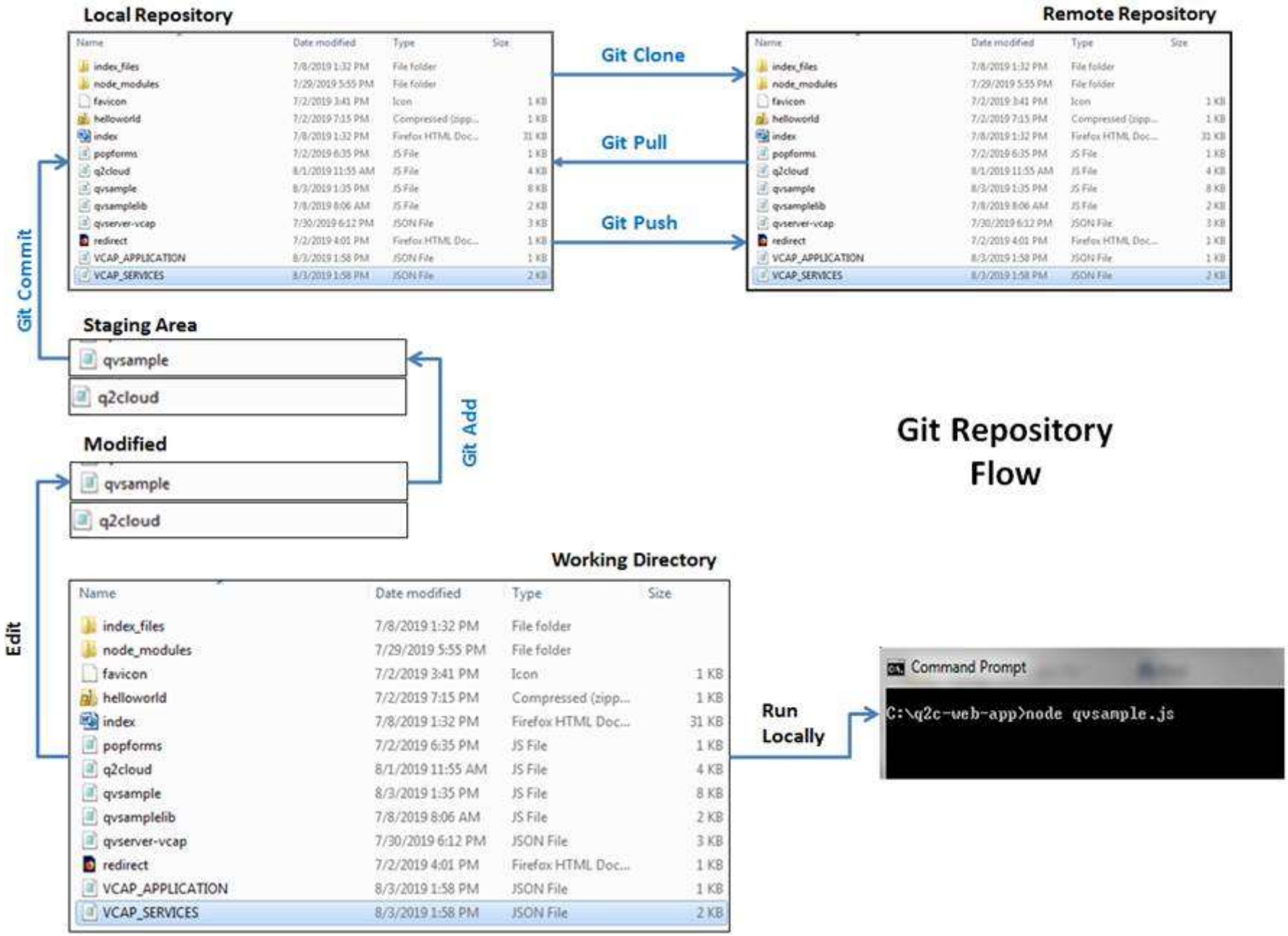
Running the Application Locally with a Repository

Local Repositories

With the introduction of a code repository on your local desktop brings a certain level of professionalism to your application development. No longer are the freedoms of you or your team editing code on-the-fly, stepping on each other□s toes and regressing logic that had already been working. With a repository comes discipline in how you manage your application versions and a methodology that allows reverting back one or more versions if something becomes a problem.

In this course we will be using the git methodology and software to create a repository to manage your local sample application in a controlled fashion. Don□t worry, all of the git repository stuff required were provided for you in your Github prerequisites you downloaded prior. If you did not obtain the git commands at that time then click here and get them now.

Here conceptually, is how a locally installed git repository works on the desktop:



Referencing the diagram above, the working directory represents our q2c-web-app directory and will continue to be the place out of where you run your local copy of your Cloud application. It is also the place where you enhance and test that application before pushing it up to the Cloud.

However, with the introduction of the repository a few extra steps need to be taken to ensure your code is free of traditional development problems that have plagued developers for years.

Keep in mind that repositories are places that store files but they are not places you can directly edit like you can in the working directory. They keep copies of the working directory protected under the safety of the git software.

By git□s own definition, any of the files you modify in the working directory with your code editor are considered **modified**. That makes sense since they actually are. Modify as many files as needed to add a feature and test it directly out of the working directory. When you have blessed your changes and believe them to be high quality you will use git methodology to get your changes into the repository. When the time comes you will actually create your own local git repository and **commit** your working directory code, with the working directory and the repository, for that moment in time, containing exactly the same files.

Consider it the repositories starting point. Since you will be modifying files only in the working directory the two will eventually diverge with the repository holding your application code before your modifications. You can think of the repository as holding **ver00** of your application with the working directory holding **features to ver00**. Having blessed the features as golden, you will then instruct git to place the features into the repositories **staging** area. Staging contains a copy of your modified files and since they are copies any changes made in the working directory, to the same files, will have no effect on the files staged.

To get changed files actually staged you issue a **git add .** command that converts them from **modified** to **staged** where they sit safe from further editing. If you created any new files since the repository was created, those new files will also reside in stage too. The next step is to commit them to the repository and that is done with a **git commit** command.

Once committed, these files replace the files in the repository and once again the repository and working directory contain the same files. But don□t worry, before the repository was overlaid with your staged files a **snapshot** was taken of the old repository to be kept around and can be reverted too at any time.

Thus, if the newly rich feature changes you committed prove to be problematic, you can **rewind** the repository to the previous snapshot and run the original ver00 again. The beauty of this snapshot methodology, every new feature you commit can be backed out if necessary. It is very simple; you just rewind back to a previous repository snapshot.

Oh, and if you are worrying about all the file space being used holding snapshots of old repositories; git is very efficient at keeping snapshots around. Git keeps pointers to unmodified files in other snapshots instead of making additional copies of them which minimizes space and makes processing snapshots very quick and efficient.

Tip: Think of a repository snapshot as an old-school camera□s roll of film with a commit being the camera□s shutter. Each commit captures a snapshot on the film, then the film is wound forward and ready to capture the next snapshot while the older images are still there behind it on the roll

Remote Repositories

Local repositories can link up with **remote repositories** that reside on a server somewhere that is reachable by all your team members. Remember, your local repository is only protecting you but the remote repository is designed indirectly to protect everyone working on the project. Since everyone on your team will also have a local repository and a working directory, the remote will be the place where you all sync up to make sure that everyone□s coding effort is not being stepped on by others.

For the purpose of this course, your local repository is all that will be covered. In **Course 104** on Continuous Delivery the remote repository will play a major role so let us wait to talk about the details until we get there

In the next section you will cover repository branches and how they affect the way you code new features.

Continue