End Lab 01:07:38

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. Learn more.

linux-instance external IP address

35.237.235.14

username

student-01-0bdc17664dd

Download PEM

Download PPK

Pushing Local Commits to Github

1 hour 30 minutes Free ★★★★½

Introduction

For this project, you'll need to fork an existing repository, fix a bug in a script, push your commit to GitHub, and create a pull request with your commit.

What you'll do

- Fork another repository
- Commit changes to your own fork and create pull requests to the upstream repository
- Gain familiarity with code reviews, and ensure that your fix runs fine on your system before creating the pull request
- Describe your pull request

You'll have 90 minutes to complete this lab.

Start the lab

You'll need to start the lab before you can access the materials in the virtual machine OS. To do this, click the green "Start Lab" button at the top of the screen.

Note: For this lab you are going to access the **Linux VM** through your **local SSH Client**, and not use the **Google Console** (**Open GCP Console** button is not available for this lab).

Start Lab

After you click the "Start Lab" button, you will see all the SSH connection details on the left-hand side of your screen. You should have a screen that looks like this:

Accessing the virtual machine

Please find one of the three relevant options below based on your device's operating system.

Note: Working with Qwiklabs may be similar to the work you'd perform as an **IT Support Specialist**; you'll be interfacing with a cutting-edge technology that requires multiple steps to access, and perhaps healthy doses of patience and persistence(!). You'll also be using **SSH** to enter the labs -- a critical skill in IT Support that you'll be able to practice through the labs.

Option 1: Windows Users: Connecting to your VM

In this section, you will use the PuTTY Secure Shell (SSH) client and your VM's External IP address to connect.

Download your PPK key file

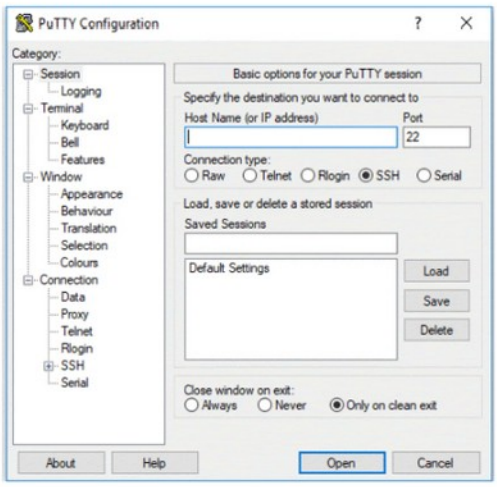
You can download the VM's private key file in the PuTTY-compatible **PPK** format from the Qwiklabs Start Lab page. Click on **Download PPK**.

Connect to your VM using SSH and PuTTY

1. You can download Putty from here

2. In the **Host Name (or IP address)** box, enter username@external_ip_address.

Note: Replace **username** and **external_ip_address** with values provided in the lab.



3. In the **Connection** list, expand **SSH**.

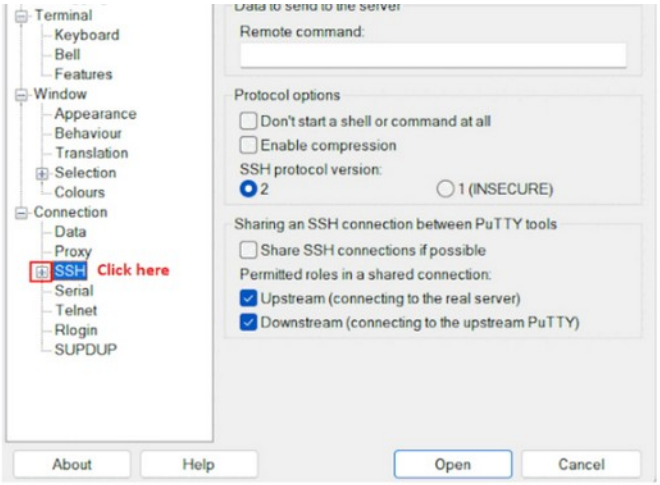
4. Then expand **Auth** by clicking on **+** icon.

5. Now, select the **Credentials** from the **Auth** list.

6. In the **Private key file for authentication** box, browse to the PPK file that you downloaded and double-click it.

7. Click on the **Open** button.

Note: PPK file is to be imported into PuTTY tool using the Browse option available in it. It should not be opened directly but only to be used in PuTTY.



8. Click **Yes** when prompted to allow a first connection to this remote SSH server. Because you are using a key pair for authentication, you will not be prompted for a password.

Common issues

If PuTTY fails to connect to your Linux VM, verify that:

- You entered **<username>@<external ip address>** in PuTTY.
- You downloaded the fresh new PPK file for this lab from Qwiklabs.
- You are using the downloaded PPK file in PuTTY.

Option 2: OSX and Linux users: Connecting to your VM via SSH

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.

Connect to the VM using the local Terminal application

A **terminal** is a program which provides a **text-based interface for typing commands**. Here you will use your terminal as an SSH client to connect with lab provided Linux VM.

1. Open the Terminal application.
    - To open the terminal in Linux use the shortcut key **Ctrl+Alt+t**.
    - To open terminal in **Mac** (OSX) enter **cmd + space** and search for **terminal**.
2. Enter the following commands.

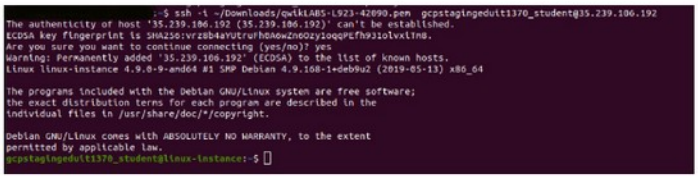
Note: Substitute the **path/filename for the PEM** file you downloaded, **username** and **External IP Address**.

You will most likely find the PEM file in **Downloads**. If you have not changed the download settings of your system, then the path of the PEM key will be **~/Downloads/qwikLABS-XXXXX.pem**

```
chmod 600 ~/Downloads/qwikLABS-XXXXX.pem
```

```
ssh -i ~/Downloads/qwikLABS-XXXXX.pem username@External Ip
Address
```

Address



Option 3: Chrome OS users: Connecting to your VM via SSH

Note: Make sure you are not in **Incognito/Private mode** while launching the application.

Download your VM's private key file.

You can download the private key file in PEM format from the Qwiklabs Start Lab page. Click on **Download PEM**.

Connect to your VM

1. Add Secure Shell from here to your Chrome browser.
2. Open the Secure Shell app and click on **[New Connection]**.

3. In the **username** section, enter the username given in the Connection Details Panel of the lab. And for the **hostname** section, enter the external IP of your VM instance that is mentioned in the Connection Details Panel of the lab.

4. In the **Identity** section, import the downloaded PEM key by clicking on the **Import...** button beside the field. Choose your PEM key and click on the **OPEN** button.

Note: If the key is still not available after importing it, refresh the application, and select it from the **Identity** drop-down menu.

5. Once your key is uploaded, click on the **[ENTER] Connect** button below.

6. For any prompts, type **yes** to continue.

7. You have now successfully connected to your Linux VM.

You're now ready to continue with the lab!

Forking and detect function behavior

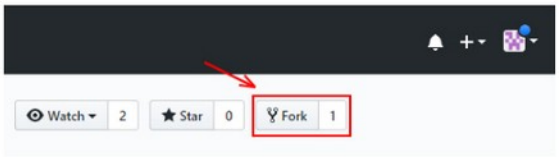
For this exercise, you need to fork an existing repository: `google/it-cert-automation-practice`.

- Open Github. If you don't already have a Github account, create one by entering a username, email, and password. If you already have a Github account proceed to the next step.
- Log in to your account from the Github login page.

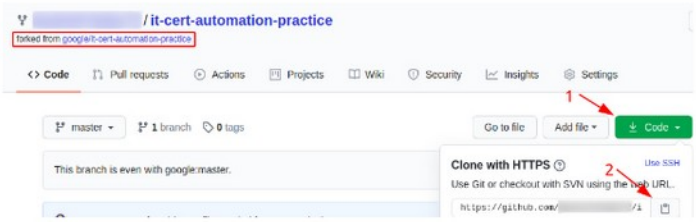
A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Forking a repository is a simple two-step process. We've created a repository for you to practice with!

- On GitHub, navigate to the google/it-cert-automation-practice repository.
- In the top-right corner of the page, click Fork.



That's it! Now, you have a fork of the original `google/it-cert-automation-practice` repository.

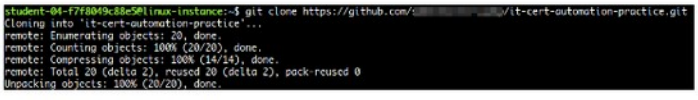


First, clone the repository using the following command:

```
git clone https://github.com/[git-username]/it-cert-automation-practice.git
```

Note: If you enabled two-factor authentication in your Github account you won't be able to push via HTTPS using your account's password. Instead you need to generate a personal access token. This can be done in the application settings of your Github account. Using this token as your password should allow you to push to your remote repository via HTTPS. Use your username as usual. Note down this personal access token as we would need it further in the lab. For more help to generate a personal access token, click here.

Output:



Go to the `it-cert-automation-practice` directory using the following command:

```
cd ~/it-cert-automation-practice
```

First, verify that you have already setup a remote for the upstream repository, and an `origin`.Type the following command and press **Enter**. You'll see the current configured remote repository for your fork.

```
git remote -v
```

Output:

In terms of source control, you're "**downstream**" when you copy (clone, checkout, etc) from a repository. Information is flowed "downstream" to you.

When you make changes, you usually want to send them back "**upstream**" so they make it into that repository so that everyone pulling from the same source is working with all the same changes. This is mostly a social issue of how everyone can coordinate their work rather than a technical requirement of source control. You want to get your changes into the main project so you're not tracking divergent lines of development.

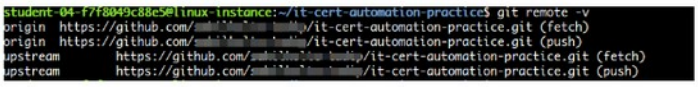
Setting the upstream for a fork you have created using the following command:

```
git remote add upstream https://github.com/[git-username]/it-cert-automation-practice.git
```

To verify the new upstream repository you've specified for your fork, type `git remote -v` again. You should see the URL for your fork as origin, and the URL for the original repository as upstream.

```
git remote -v
```

Output:



Configure Git

Git uses a username to associate commits with an identity. It does this by using the **git config** command. Set the Git username with the following command:

```
git config --global user.name "Name"
```

Replace **Name** with your name. Any future commits you push to GitHub from the command line will now be represented by this name. You can even use **git config** to change the name associated with your Git commits. This will only affect future commits and won't change the name used for past commits.

Let's set your email address to associate them with your Git commits.

```
git config --global user.email "user@example.com"
```

Replace **user@example.com** with your email-id. Any future commits you now push to GitHub will be associated with this email address. You can also use **git config** to change the user email associated with your Git commits.

Fix the script

In this section we are going to fix an issue that has been filed. Navigate to the issue, and have a look at it.

Branches allow you to add new features or test out ideas without putting your main project at risk. In order to add new changes into the repo directory `it-cert-automation-practice/Course3/Lab4/`, create a new **branch** named `improve-username-behavior` in your forked repository using the following command:

```
git branch improve-username-behavior
```

Go to the `improve-username-behavior` branch from the master branch.

```
git checkout improve-username-behavior
```

Now, navigate to the working directory `Lab4/`.

```
cd ~/it-cert-automation-practice/Course3/Lab4
```

List the files in directory `Lab4`.

```
ls
```

Output:

```
student-04-6bbc464c3b60@linux-instance:~/it-cert-automation-practice/Course3/Lab4$ ls
validations.py
```

Now, open the **validations.py** script.

```
cat validations.py
```

Output:

```
student-04-6bbc464c3b60@linux-instance:~/it-cert-automation-practice/Course3/Lab4$ cat validations.py
#!/usr/bin/env python3

import re

def validate_user(username, minlen):
    """Checks if the received username matches the required conditions."""
    if type(username) != str:
        raise TypeError("username must be a string")
    if minlen < 1:
        raise ValueError("minlen must be at least 1")

    # Usernames can't be shorter than minlen
    if len(username) < minlen:
        return False
    # Usernames can only use letters, numbers, dots and underscores
    if not re.match('^[a-z0-9._]*$', username):
        return False
    # Usernames can't begin with a number
    if username[0].isnumeric():
        return False
    return True
```

This script should validate usernames if they start with an letter only.

Here, you can check the `validate_user` function's behavior by calling the function. To edit the validations.py Python script, open it in a nano editor using the following command:

```
nano validations.py
```

Now, add the following lines of code at the end of the script:

```
print(validate_user("blue.kale", 3)) # True
print(validate_user(".blue.kale", 3)) # Currently True, should
be False
```

```
print(validate_user( red_quinoa , 4)) # True
print(validate_user("_red_quinoa", 4)) # Currently True, should
be False
```

Once you've finished writing this script, save the file by pressing Ctrl-o, the Enter key, and Ctrl-x.

Now, run the validations.py on the python3 interpreter.

```
python3 validations.py
```

Output:

```
student-04-6bbc464c3b60@linux-instance:~/it-cert-automation-practice/Course3/Lab4$ python3 validations.py
True
True
True
True
```

Here, as we see the output, it function returns true even if the username doesnot start with an letter. Here we need to change the check of the first character as only letters are allowed in the first character of the username.

Continue by opening validations.py in the nano editor using the following command:

```
nano validations.py
```

There are lots of ways to fix the code; ultimately, you'll want to add additional conditional checks to validate the first character doesn't start with either of the forbidden characters. You can choose whichever way you'd like to implement this.

Once you've finished writing this script, save the file by pressing Ctrl-o, the Enter key, and Ctrl-x.

Now, run the validations.py.

```
python3 validations.py
```

Output:

```
student-04-6bbc464c3b60@linux-instance:~/it-cert-automation-practice/Course3/Lab4$ python3 validations.py
True
False
True
False
```

Now, you've fixed the function behavior!

Commit the changes

Once the issue is fixed and verified, create a new commit by adding the file to the staging area. You can check the status using the following command:

```
git status
```

The **git status** command shows the different states of the files in your working directory and staging area, like files that are modified but unstaged and files that are staged but not yet committed.

You can now see that the validations.py has been modified.

```
student-04-6bbc464c3b60@linux-instance:~/it-cert-automation-practice/Course3/Lab4$ git status
On branch improve-username-behavior
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   validations.py

no changes added to commit (use "git add" and/or "git commit -a")
```

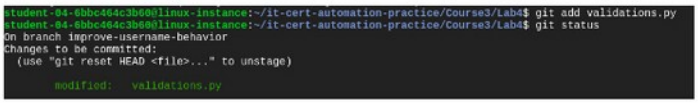
Now, let's add the file to the staging area using the following command:

```
git add validations.py
```

Use the **git add** command to add content from the working directory into the staging area for the next commit.

```
git status
```

Output:



Let's now commit the changes. A git commit is like saving your work.

Commit the changes using the following command:

```
git commit
```

This now opens up an editor that asks you to type a commit message. Every commit has an associated commit message, which is a log message from the user describing the changes.

Enter a commit message of your choice and append a line: "Closes: #1" at the beginning to indicate that you're closing the issue. Adding this keyword has an additional effect when using Github to manage your repos, which will automatically close the issue for you (for more information, please see the documentation here).

```
Closes: #1
Updated validations.py python script.
Fixed the behavior of validate_user function in validations.py.
```

Once you've entered the commit message, save it by pressing Ctrl-o and the Enter key. To exit, click Ctrl-x.

Output:

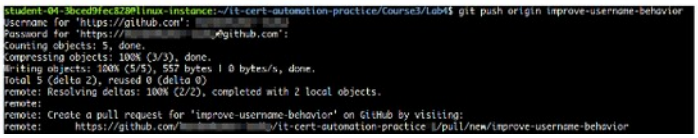
Push changes

You forked a repository and made changes to the fork. Now you can ask that the upstream repository accept your changes by creating a pull request. Now, let's push the changes.

```
git push origin improve-username-behavior
```

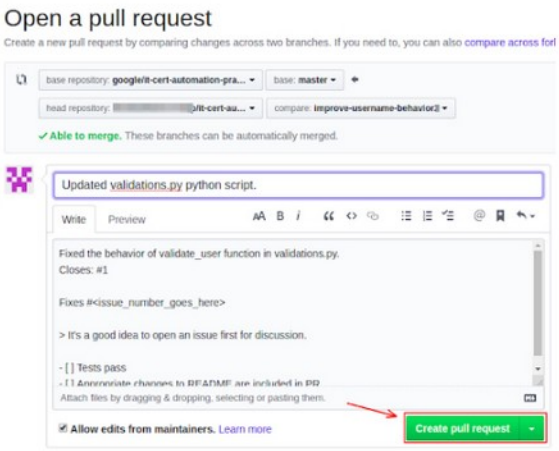
Note: If you have enabled two-factor authentication in your Github account, use the personal access token generated earlier during the lab as the password. If you have not noted down the personal access token earlier, go to your **Github homepage** > Settings > **Developer settings** > **Personal access tokens**. Now click on the token you generated earlier and click 'Regenerate token'. Use your username as usual.

Output:



```
remote:
To https://github.com/[redacted]/it-cert-automation-practice-1.git
 * [new branch]      improve-username-behavior -> improve-username-behavior
```

Then, from GitHub, create a pull request from your forked repository `[git-username]/it-cert-automation-practice` that includes a description of your change. Your branch `improve-username-behavior` is now able to merge into the master branch. It should look like the image below:



After initializing a pull request, you'll see a review page that shows a high-level overview of the changes between your branch (the compare branch) and the repository's base branch. You can add a summary of the proposed changes, review the changes made by commits, add labels, milestones, and assignees, and **@mention** individual contributors or teams.

Once you've created a pull request, you can push commits from your topic branch to add them to your existing pull request. These commits will appear in chronological order within your pull request and the changes will be visible in the "Files changed" tab.

Other contributors can review your proposed changes, add review comments, contribute to the pull request discussion, and even add commits to the pull request.

You can see information about the branch's current deployment status and past deployment activity on the "Conversation" tab.

> **Note:** PR won't be merged on the master branch so that other users can also make a similar change to fix the issue.

Congratulations!

In this lab, you successfully forked a repository, committed changes to your own fork, and created a pull request to the upstream. Well done!

End your lab

When you have completed your lab, click **End Lab**. Qwiklabs removes the resources you've used and cleans the account for you.

You will be given an opportunity to rate the lab experience. Select the applicable number of stars, type a comment, and then click **Submit**.

The number of stars indicates the following:

- 1 star = Very dissatisfied
- 2 stars = Dissatisfied
- 3 stars = Neutral

- 4 stars = Satisfied
- 5 stars = Very satisfied

You can close the dialog box if you don't want to provide feedback.

For feedback, suggestions, or corrections, please use the **Support** tab.