Tool introduction challenge

Introduction

The purpose of this challenge is to provide a hands-on introduction to all the basic technical development tools that would be required for developing a project for the “Beyond OS” course.

This challenge must be performed in its entirety before the 1st session of the course. More specifically it must be submitted by Friday, March 3rd, at 12:00.

An online form for submitting the challenge results can be found here.

In this challenge you will learn:

1. How to set up a personal code repository in GitHub and use Git to track source files in it.
2. How to use feature branches in Git and push them as pull-requests (PRs) to GitHub for code review.
3. How to use Vagrant and VirtualBox to set up a Linux-based development environment.
4. How to use Pipenv to manage requirements for a Python and Django application.
5. How to bootstrap a Django application.
6. How to use GitHub Actions to add basic CI checking to a repository.
7. How to create basic application features in Django.

General notes and guidelines:

- Though programming skills will be required during the course, this challenge does not require prior knowledge of development tools and languages. Students do need to know how to edit source code files that reside on their computers.
- While this document contains links to external resources, you generally should not need to refer to them to complete it, except in some cases where you are referred directly to some installation instructions.
- While it is not necessary to reference external resources, you are still encouraged to review them to gain a deeper understanding of the tools being used. That should be done while taking time constraints into account.
- If you get stuck at any point, **you are encouraged to seek help** by using the Beyond slack channel

- When submitting the results of your work we expect the repository to be in fully working condition. More specifically we expect the `vagrant up` command to work and load a working application.
- We also expect you to use well-documented *commits* and *pull requests* to edit the files in your GitHub repository as explained in this document, we may review the repository to check for that.

Part 1: Git and GitHub

What is Git?

Git belongs to a family of tools that are jointly called Source Code Management (SCM) tools. The purpose of those tools is to allow teams of developers to collaborate on the same code without inadvertently deleting each other's work and to enable each developer to be aware of the code changes that their teammates make.

A basic way to understand Git is to think of it as a file-sharing tool for sharing source code where it not only allows us to share the source code files themselves but also metadata information about how the files were changed over time, by who, and for what purpose.

To enable the tracking of changes to code, Git introduces the concept of a **commit**. A **commit** describes the way that our **code looks at a particular point in time**, as well as some information about how we've changed the code to make it look like it does.

The basic process for working with Git is:

1. Start with an initial state of the code (For a brand new project the state can be simply having an empty folder with no files in it)
2. Make code changes. Those can include:
    - Creating new files
    - Deleting files
    - Moving and renaming files
    - Changing the content of files
3. Tell Git about the changes you've made by using the `git add` command.
4. Create a new commit tracking the changes you've made by using the `git commit` command and including a meaningful short **commit message**.

As we work on our code we will create a series of commits. The commits serve two main purposes:

- They allow us and other developers to learn how the code became the way it is and *why*.
- We can *go back (revert)* to a particular commit in various ways to get the code the way it was in that commit.

The series of commits for a project is also called the project's *commit history* or its "Git log".

Some additional important basic Git concepts include:

- A Git **branch** is a separate series of commits stored by Git. Git lets us keep several different versions of our code that can live side-by-side and have separate commit histories. The ability to have *multiple branches* and to bring them together is key to allowing multiple developers to work on the same project in parallel.
- A Git **repository** is represented by a directory that includes a particular set of files, commits and branches. There are two kinds of repositories:
    - A **local repository** - A repository that is stored on our own computer
    - A **remote repository** - A repository that is stored on another computer

  Git provides us with ways to:

    - *Download* (**pull**) commits from a remote repository to a local repository
    - *Upload* (**push**) commits from the local repository to a remote repository.

What is GitHub?

GitHub is a *website* that allows us to store Git repositories "in the cloud" and make them accessible by other people. There are other websites like GitHub but since GitHub is very popular and is used by many developers and teams, we have picked it up in the course tech stack.

Task 1: Create a GitHub account

If you've never done so before, please go to the following url and sign-up for a new GitHub account. Creating an account on GitHub is free.

The process for creating an account on GitHub is similar to creating an account on any website, you fill-in a username, a desired password and an email address. And then you verify that the email address works by clicking a special link that is sent to you via Email.

You will need to type your account name and password in various commands later so it's best to choose a name and a password that are easy for you to remember and type. Please choose a secure password though, anyone with access to your GitHub password may delete or change the source code of all your projects or projects you're involved in.

Task 2: Create a new repository in GitHub

While we can certainly create new repositories by using Git on our local computer, when working with GitHub it's a little more convenient to first create the repositories on the website. To do this:

1. Go to github.com

2. If you haven't done so already - log into your account (If you are not logged-in the main GitHub page will show a sign-up form. To log-in you need to click the "**Sign in**" link at the top-right corner and fill-in your username and password.
3. Once you're logged in, the left-column of the page will show your existing GitHub repositories. It might be empty at this point. There should be a green "**New**" button at the top. Click on it to go into the "**Create a new repository**" form.
4. In the "**Create a new repository**" form, fill in a name and a description for your new repository. The *name* part is important since you will use it when interacting with the repository in the future. For the examples in this document, we will use the name "**beyond-tutorial**".
5. We will create a public repository so make sure the "**Public**" option is selected.
6. We will let GitHub automatically create the first commit in the repository for us so check all the following check boxes at the bottom of the form:
    - "**Add a README file**"
    - "**Add .gitignore**" - in the drop-down menu that appears, search and select "*Python*". This will make Git automatically ignore temporary files that are automatically created when Python software runs and should not be tracked along with our source code.
    - "**Choose a license**" - in the drop-down menu that appears, please choose the "*MIT License*" which is a very simple and permissive license.
7. Finally, click the "**Create repository**" button.
8. Once we click the button, GitHub may take a few seconds to create our repository. When it's done it should redirect our browser to the main page of our new repository. At this point the page should show the files that GitHub created for us and the (currently) very simple rendered content of the "**README.md**" file.

Task 3: Install Git on your computer

In order to use Git to manage source code we need to have it installed on our computer. There are many ways to have Git installed and the available ways depend on the kind of operating system you have installed.

This page describes several ways to get Git installed on different operating systems. Please avoid trying to install Git from source code!

You need a text editor to work with Git. By default git uses "vim" as the default editor it launches. "Vim" can be a little hard to use for new users. When you install Git in Windows it will ask you which editor to use. If you don't have a text editor or an IDE you already know and prefer to work with, we recommend you install Notepad++ or Visual Studio Code, which are basic, lightweight editors that are, nevertheless, extendable and good enough for programming.

Task 4: Open a command-line window

While there are many Git graphical clients and most integrated development environments (IDEs) can work with it, it is best to learn Git by first using it from the command line.

If you've never used it before, the command line is a textual interface for interacting with the computer. It works a little bit like a chat where you type in text and the computer prints back the results of the command you've typed.

Note: Command-line interfaces might be frightening at first, but they are a very efficient way to interact with the computer, and many development tools, including all the tools we use in this course, are built for use from command line interfaces.

Different operating systems have different command line interfaces, and most operating systems these days have several different kinds of command line interfaces. In this course we will focus on the basic "**cmd"** command-line interface for *Windows*, the default "**zsh**" for *Mac*, and "**bash"** for *Linux*.

Opening a cmd window in Windows

Probably the quickest way to open a "**cmd**" window in Windows is:

1. Hold down the "**Windows**" key (The one with the windows logo between "Ctrl" and "Alt" on the left side of the keyboard).
2. Without releasing the "**Windows**" key press the "**R**" key.
3. Release both keys - that should open the "Run" window.
4. Type "**cmd**" into the test line in the "Run" window and press "**Enter**"

Another way you can open a "**cmd**" window by clicking the "start" button, selecting "**search**" and typing "**cmd**". The "**Command Prompt**" option should show up in the search results and you can click on it to open the window.

This page lists several other ways to open "**cmd**". It also mentions how to run it with administrative privileges, but you don't have to do that.

Accessing "zsh" on Mac

To access the "**zsh**" command-line interface on Mac, you need to open a "**Terminal**" window. Here are instructions on how to do that.

Accessing "bash" on Linux

(If you're a Linux user, it's unlikely you've never used the shell before, but we're including some basic instructions just in case).

Different Linux distributions can look and feel very different so it's impossible to provide a single way that will work on all of them. We will focus on distributions that provide a Gnome - based interface. Those include the modern versions of Ubuntu, Debian, Fedora and CentOS.

To access "**bash**" from the Gnome desktop interface, you can click on the "**Activities**" button at the top left, type "terminal" into the search box and click on the "Terminal" icon.

Learning to use the command line

If you've never used a command-line interface before, we recommend reading this page from the Django girls tutorial. It mentions several ways to access the command line on different operating systems and teaches some basic commands.

Task 5: Tell Git about yourself

When we create commits with Git, it includes information about who created them automatically. To do that it needs to know our name and email address. We tell it those details by opening a command-line window and typing in commands like the following:

```
git config --global user.email "john.doe@example.com"
git config --global user.name "John Doe"
```

(Please fill-in your actual name and email address)

Note: we add quotes to ensure special characters included in the name and email address such as space and the “@” sign do not influence the execution of the command. They are not included in the final details seen by Git.

Task 6: clone your remote repository

In order to work with the repository we've created in GitHub, we need to create a local copy of it, or *clone* it in Git terminology.

The repository will reside in a directory on your computer, since you may work on many different projects, we recommend keeping things organized by server, account and repository names and to keep all the source code repositories under a common “src” directory (“C:\src” on Windows or “~/src” in your home directory on Mac and Linux, where “~” is a shortcut for our home directory).

For example, if we've called our project in GitHub “beyond-tutorial” and our GitHub account name is “usr1”, in Windows we will store our repository in:

```
C:\src\github.com\usr1\beyond-tutorial
```

On Linux or Mac we will store our repository in:

```
~/src/github.com/usr1/beyond-tutorial
```

Git will create the project directory for us when we clone it, but we still need to create the directory above it.

To create the directory structure we describe above in windows we can use the following command in cmd:

```
C:
md \src
md \src\github.com
md \src\github.com\usr1
```

To create a similar structure in Linux/Mac we can use the following command:

```
mkdir -p ~/src/github.com/usr1
```

We now need to go into the directory we've just created using the "cd" command, on Windows we type:

```
cd \src\github.com\usr1
```

On Linux/Mac:

```
cd ~/src/github.com/usr1
```

Now we can clone our Git repository. To do this we will go into the main page of our repository in GitHub, click on the green "Code" button on the top-right, make sure "HTTPS" is selected and copy the URL shown to us. We can use the convenient clipboard icon shown to the right of the link.

Please do not try to use SSH or the GitHub CLI for now unless you know what you are doing (refer to the note below if you're on Mac or Linux).

Once we've copied the URL, we paste it as a parameter to a "git clone" command (In Windows, to paste in the "cmd" window, you right-click on it). For example, if our account name is "usr1" we will run the following command:

```
git clone https://github.com/usr1/beyond-tutorial.git
```

Git will begin the process of cloning our repository. The output of the process should look like the following:

```
Cloning into 'beyond-tutorial'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
```

Note: The command above uses the "https" URL for cloning the repository because on Windows, that would provide the smoothest working experience since a tool called geit-credential-manager which knows how to deal with the GitHub authentication process is included by default with the Git for Windows installation. On Linux or Mac it is recommended to use the SSH url instead for a smoother experience. Here are instructions about how you create your own SSH key pair and upload it to GitHub.

When we are done cloning the repo we can look at the content of the current directory by using the `dir` command in Windows or the `ls -l` command in Linux/Mac, and confirm a new “beyond-tutorial” directory was created. We should now go into the new directory with the following command (Same on all operating systems):

```
cd beyond-tutorial
```

(Note: to avoid having to type long directory names, you can type the first few letters of the name and then hit the “**tab**” key)

If we want to look at the content of the directory we just went into, using the `dir` command in Windows or the `ls -l` command in Linux/Mac, we will see the files from our repository like so (this example is from Windows):

```
10/12/2020  01:01 PM    <DIR>          .
10/12/2020  01:01 PM    <DIR>          ..
10/12/2020  01:01 PM             1,928 .gitignore
10/12/2020  01:01 PM             1,090 LICENSE
10/12/2020  01:01 PM                76 README.md
               3 File(s)          3,094 bytes
               2 Dir(s)  451,720,884,224 bytes free
```

Note: The `.gitignore` file will not be shown by default on Linux/Mac because files which start with a dot are considered to be hidden. We can see the file if we add the `-a` option to `ls`, but then we will also see the hidden `.git` directory where Git stores the extra information about commit, branches and other things. To see the hidden directory on Windows we can add the `/a` option to `dir`.

Task 7: Editing, committing and pushing

Git and GitHub are flexible tools and there are many ways in which they could be used. In this course we try to teach a process where each code change can get reviewed by team members, automatically checked by a CI system and possibly be rejected. In this process:

- There is one “main” version of the code that is the version that all the team members agreed on.
- That “main” version is stored in the “main” branch in our repository in GitHub (In the past it used to be called the “master” branch, and you will find many repositories that still include such a branch)

- When we make code changes, we don't push them directly into "main". Instead, we create a "pull request" (PR) that contains the changes.
- The other team members can use the GitHub UI to review the PR, add comments and request changes or approve it.
- We can update the PR in various ways to adapt it according to the comments from the other team members.
- When everybody is happy with the PR and approves it, we use the "merge" button in the GitHub UI to merge the change into "main".
- We can decide not to merge the PR at all and simply "close" it.

As changes from other developers can be merged into the "main" branch on GitHub, it is a good idea to make sure that our local copy of it is up-to-date. Otherwise changes that we make may be incompatible with the most recent code.

Updating the "main" branch

To make sure our local "main" branch is in sync with the remote one, we must first go into it by running the following command:

```
git checkout main
```

Note:

- If we're already in the main branch this command will simply tell us so and do nothing
- If we made any changes to local files in the repository without committing them, this command might fail and display an error message

To fetch changes from the remote GitHub repository, we run the following command:

```
git pull --ff-only
```

The `--ff-only` option is a precaution, if we made any changes to our local "main" branch that are different from the changes that were made on GitHub, Git will, by default, try to merge all the changes together locally. This is probably not what we want, so `--ff-only` tells Git to avoid trying to merge and instead print an error message.

Creating a feature branch

Generally we should avoid making any changes to the local "main" branch so it's easy to sync it with the remote branch.

The next thing we're going to do is create a new Git branch so we can use it to store the changes we're going to make separately from the "main" branch. Such a branch is sometimes called a "feature branch" since we use it to work on a new feature for our software. The name we give to a feature branch is not very important, but it's good to give it a descriptive name, especially if we work on multiple features in parallel.

When we're on the "main" branch we can run the following command to create a new branch called "update-readme":

```
git checkout -b update-readme
```

The command above both creates the branch and goes into it (In Git terminology to "go into" a branch is to "check it out"). If we just want to go into a branch that already exists, we can run the command above without the `-b` option.

We can use the `git branch` command to inspect the branches we have in our local repository. It will yield the following output:

```
main
* update-readme
```

We can see that we have both the "main" branch as well as the new branch we've just created "update-readme". The branch we've just created is marked with an asterisk to indicate that it is the branch we're currently working on.

The branch we created is initially identical to our "main" branch, but it will diverge from it as we make and commit changes.

Making changes to the README.md file

We're going to add some content to our `README.md` file which is displayed on our project's main page in GitHub. The `README.md` file is written in a simple markup language called MarkDown. The idea of MarkDown is that for the most part MarkDown files can look very well when viewed as plain-text files in a text editor, but they can also look very well in rich text environment such as websites and include graphical elements such as titles with large fonts, tables and links.

The basic documentation about the MarkDown language can be found here. When GitHub displays MarkDown it supports some extensions over the basic MarkDown specification, and those are documented here.

Use your favorite text editor to open the `README.md` file and change it to look like the following:

```
# beyond-tutorial
A basic tutorial for the tools used in the Beyond course.

This tutorial includes instructions about how to use the following tools:

* [Git](https://git-scm.com/)
* [GitHub](https://github.com/)
* [Vagrant](https://www.vagrantup.com/)
* [PipEnv](https://github.com/pypa/pipenv)
* [Django](https://www.djangoproject.com/)
```

Staging changes for a commit

Once you did your changes and saved them, you need to tell Git about them so it creates a commit that contains them.

The first thing to do is to check that Git detected the changes we've made using the `git status` command. We will get the following output:

```
On branch update-readme
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

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

Git tells us that it sees that `README.md` was changed, but that this change was not *staged* to be committed.

Git's *staging area* exists in order to allow us to control which changes we include in a commit. We can change many files, but only those that we add to the staging area will be included in the commit we create. There are even ways to make it so only certain changes in a given file but not others are included.

Git can also show us what are the exact changes it sees when we run the `git diff` command. In this case we will see several lines shown in green which means Git detects that they were added. It is generally a good idea to run `git diff` and check the changes we've made to ensure we're not going to commit any changes we do not intend to.

To add the changes we've made to the Git staging area, we run the following command:

```
git add README.md
```

Now when we run the `git status` command we will see the following:

```
On branch update-readme
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
```

Git now tells us that the changes to `README.md` are ready to be committed.

We can see exactly which changes were staged by using the `git diff --staged` command.

Creating a commit

Now it's the time to finally create the Git commit we've been talking about. The command to do it is:

```
git commit
```

When we run the command Git will automatically open our text editor with a temporary file that includes some basic instructions about creating a commit message. We need to type our commit message into this file, save it and exit the editor window/tab.

When typing in commit messages, we should generally follow the following rules:

- The first line is the title of the message, it should be no longer than 63 characters.
- The second line should be left empty
- The remaining consist of the body of the message, they should be no longer than 72 characters each. We can include some MarkDown here and it will be displayed by GitHub.

Here is an example of the full commit message file we can write for the commit we're creating:

```
Adding information to the README.md file

Adding information about the types of tools used in developing this
tutorial application.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch update-readme
# Changes to be committed:
#     modified:   README.md
#
```

Once we exit the editor Git will print some brief information about the commit we've just created:

```
[update-readme a9346cd] Adding information to the README.md file
 1 file changed, 9 insertions(+), 1 deletion(-)
```

We can now use the `git show` command to inspect the commit we just created. It will show the commit message and all the changes that were included. It can show quite a bit of text. We can use the arrow keys and the "Page Up" and "Page Down" keys to scroll through it, and "Esc" or "q" to leave the text display and get back to the command line.

Inspecting the commit history

We're going to send our new commit for review in GitHub, before we do that, it is a good idea to inspect the commit history in our local repository and check which commits are going to be sent. We generally do not want our PR to include any commits that do not belong in it as that will make the job of the people reviewing it harder.

The following command shows some brief information about the commits we have in our repository:

```
git log -20 --oneline --decorate
```

The output of the command looks like the following:

```
a9346cd (HEAD -> update-readme) Adding information to the README.md file
8c29a2c (origin/main, origin/HEAD, main) Initial commit
```

This command shows the 20 last commits in the branch we're on (`-20`) while showing one text line per commit (`-oneline`), and adding information about branches and tags (`-decorate`).

In this log we can see:

- The commit we've just created (On the first line)
- The fact that this commit is the top commit in the "update-readme" branch
- The first commit in the repository that GitHub created for us (It's shown on the 2nd line)
- The fact that the first commit is the top commit of the "main" branch as well as the "main" branch in the GitHub repo (It is called "origin" here because that is the default name that Git assigns to the repository we initially clone from).
- The fact that we're currently working on the "update-readme" branch, this is shown by the fact that the "HEAD" pointer points to it.

Note: Any information that is shown about the "origin" repository or any other remote repository (We can configure more of them), is correct to the last time we've run a `git fetch` or a `git pull` command.

When we push our branch and create a PR, all the commits we see between the top commit of the "origin/main" branch, and the top commit of the branch we're pushing will be included in the PR.

Pushing our branch to GitHub and creating a PR.

It's finally time to push (upload) the changes we've made to GitHub. We will push the branch in a way that causes a similarly named branch to be created in the GitHub repository.

The command for pushing the branch is:

```
git push origin update-readme:update-readme
```

The parameters for this command include:

1. The remote repository we're pushing into - here it's `origin`, which as we've said before, is what Git calls the repository we initially cloned from.
2. The name of the local branch we're pushing, given in the first part of the second command parameter, before the ":".

3. The name of the remote branch we're pushing into, here we gave it the same name as the local branch.

Once we've run the command Git will ask us for the username and password to login to GitHub. The way it does that is a bit different for different operating systems. On windows it shows a graphical GitHub login window.

If all goes well, the output of the push command looks like the following:

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 519 bytes | 519.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'update-readme' on GitHub by visiting:
remote:      https://github.com/ifireball/beyond-tutorial/pull/new/update-readme
remote:
To https://github.com/ifireball/beyond-tutorial.git
 * [new branch]      update-readme -> update-readme
```

We can see here that one of the useful things that GitHub does for us is make the Git command print out a link we can use directly to create a PR.

When we copy this link and paste it into the URL line in a browser, it will take us to the "Open a pull request" page. That lets us fill-in the title and the description of the PR.

It's important to spend some time on the PR title and description and make sure they properly describe what our PR includes and why. The better we make those, the easier the job of the reviewers becomes. Many reviewers will reject PRs where the title and message are not clear enough or where they see code changes that were not mentioned in the title or in the description.

If the branch we've pushed only includes a single commit, GitHub will automatically populate the PR title and description for us from the commit message.

We can scroll down the "Open a pull request" page to see all the details about the commits and changes that are included in the PR. It's a good idea to go over it and make sure the PR will include only things we want it to include.

Once we're done, we can click the "Create pull request" button to create the PR.

Task 8: Reviewing and merging PRs

We're currently working as sole developers on our own project, so there is no much point in doing PR reviews for PRs we've created on our own, but it's still a good idea to take the time and get to know the PR review process.

Creating a PR takes us directly to the review page of the new PR we've just created. We typically review other people's PRs, so we can find those by clicking the "Pull requests" link that appears at the top of our repository pages in GitHub.

On the right-hand column of the PR review screen we can see the reviewers section. When we add reviewers, they get an Email notification about our PR. We should generally do that for every PR we create.

Another thing we can see on this column is the "Labels" section. Labels are typically used to mark some relevant information about the PR like whether it is ready to be reviewed, whether the code in it was tested or if it should not be merged for some reason. Different repositories may define and use different labeling schemes according to the development process around the repository.

When we review a PR we should generally follow the following process:

1. Read the PR title and the description:
    - Do they properly describe changes included in the PR?
    - Do they allow us to understand the reasons behind making the changes that were included in the PR?
    - Can we understand where the PR fits in the larger scheme of our project development?
    - Does the PR deal with a whole but singular thing? PRs should generally include a set of tightly-related changes. Things that are not directly related to one another should generally be done in separate PRs.
2. Click on the "Commits" link at the top of the PR page and look at the commits included in it. Many PRs may include only a single commit, but some bigger PRs might include multiple commits to make it easier to understand the code included in them. When reviewing ask yourself:
    - Is the amount of commits appropriate for the scale of change included in the PR?
    - Do the commits describe a set of clear and concrete steps to achieve the changes that were included in the PR?
    - Are there any "junk commits" that include changes that are later rolled back or re-done differently? PR authors should generally remove such commits before sending the PRs.
3. Review the code changes that are included in the PR. You can either view all the changes included in the PR together by clicking on the "Files changed" link or view the changes of each commit individually by clicking on the commit itself. Ask yourself:
    - Does the code change belong in this PR?
    - Do the code changes do what the PR description and commit messages say they do?
    - Is the code well-written? Is it easy to understand?
    - Is there sufficient documentation for the changes being introduced?
    - Was there sufficient testing done to the changes being introduced?

If you have any comments about the code you can click the relevant lines of code to add them. It is advisable to use the “Start a review” and “Add review comments” to group related comments together in a single review.

When you're done adding your comments on the code, you can click the “Finish your review” button at the top-left. In the pop-up window you can also add a general comment about the whole PR and select whether you’re approving the PR to be merged, requesting it to be changed or simply making general suggestions and comments.

When enough reviewers have approved a PR it can be merged. How many is enough depends on our project policy. It is common for projects to require at least two approvals. In the first few weeks of this course it is highly advisable to wait for approval from experienced Red Hatters. To merge the PR one simply needs to click the “Merge pull request” button on the bottom of the “Conversation” page.

By default, merging a PR creates what is called a “merge commit” in Git. When we click the “Merge pull request” button, GitHub lets us put in some details about this commit. The defaults it suggests are usually good enough. We need to finish the merge process by clicking the “Confirm merge” button.

Once we’re done merging the PR, GitHub provides us a button to delete the branch we made it from. It is generally advisable to click it to keep the amount of branches we have in our repo low. Clicking this button only deletes the branch from the GitHub repo, we still retain a copy in our local repository.

Task 9: Updating our local repository (Again)

Merging a PR in GitHub adds changes to the “main” branch stored there. We need to update our local copy with those changes. This could be done with the following commands:

```
git checkout main
git pull --ff-only
```

We can review the commits we now have with the following command:

```
git log -20 --oneline --decorate
```

Tip: We can re-run commands we ran before in the command-line by looking for them using the up and down arrow keys. That can save us quite a bit of typing.

The output of the `git log` command should look like the following:

```
52a941a (HEAD -> main, origin/main, origin/HEAD) Merge pull request #1 from
usr1/update-readme
a9346cd (origin/update-readme, update-readme) Adding information to the README.md
file
8c29a2c Initial commit
```

We can see here:

- The new merge commit that GitHub created
- The fact that it is now the top commit for both the local and the remote “main” branches

Keen eyed observers will notice that:

- We still have a local “update-readme” branch that points to the commit we’ve created
- Git also thinks we have a remote “update-readme” branch even though we told GitHub to delete it (And it did).

To make Git update what it knows about remote branches that were deleted we can run the following command:

```
git remote prune origin
```

We can also delete the local branch we no longer need with the following command:

```
git branch -d update-readme
```

If we now look at the output of `git log -20 --oneline --decorate` it will be much cleaner:

```
52a941a (HEAD -> main, origin/main, origin/HEAD) Merge pull request #1 from
usr1/update-readme
a9346cd Adding information to the README.md file
8c29a2c Initial commit
```

Part 2: Creating a Linux development environment with Vagrant and VirtualBox

What is Vagrant?

One of the issues of working in a software development team is that each developer has a different computer they work on. The different developer computers may and typically do have different versions of operating systems and software tools installed on them. Those differences may introduce difficulties in the development process since they may cause the software we're developing to behave differently.

Vagrant is a tool that is meant to solve this problem. It does it by using Virtual Machines.

What are Virtual Machines?

We usually think of computers as elaborate machines made out of silicone and other materials that can run the software we write for them. Virtual machines are essentially computers that are built out of software instead of silicone.

Virtual machines can do the things that "real" computers can do like run software and let us interact with it, but since they are implemented in software they enable us to do a few things that are much harder to do with real, physical machines.

One thing that virtual machines can do is let us run an operating system inside them that is different than the one we're running on our actual machine.

Another useful quality of virtual machines is that the storage devices (Hard drives) attached to them are usually just files that can easily be copied between computers.

Together, these two qualities enable us to install a particular operating system in a virtual machine as well as other development tools and then give copies of it to all the developers in our team so they all end up working with an identical operating system and set of tools.

What is VirtualBox?

The software that lets us create and run virtual machines is called a "Hypervisor". Vagrant is not in itself a hypervisor, instead, it is a front-end that can use several popular hypervisors that exist in the market.

We chose to use the "VirtualBox" hypervisor in this course for several reasons:

1. It is the default hypervisor that is typically used with Vagrant
2. It has a friendly graphical user interface we can use sometimes to interact with our virtual machines (we don't have to use it often though)

3. It can be installed and used on all the popular operating systems in the market, including Windows, Mac and several versions of Linux.

What are Vagrant Boxes?

The task of installing an operating system on a computer or a virtual machine can be boring and tedious. One thing Vagrant does for us, is save us the trouble of having to do this. The way it does that is by providing us with “boxes”, ready-made virtual machines where the operating system is already installed for us.

The Vagrant Cloud is a website where a wide variety of boxes for many different operating systems and configurations could be found. Since Vagrant is very popular, many operating system vendors upload official boxes of their operating systems to this website.

Task 1: Install Vagrant

We can download the appropriate Vagrant version for our operating system from this page.

Once the Vagrant installation is done, we should be able to run the following command from a command-line window:

```
vagrant --help
```

Running this command should make Vagrant print out a brief help message including instructions about how to use it.

Task 2: Install VirtualBox

This page includes links for downloading VirtualBox for various operating systems.

During the installation process, Windows might pop-up some warning windows about installing device driver software, please confirm for it that the software should be installed.

Tesk 3: Enable virtualization hardware on your computer

While virtualization could theoretically be done purely in software. Modern hypervisors require the use of virtualization hardware in order to gain reasonable performance. Virtualization capabilities are typically built-in to modern CPUs but most computers ship with those capabilities disabled for security reasons.

You will need to access your computer's BIOS settings to enable its virtualization capabilities. To access the BIOS setting you need to turn off your computer, and turn it back on and quickly press a special key. For many computers the key is F1 or ESC, and they typically print a message telling us what their BIOS access key is when they start. If you really don’t know what the key you need to press is, you can probably find it documented in your computer model's user guide or in the guide for your motherboard.

The setting you need to enable in the BIOS is typically called "Intel ® Virtualization Technology", and it could typically be found under a "CPU Settings" or an "Advanced" menu.

Don't worry if it takes you a while to access the BIOS and find the right setting, you only need to do it once for a particular machine, unless you reset its settings back to the manufacturer defaults.

Task 4: Prepare to make changes

In a well managed project, all the details are managed within the Git repository. This includes the configuration for tools such as Vagrant. Since we're going to be adding some files to our repository, we need to create a new feature branch like we did before.

First, open a command line window and go into the directory where our repository is. On windows the commands to do so would be:

```
C:
cd \src\github.com\usr1\beyond-tutorial
```

On Linux/Mac:

```
cd ~/src/github.com/usr1/beyond-tutorial
```

Note: When you type the command above, put in your GitHub account name instead of "usr1".

We need to ensure our "main" branch is up to date and then create a new feature branch:

```
git checkout main
git pull --ff-only
git checkout -b vagrantfile
```

Task 5: Create a Vagrantfile

The way Vagrant works is that we create a file called `Vagrantfile` that contains the instructions about how to create and bring up a virtual machine (VM) for working on our project. After we create the file we put it in our Git repository so that other developers that clone it can use it to bring up identical VMs.

The documentation about how to write a `Vagrantfile` can be found here.

While we can write a Vagrantfile on our own, Vagrant provides us with a convenient command to generate a basic file for us:

```
vagrant init fedora/34-cloud-base
```

This command tells Vagrant to generate a `Vagrantfile` that would bring up a VM that is based on the `fedora/34-cloud-base` box. This box is officially provided by the Fedora community, a Linux development community from which Red Hat's operating system products are derived. The "cloud base" version is a relatively small operating system installation that is suitable for use when developing and running cloud applications.

If we look at our repository directory now, we should see the `Vagrantfile` that was generated for us. If you open it in your text editor, you'll see it contains quite a few lines, but the vast majority of them are commented out. In fact, the only lines that are not commented out should look like the following:

```
Vagrant.configure("2") do |config|
  # ...
  config.vm.box = "fedora/34-cloud-base"
  # ...
end
```

You can spend some time looking at the comments in the file to get some ideas about what could be done with it, but we can keep the file as it is right now.

Task 6: Bringing up a VM with Vagrant

Before we bring a virtual machine up with Vagrant for the first time, it is recommended to open the VirtualBox GUI application, so we can see what Vagrant does for us when we run it. This is not something we usually need to do, but it can be interesting to see it if you have never seen it before.

The command that makes Vagrant bring up the VM is simply:

```
vagrant up
```

Vagrant will print some information about what it is doing, including the fact that it downloads the box file, and brings up (boots) the VM. If you have the VirtualBox GUI open, you can see the VM appearing there and starting up.

You might see Vagrant printing the following error message:

```
There was an error while executing `VBoxManage`, a CLI used by Vagrant
for controlling VirtualBox. The command and stderr is shown below.

Command: ["startvm", "8c9458ec-a144-4862-b4d8-6653c24a1499", "--type", "headless"]

Stderr: VBoxManage.exe: error: Not in a hypervisor partition (HVP=0)
(VERR_NEM_NOT_AVAILABLE).
VBoxManage.exe: error: VT-x is disabled in the BIOS for all CPU modes
(VERR_VMX_MSR_ALL_VMX_DISABLED)
VBoxManage.exe: error: Details: code E_FAIL (0x80004005), component ConsoleWrap,
interface IConsole
```

What it means is that your computer hardware's virtualization capabilities are disabled. Refer to Task 3 above to learn how to enable them.

Task 6: Accessing the VM

Our Vagrant VM is now up, so we can access and use it.

One way we could gain access to the VM is by clicking the "Show" button on the VirtualBox GUI. That button opens up a window that shows us the screen of the virtual machine. Since the operating system we installed is meant for usage in the cloud, it does not actually include a graphical user interface and so the only thing we will see on the screen is a textual "login" line.

Using the VirtualBox GUI is not the typical way we interact with a Vagrant VM. Instead, what we typically use is the following command:

```
vagrant ssh
```

When we run this command it may "think" for a while and then would seem to "finish" and let us type commands again, but we can notice that the prompt (The thing that is printed on the beginning of the line where we can type commands) has changed. It would now look like this:

```
[vagrant@localhost]$
```

What is happening here is that `vagrant ssh` has connected us to the virtual machine and is now providing us with access to its command line interface (While doing it through our own machines command-line window). The commands we will type now will actually run on the virtual machine.

One thing we can do now is check the version of the operating system we have installed in the virtual machine. We can do this with the following command:

```
cat /etc/system-release
```

The output we will get is like so:

```
Fedora release 34 (Thirty Four)
```

For most people, this is probably very different from the operating system they are running on their computer. For Windows users, the `cat` command and the `/etc/system-release` files do not even exist on their system!

Since Vagrant is meant to be used for developing and testing our code, Vagrant automatically copies the files from our repository into the `/vagrant` directory on the virtual machine. We can see the files by running the following command:

```
ls -l /vagrant
```

Note: Since our virtual machine is running Linux, we're using the Linux command for listing the files.

We can exit the virtual machine and get back to our own computer by typing the `exit` command, or by pressing `CTRL+D`.

Note: if the /vagrant directory was not found, see in Task 8 below how to configure vagrant to add it.

Task 7: Shutting down the VM

The VM can take quite a bit of resources from our computer when it is up and running. Therefore, we typically want to "turn it off" when we're not using it. Vagrant provides us with the following command to do that:

```
vagrant halt
```

This command turns off the virtual machine but keeps all its files in place. If we want to resume working on the VM, we can simply run `vagrant up` to turn it back on.

The files of the VM might take up quite a bit of space on our computer. Since we generally automate the setup of the VM, we can safely delete it to clear some space when we want to. This could be done with:

```
vagrant destroy
```

Note that any information we may have put into the VM and not saved somewhere else will be destroyed along with it. It's a good idea to not keep any important information we cannot recreate inside the Vagrant VM.

Task 8: Setting up file sharing

As mentioned before, the purpose of a Vagrant VM is to be used for running and testing the applications we develop. For a smooth development experience, we would like that when we make changes to the files of our application on our computer (Using our favorite text editor or IDE), we will see those changes appear immediately inside the Vagrant VM.

Vagrant does try to set up such automated file sharing by default, but since the underlying technology is a bit complex. It may not work initially, depending on the operating system, hypervisor and box we've chosen.

In the case of the Fedora box we're using, we need to add a configuration option in the `Vagrantfile`. To do this we need to first shut down the VM if we haven't done so already:

```
vagrant halt
```

Now we need to add the following line to the `Vagrantfile`, inside the `do - end` block right below the existing line that starts with `config.vm.box`:

```
  config.vm.synced_folder ".", "/vagrant", type: "virtualbox"
```

Once you added the line you can bring the VM up:

```
vagrant up
```

To check that file sharing actually works do the following:

1. Use `vagrant ssh` to go into the VM
2. Run the `ls -l /vagrant` command in the VM to see your project files
3. Use your operating system's file browser or your text editor to create and save a new file in your project repository directory
4. Run the `ls -l /vagrant` command in the VM again. You should see the file you've just created
5. Delete the file you've created from your computer
6. Run the `ls -l /vagrant` command in the VM again. You should not see the file there any more.

Task 9: Increase the amount of memory in our VM.

By default, Vagrant allocates 500MB of RAM for our virtual machine. This will not be enough for some of the things we need to do, so we need to allocate a bit more. We do this by adding the following lines to the `Vagrantfile`, inside the `do - end` block:

```
config.vm.provider "virtualbox" do |vb|
  vb.memory = "1024"
end
```

Once we changed and saved the `Vagrantfile`, we need to make Vagrant load the new configuration and restart our VM, this can be done with the following command:

```
vagrant reload
```

Task 10: Make Git ignore Vagrant's internal data

If you look at the files in your project repository directory right now, you will probably see a `.vagrant` directory. This is where Vagrant keeps some internal data it needs about the VM.

If you run `git status` you will see that Git also sees that directory and suggests that we will add it to our repository. This is not something we actually want to do, and therefore we can tell Git to ignore that directory. We do this by editing the `.gitignore` file.

Since we previously told GitHub to generate the `.gitignore` file for us, we should already have it in our repository and have a bunch of Python-related lines in it. To make Git ignore the `.vagrant` directory, we need to add the following lines to the end of the file using a text editor:

```
# Vagrant
.vagrant
```

If we now run the `git status` command, we will see that Git no longer shows us the `.vagrant` directory.

Task 11: Commit, push and merge our changes

In order to enable other developers to use the Vagrant VM we've created, we need to share the `Vagrantfile` with them by committing it and pushing it to GitHub.

If you haven't done so already, please create the “vagrantfile” branch as described in Task 3 above. Once you have the branch you can use to following commands to stage the changed files and create the commit:

```
git add .gitignore
git add Vagrantfile
git commit
```

Here is an example for the commit message you can put in when your text editor opens up:

```
Adding a Vagrantfile for spinning up a development VM

Vagrant is a tool for using virtual machines to share development environments.
Learn more about it at:
    https://www.vagrantup.com/

Also making changes to the `.gitignore` file to ignore any data that Vagrant might
create.
```

Once you save the commit message and exit the editor, you can inspect the commit you’ve created and push the branch if all looks well:

```
git show
git log -20 --oneline --decorate
git push origin vagrantfile:vagrantfile
```

Once we’ve pushed our commit we can follow the link we get to create the PR, review and merge it. Don’t forget to click the “Delete branch” button to remove the branch in GitHub once we no longer need it.

After merging the PR we can update our local “main” branch and clean up:

```
git checkout main
git pull --ff-only
git remote prune origin
git branch -d vagrantfile
```

Part 3: Setting up a Django development environment

What is Django?

Django is a software framework for writing web applications using the Python programming language.

What is a software framework?

A software framework is a set of software tools and libraries that helps us write a certain kind of application. There are many software frameworks for many different languages. Some examples you may have heard of include React, Angular, Ruby on Rails and J2EE.

What is software dependency management?

Software that we write typically does not live on its own. Typical software makes use of 3rd party libraries to do its work. The software we write using Django, will, for example, depend on Django itself, as well as many other software libraries that Django needs.

It is important to manage the versions of the libraries our software uses, if we try to run our software with a wrong library version it may not work at all or provide the wrong results.

There are many different ways to manage software dependencies, but the trend in recent years has been for programming languages to include dependency management tools for software that was written using them. One popular dependency management tool that we will use in this course is called “Pipenv”.

Task 1: Install Pipenv

We’re going to install Pipenv in our development environment. Since our development environment lives inside a Vagrant VM, we first need to bring the VM up and enter it.

```
vagrant up
vagrant ssh
```

In the following tasks, unless we say otherwise, assume that all the commands we mention need to be run from inside the Vagrant VM.

To install Pipenv in the Fedora operating system we have to install it on our VM. We can use the following command to do that:

```
sudo dnf install pipenv
```

In this command:

- We use dnf, the Fedora operating system's package manager, to install Pipenv.
- Since administrative (root) privileges are required to run Dnf, we use sudo to obtain them (The Fedora Vagrant box comes with sudo preconfigured to allow us to do that).

Once Pipenv is installed we can run it to get some brief instructions about how to use it:

```
pipenv
```

Task 2: Setup Pipenv for our project

The way Pipenv works is a bit similar to how Vagrant works. It creates and uses two files, `Pipfile` and `Pipfile.lock`, to store information about our software dependencies, and we push those files to our Git repository to enable other developers to obtain and use the same dependencies that we did.

Why two files?

When managing information about software dependencies there are two kinds of information we need to deal with.

One kind of information is the high-level knowledge developers have about their software dependencies. For example, we know that to run our software we need library X version 3 or above.

The other kind of information is the detailed lower-level knowledge that software needs to keep behaving in exactly the same way when we run it in different places and times. For example, the software needs to "know" it runs with library X version 3.0.2, and that the library also needs library Y version 1.20.3 and library Z version 2.3.0.

The `Pipfile` is used to store the high-level human friendly information while the `Pipfile.lock` stores the detailed low-level information. We can generally edit the `Pipfile` (Pipenv provides us with commands to do it, so we don't typically need to do it directly), while `Pipfile.lock` is generated automatically and we should never need to alter it manually.

Both files need to be committed to Git in order to store the full information we need about our software dependencies.

To generate initial versions of the `Pipfile` and `Pipfile.lock`, we need to go to our repository directory and run a simple command:

```
cd /vagrant
pipenv --python 3
```

This command tells Pipenv that our project would be using Python 3. If we now use the `ls` command to look at our project files, we will see that a `Pipfile` was created for us. A

`Pipfile.lock` file was not yet created because we did not ask for any software dependencies yet.

Task 3: Installing Django

To install Django for our project, we run the following command:

```
pipenv install django
```

This command will both install Django itself, and create the `Pipfile.lock` file to store the information about which exact version was installed.

We can verify that Django was installed successfully and see which version was installed with the following command:

```
pipenv run python -m django --version
```

Note: Any Python-based command we run in our project needs to be run via Pipenv in order to have access to the dependencies it manages for us. If we try to run the command above without the `pipenv run` part, it will fail with an error like the following:

```
/usr/bin/python: No module named django
```

Task 4: Bootstrapping a Django website

One of the things Django provides us with is the ability to quickly bootstrap our website and generate a skeleton application with a single command.

```
pipenv run django-admin startproject beyond_tutorial .
```

We provide Django with a name for our project (“`beyond_tutorial`”). That name serves as the name of the directory Django generates. Since the directory works as a Python module, the name must be a legal Python identifier name and cannot include characters such as spaces or hyphens.

The second parameter we provide to the command (“`.`”), tells Django to create the project files in the current directory (You can think of “`.`” as a pointer to the current directory, and you should be in “`/vagrant`”, the project’s repository directory, when you run the command above). If we don’t provide it, Django would create an extra directory that we don’t need.

If we look at the files we have right now, we will see that we indeed have a new `beyond_tutorial` directory as well as a `mange.py` script. That script is what we can use to interact with Django to develop and test our application. We can ensure it works by asking it to display its help message:

```
pipenv run ./manage.py --help
```

Task 5: Providing HTTP access into our VM

Django is generally used to create web (HTTP) applications, since we're going to run our Django application on our Vagrant VM, we need to create a way for us to access it from our machine.

Even though the Vagrant VM is just another software program running on our computer, it has its own (virtual) networking hardware and its own network software stack. From the point of view of other software running on our computer it looks just as if it was a different machine connected over the network. Therefore we need to use some advanced networking techniques to gain access to it, but Vagrant mostly hides those away.

Vagrant enables us to use a technique called "port forwarding", with port forwarding we can make a network TCP port on our computer act as if it was a port on another computer (In our case, the virtual machine).

To setup port forwarding for port 8000 (The default port Django runs in at development time), we need to add the following lines to the `Vagrantfile`, inside the `do - end` block:

```
config.vm.network(
  "forwarded_port", guest: 8000, host: 8000, host_ip: "127.0.0.1"
)
```

We need to reload our Vagrant configuration and restart the VM. Since we are logged in to the VM at this point we need to first exit it with:

```
exit
```

Once we exit the VM and are back in our own machine's command line (Note what the prompt says!), we can run the following command to reload the Vagrant configuration and restart the VM:

```
vagrant reload
```

Once the VM is done restarting, we can log back into it with:

```
vagrant ssh
```

Finally we go back to our project code directory with:

```
cd /vagrant
```

Task 6: Running our Django application

To run our skeleton application, we use the following command:

```
pipenv run python manage.py runserver 0.0.0.0:8000
```

Note: By default, Django tries to be secure when it runs in development mode and blocks connections that do not originate from the machine it runs on. Since we want to connect to Django from our host computer, which is considered to be a remote machine from the virtual machine's point of view, we need to add the `0.0.0.0:8000` parameter to `runserver`.

The application server will start up, print a few messages like the following, and then stop and listen for user interaction.

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly
until you apply the migrations for app(s): admin, auth, contenttypes,
sessions.
Run 'python manage.py migrate' to apply them.
October 16, 2020 - 05:37:10
Django version 3.1.2, using settings 'beyond_tutorial.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

You can notice a warning about unapplied migrations. The migrations are used to set up your application's database. Since we did not create any data model classes for our application, this is not important at this phase, and we can ignore the warning.

We can now use the address Django displays to access our application. Open a web browser and navigate to:

ht27.0.0.1:8000/tp://1

We will get back a page congratulating us on a successful Django installation. If we look at the command-line window that runs Django, we will also see a log line printed that notified us about the access that was made to the application. Try moving the windows around so you can see both the browser windows and the command line window and then refresh the browser. You should see additional log lines getting added as you do the refresh.

We can shut down the Django application by pressing Ctrl+C in the command-line window. After we do that, the browser will show an error message if we try to refresh it.

Task 8: Automating our environment

We're going to set things up so that running `vagrant up` will be the only thing we need to do to bring up a functional development environment before we can start writing code.

Most of the settings we've made are saved as files into our repository and therefore we do not need to re-make them if we shut down and delete our VM, or start it on a different computer. There are a few things that we did that only modified the VM, and therefore we will need to have our automation re-do them as needed. Those things are:

- Installing the Pipenv tool
- Telling Pipenv to install our software dependencies (Pipenv did that automatically when we created the `Pipfile` and `Pipfile.lock` via the `pipenv install` commands, but a different command is needed when we already have the files and simply want to get the dependencies listed in them.
- Starting our application

For automating VM setup, Vagrant provides us with the "provisioning" feature. There are many ways to provision things in a vagrant VM, but one of the simple ones is to have Vagrant simply run a shell script for us when the VM starts up for the first time.

Let's create our provisioning script by writing the following content into a file called "`setup.sh`":

```
#!/bin/bash -ex
# The -e option would make our script exit with an error if any command
# fails while the -x option makes verbosely it output what it does

# Install Pipenv, the -n option makes sudo fail instead of asking for a
# password if we don't have sufficient privileges to run it
sudo -n dnf install -y pipenv

cd /vagrant
# Install dependencies with Pipenv
pipenv sync --dev

# run our app. setsid, the parentheses and "&" are used to perform a "double
# fork" so that out app stays up after the setup script finishes.
# The app logs are redirected to the `runserver.log` file.
(setsid pipenv run \
    python manage.py runserver 0.0.0.0:8000 > runserver.log 2>&1 &)
```

Note: Google docs and other word processors treat quotes and apostrophes in strange ways. If you try to copy and paste the code above, it may not work and fail with strange errors. Please, take the time to manually type it into your text editor.

To make Vagrant run this file for us, we add the following line to the `Vagrantfile`:

```
config.vm.provision "shell", path: "setup.sh", privileged: false
```

By default, Vagrant runs provisioning scripts in privileges mode (With the “root” user), since our script mostly needs to run unprivileged commands, we set the `privileged` option to `false`.

The only thing left to do now is destroy our VM and recreate it, remember you need to `exit` the VM back to your own operating system’s command line to do that:

```
vagrant halt
Vagrant destroy
vagrant up
```

If all goes well, when the VM comes back up you’ll be able to use your browser to see the Django welcome page at http://127.0.0.1:8000/. You’ll also see the output of your `setup.sh` script as part of the `vagrant up` output.

Task 9: Commit push and merge your changes

It’s time to commit all the new files we’ve added and the changes we’ve made. If you’re still logged into your VM via `vagrant ssh` don’t forget to exit it before trying to run any Git commands.

Here is the brief reminder of the process that needs to be done:

1. Create a new feature branch with `git checkout -b` (So far, we’ve always created the branch prior to making any changes, but it's also possible to do that afterwards, as long as we create the branch from the current commit).
2. Examine which files had been changed or added with `git status`
3. Stage the files for committing with `git add` (Note: Your project directory might include the `nohup.out` log file, it should not be committed, try to make Git ignore it!; Also note: adding a directory adds all the files in it, except the ignored ones)
4. Commit the changes
5. Review your commits with `git show` and `git log`
6. Push the branch to GitHub
7. Follow the link displayed by the push command to create a PR
8. Review and merge the PR
9. Update the local main branch and delete the un-needed branches

Some of the files we've added have been created automatically by running the tools. It's usually a good idea to separate auto-created changes from manually written ones, by putting them in separate commits or even separate PRs. In this case we made a fairly small amount of actual changes, so it can be sufficient to include a commit message like the following:

```
Adding a Django development environment

- Setup [Pipenv][1] to install and manage Django and other Python
  dependencies
- Bootstrapped an empty [Django][2] application
- Setup Vagrant to automatically bring our application up as well as
  make it accessible from a web browser

After running `vagrant up` the application can be accessed at:

> http://127.0.0.1:8000/

**Note:** Most changes had been auto-generated, only `Vagrantfile` and
`steup.sh` had been edited manually.

[1]: https://github.com/pypa/pipenv
[2]: https://www.djangoproject.com/
```

Part 4: Adding some CI

The essence of CI is providing rapid feedback to developers about the quality of the code they write so they can find and resolve issues quickly.

We're going to be adding some Python code to our repository, so we'd like to at least be able to verify that the code we add conforms to proper Python syntax and coding standards. To do this we will use the "Flake8" tool and set things up so that it scans our code automatically as we send PRs and provides feedback on the PR review screen.

Task 1: Install Flake8

We can install Flake8 via Pipenv, just like any other tool (The following command should be run from inside the `/vagrant` directory on our Vagrant VM):

```
pipenv install --dev flake8
```

We add the `--dev` option to tell Pipenv that Flake8 is a development environment dependency. That way we can avoid having it installed in production environments. (Note: This will add flake8 to the dev_packages section of the Pipfile, which is called when using `pipenv sync --dev` (see example below).

Task 2: Inspect our code with Flake8

Let's do a preliminary inspection of our code, this could be done with the following command:

```
pipenv run flake8
```

The command will output several error messages like the following:

```
./beyond_tutorial/settings.py:89:80: E501 line too long (91 > 79 characters)
```

This error message indicates that Flake8 found some lines of code that are longer than the 80 character limit it enforces by default. We can add a parameter to tell it to expect longer lines:

```
pipenv run flake8 --max-line-length 120
```

This time Flake8 would return no output. That is its way of indicating all the code passes its checks.

Task 3: Add GitHub actions configuration

We set up GitHub actions to run flake8 on the code as we submit new PRs. The GitHub actions configuration files reside in the `.github/workflows` directory, so we must first create it:

```
mkdir -p .github/workflows
```

Please note that the directory needs to be created inside our code repository so use `cd` to go to where it's located if you're not already there.

We can now create a file called `.github/workflows/flake8.yml` and place the following code in it:

```
name: Flake8
on: pull_request
jobs:
  flake8:
    name: Check code with Flake8
    runs-on: ubuntu-20.04
    container: fedora:34
    steps:
      - name: Install Pipenv and Git
        run: dnf install -y pipenv git
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Setup environment
        run: pipenv sync --dev
      - name: Run Flake8
        run: pipenv run flake8 --max-line-length 120
```

We use a Fedora 34 container to get a running environment that is similar to what we've set up in Vagrant. GitHub actions make it very easy to use containers by requiring just a single YAML line to activate them.

Note: YAML is an *indentation-sensitive* language. Indentation is used to specify which configuration elements are contained within other elements. When you create the YAML file for your project, make sure you indent the lines in exactly the same way as they appear above. Also be careful about having your text editor automatically introduce tab characters into the file. Most text editors have various configuration options that deal with their behavior with regards to tab characters.

Task 4: Commit the files and send a PR

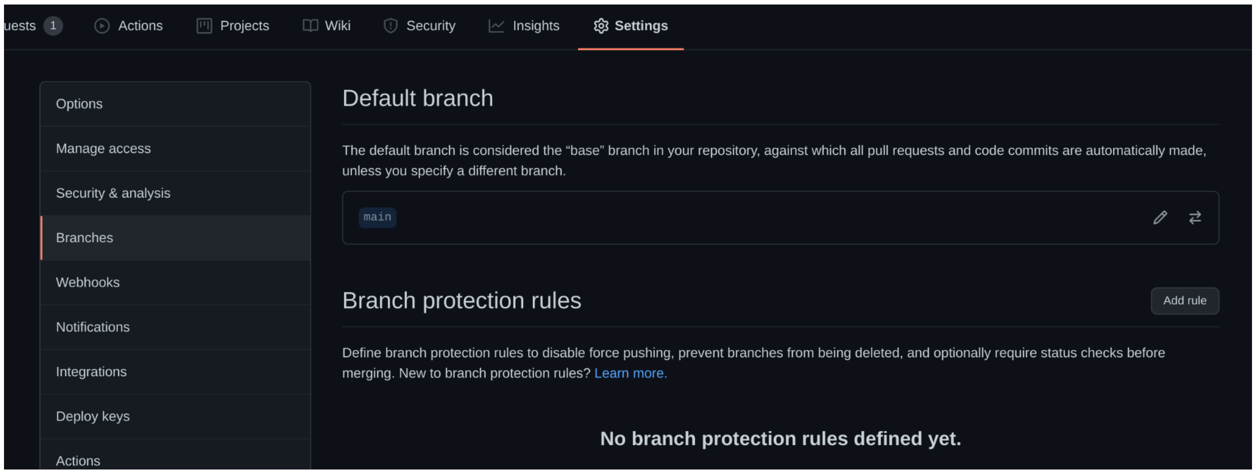
We'll stage the files we've modified, commit them, push a branch and send a PR. Make sure you add all the files we've modified to the commit including `Pipfile`, `Pipfile.lock` and the new `flake8.yml` file.

As soon as you create the PR, you should be able to scroll down and see the new action running in the “checks” section. You will be able to click on “details” and see the various action steps run and produce output.

When the check finishes successfully, merge the PR. Merging it will make the test run on all PRs sent from now on.

Task 5: Using the CI to Protect the main Branch

The main branch is the formal source of our project, and we must protect it. GitHub allows us to protect branches by adding a set of rules. In your beyond-tutorial github project, go to “settings” and select “branches”, then click the “add rule” button:



Default branch

The default branch is considered the “base” branch in your repository, against which all pull requests and code commits are automatically made, unless you specify a different branch.

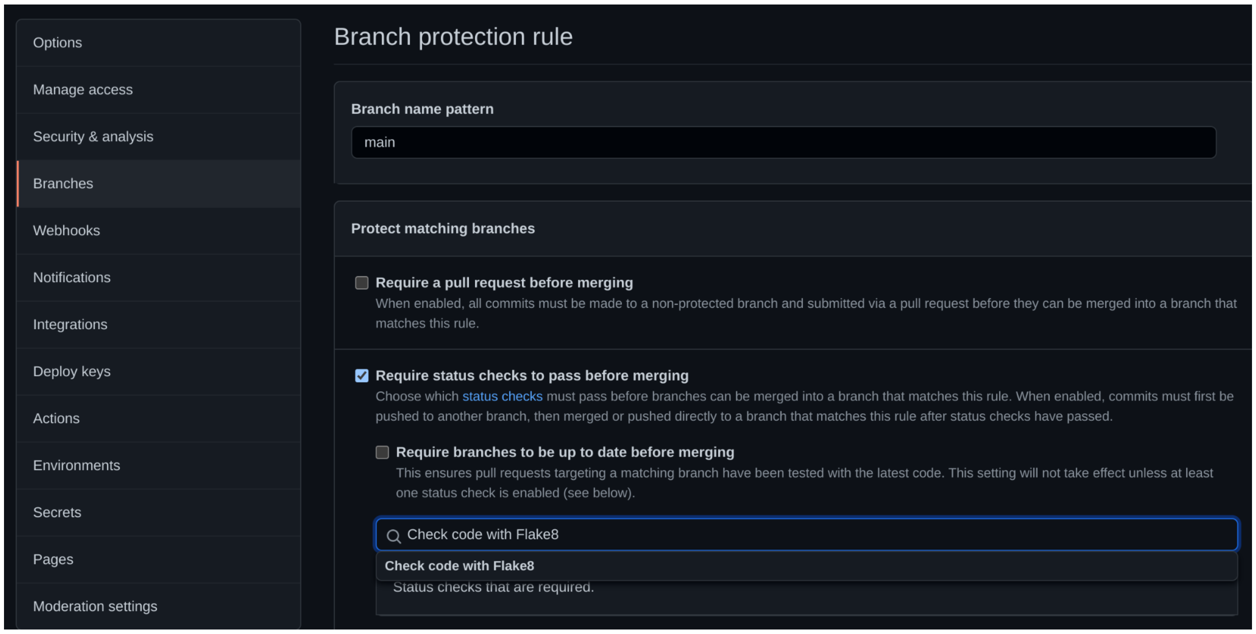
main

Branch protection rules

Define branch protection rules to disable force pushing, prevent branches from being deleted, and optionally require status checks before merging. New to branch protection rules? Learn more.

No branch protection rules defined yet.

Write “main” in the “Branch name pattern” field. You can select and configure rules for the main branch. You can read about these rules to learn more, but for now, just select “Require status checks to pass before merging”. In the search field that now appears, write the name of our CI test from the yaml file: “Check code with Flake8”, and select the same name from the dropdown.



Branch protection rule

Branch name pattern

main

Protect matching branches

☐ **Require a pull request before merging**
When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

☑ **Require status checks to pass before merging**
Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

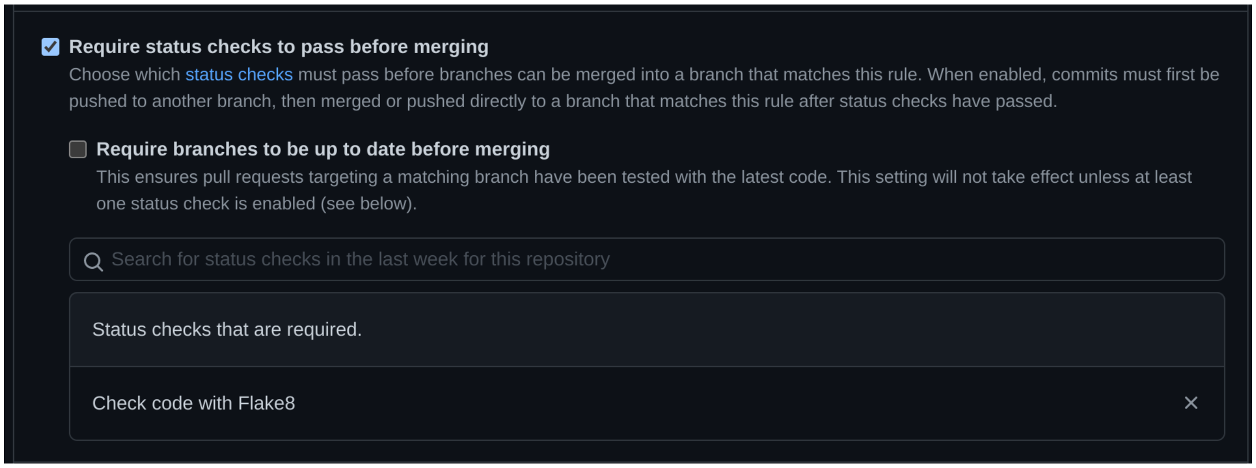
☐ **Require branches to be up to date before merging**
This ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled (see below).

Check code with Flake8

Check code with Flake8

Status checks that are required.

It should now look like this:



☑ **Require status checks to pass before merging**
Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

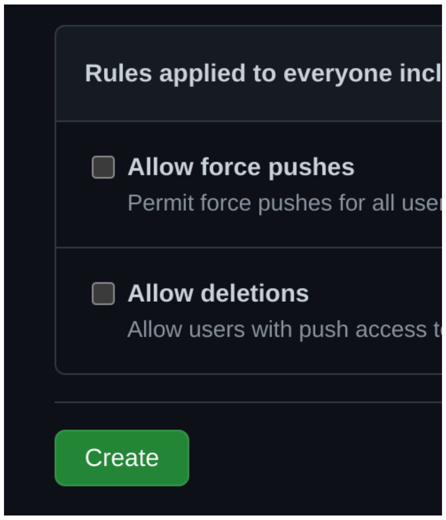
☐ **Require branches to be up to date before merging**
This ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled (see below).

Search for status checks in the last week for this repository

Status checks that are required.

Check code with Flake8 ✕

Click the “Create” button at the bottom of this page to create the rule


Rules applied to everyone incl

Allow force pushes
Permit force pushes for all user

Allow deletions
Allow users with push access t

Now, the CI action not only runs, but it is a gate for merging PRs into the main branch. We can add more checks if we want. For example: run unit tests and require them to pass before merging the PR.

Task 6: Send bad code to test CI functionality

The purpose of CI is to fail when issues exist in the code. We must ensure our CI action can fail by sending some illegal code.

Before making any of the changes below, please ensure you've merged the CI code we created before and updated your local “main” branch. This is to ensure the next changes we’ll send will include the updated CI configuration.

Now lets create a `bad.py` file with the following content:

```
<<<This file is not a legal Python file>>>
```

We can make sure the file would indeed fail a Flake8 check by running it:

```
pipenv run flake8 --max-line-length 120
```

We will get output like the following:

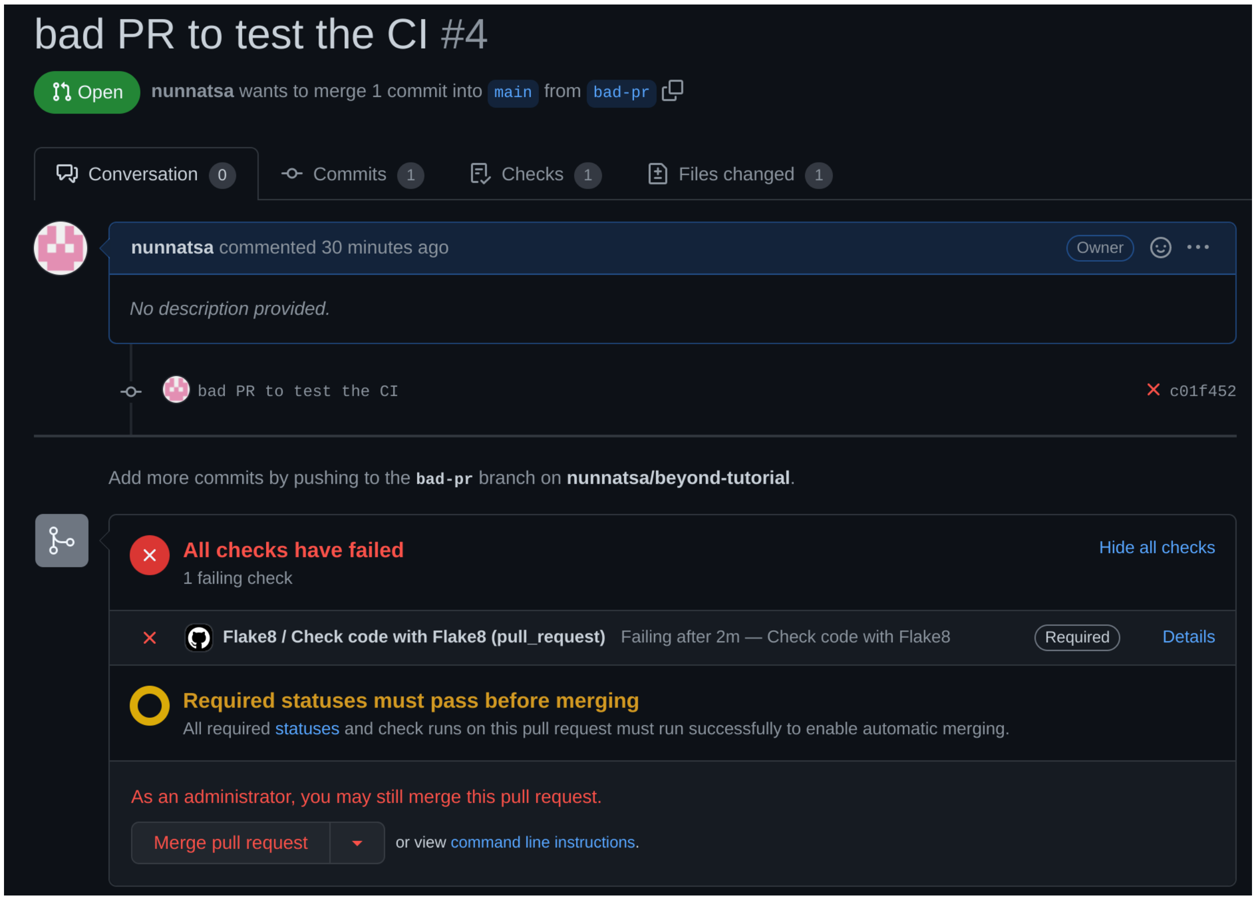
```
./bad.py:1:1: E999 SyntaxError: invalid syntax
./bad.py:1:3: E225 missing whitespace around operator
./bad.py:1:40: E227 missing whitespace around bitwise or shift operator
```

Despite the errors, let’s commit the file into a feature branch with Git, push it and create a PR.

Our new CI action should start running on the new PR, take a little while to do its work, and finally indicate a failure.

Once you see the failure as expected, you can close the PR, since we have no intention to merge it.

Since we added the branch rule in Task 5, our PR is failing, and we can't merge the PR until we'll fix it:


bad PR to test the CI #4

Open nunnatsa wants to merge 1 commit into `main` from `bad-pr`

nunnatsa commented 30 minutes ago

No description provided.

`bad PR to test the CI` ✗ `c01f452`

Add more commits by pushing to the **bad-pr** branch on **nunnatsa/beyond-tutorial**.

All checks have failed
1 failing check

✗ **Flake8 / Check code with Flake8 (pull_request)** Failing after 2m — Check code with Flake8 Required Details

Required statuses must pass before merging
All required statuses and check runs on this pull request must run successfully to enable automatic merging.

As an administrator, you may still merge this pull request.

or view command line instructions.

Part 5: Adding some features to our website

Now that we have our development environment set up and ready, we can add some features to our website. We will create a (very) basic one-page message board application where users can post messages and see messages by other users.

A Django website is built out of one or more “apps”. Each Django app typically contains:

- **Model classes** that describe the data model of the application
- **Views** that describe the processes by which users view and interact with the data
- **Templates** that describe how data is sent and displayed to users
- **Routes** that describe how web addresses map to views

Additionally a Django app can contain:

- **Migrations** that describe how to setup the database for the application
- **Static files** that can be sent to the user’s browser along with content that is generated via templates. Such files may include image and CSS files.
- **Form classes** that represent forms shown in the UI.

Task 1: Bootstrapping a Django app

Just like it provided us with a command to get started with our website, Django also provides us with a command to quickly get started with an application. To run the command we must first connect to our VM:

```
vagrant ssh
```

And go to our application directory:

```
cd /vagrant
```

Now we can run the following command to bootstrap our app:

```
pipenv run python manage.py startapp msgboard
```

The command will create the `msgboard` directory and put a few files and directories in it.

Task 2: Registering our app with Django

After we generate our app, we need to tell our Django website about it. We do that by modifying the `beyond_tutorial/settings.py` file. We modify the lines that start with `INSTALLED_APPS` to add our app to the list of apps Django knows about. The end result with our modification should look like the following:

```
INSTALLED_APPS = [
	'django.contrib.admin',
	'django.contrib.auth',
	'django.contrib.contenttypes',
	'django.contrib.sessions',
	'django.contrib.messages',
	'django.contrib.staticfiles',
	'msgboard.apps.MsgboardConfig',
]
```

Task 3: Creating model classes for our app

Django model classes describe the structure of our application's database. And provide us with ways to access it. Each model class corresponds to a database table, and it contains attributes that correspond to the table fields. When we interact with the database we can use class methods to query it and get back class instance objects that correspond to database records. We can also modify the object to alter the database records or create new objects to create new database records.

Our application model is going to be rather simple. We'll have a single table that stores the messages in our message board, so we'll only need a single model class.

To create our model class we'll change the `msgboard/models.py` file to look like the following:

```
from django.db import models
from django.utils import timezone


class Message(models.Model):
    author = models.CharField(max_length=200)
    text = models.TextField()
    date = models.DateTimeField(default=timezone.now)
```

Note: Like YAML, Python is an *indentation-sensitive* language. Make sure you keep line indentation in your files similar to what you see in the examples here and consistent.

Task 4: Creating database migrations

Before we can use our newly created model class, we need to create the database table that stores the data it represents. In Django, this task is done via “database migrations”. Database migrations are scripts that can create or alter the database structure, and Django provides us with a way to run them as needed. Django also provides us with the ability to automatically generate database migrations that correspond to changes we make to our model classes.

To create the database migrations for our new model class, we run the following command:

```
pipenv run python manage.py makemigrations msgboard
```

The command will create the `msgboard/migrations/0001_initial.py` file. If we look inside it we can see the command for creating our database table.

Note: If we did not register our app as specified in Task 2 above, the command will fail with the following error message:

```
No installed app with label 'msgboard'.
```

Task 5: Adding test data

When developing an application it's useful to have some test data in it that we can work with at development time.

We can create a custom database migration file that will automatically add some test data for us. To do that, we'll create the `msgboard/migrations/0002_test_data.py` file and put the following content in it:

```
from django.db import migrations, transaction


class Migration(migrations.Migration):

    dependencies = [
        ('msgboard', '0001_initial'),
    ]

    def generate_data(apps, schema_editor):
        from msgboard.models import Message

        test_data = [
            ('Test User1', 'A simple test message'),
            ('Test User2', 'Another simple test message'),
        ]

        with transaction.atomic():
            for author, text in test_data:
                Message(author=author, text=text).save()

    operations = [
        migrations.RunPython(generate_data),
    ]
```

Task 6: Running the database migrations

To run the database migrations we've just created we run the following command:

```
pipenv run python manage.py migrate
```

In the output of the command we can see Django runs the migrations we've created as well as several other migrations that are built-into Django and the libraries it provides for us.

After we've run this command we can notice a file called `db.sqlite3` was created in our project directory. That file stores our database data. We need not and should not commit this file to Git, so it's a good idea to add it to the list of files in the `.gitignore` file.

We can inspect our newly created database using the following command:

```
pipenv run python manage.py shell
```

That command runs an interactive Python shell that lets us interact with our application's objects.

We can type in the following Python commands to query our database data:

```
from msgboard.models import Message
messages = Message.objects.all()
[(str(m.date), m.author, m.text) for m in messages]
```

Once we're done with our interactive shell, we can exit it with:

```
exit()
```

Task 7: Creating a view for displaying our data

To let users view the messages we have stored in our database, we need to create a view function that will query the data and send it to the users, a template to generate HTML from the data and a route to direct user requests to the view function.

Creating the view function

We'll edit the `msgboard/views.py` file so it looks like the following:

```
from django.shortcuts import render
from .models import Message


def board(request):
    messages = Message.objects.order_by('-date')
    return render(request, 'msgboard/board.html', {'messages': messages})
```

Creating the template

To create the template we must first create a directory for it:

```
mkdir -p msgboard/templates/msgboard
```

(You can also create the directory in your operating system's file manager or from your text editor if it lets you do that)

Now we can create the `msgboard/templates/msgboard/board.html` template file:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Beyond message board</title>
    <style type="text/css">
      .author, .eom { color: grey }
    </style>
  </head>
  <body>
    <h1>Beyond message board</h1>
{% for message in messages %}
    <div class="message">
      <p class="author">
        Posted by {{message.author}} at {{message.date}}
      </p>
      <p>{{ message.text }}</p>
    </div>
{% endfor %}
    <p class="EOM">(End of messages)</p>
  </body>
</html>
```

Creating the route

To create the route, we edit the `beyond_tutorial/urls.py` file so it looks like the following (Please note that the file resides in our site settings directory and not in the app directory):

```
from django.contrib import admin
from django.urls import path
from msgboard import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.board, name='board')
]
```

Testing our changes

If all goes well we should be able to refresh our browser window that was looking at the Django welcome page and see our application page instead. The Django development server monitors our code for changes and reloads it automatically.

Sometimes we can make code changes that can cause the server to crash when it tries to load them. If that's the case we will get a connection error in our browser window instead of a proper web page. We can make Vagrant rerun our setup script and restart our server by running the following command (Note that this command must be run from the command line of our computer and not from inside the VM!):

```
vagrant provision
```

Task 8: automating migrations in Vagrant

Since our application has a data model now, running the migrations must now be done as part of the overall process of starting it up. To do that we'll alter the "`setup.sh`" that Vagrant runs for us.

In the file, we'll add the line to run the migrations after we run `pipenv sync` and before the final few lines that do the work of bringing up our application. When we're done the file should look like the following

```
#!/bin/bash -ex
# The -e option would make our script exit with an error if any command
# fails while the -x option makes verbosely it output what it does

# Install Pipenv, the -n option makes sudo fail instead of asking for a
# password if we don't have sufficient privileges to run it
sudo -n dnf install -y pipenv

cd /vagrant
# Install dependencies with Pipenv
pipenv sync --dev

# Run database migrations
pipenv run python manage.py migrate

# run our app. Nohup and "&" are used to let the setup script finish
# while our app stays up. The app logs will be collected in nohup.out
nohup pipenv run python manage.py runserver 0.0.0.0:8000 &
```

Task 9: Allowing the users to post

To allow our users to post new messages on the Board we will need to add a form object that allows user data input to our application.

We will also need to alter our page template to display the message posting form and our view to be able to accept form data and store it in the database.

Adding a form object

The form objects of Django save us the trouble of writing code to perform repetitive tasks such as generating HTML to display forms, validating form data and saving it to the database.

To create a form object for our message board messages, we will create the `msgboard/forms.py` file, and put the following content in it:

```
from django import forms
from .models import Message


class MessageForm(forms.ModelForm):
    class Meta:
        model = Message
        fields = ('author', 'text')
```

For the most part, Django can generate the object from our existing model class, we just need to tell it which fields should be included in the form.

Altering the view

We will alter our views file, `msgboard/views.py` to look like the following:

```
from django.shortcuts import render, redirect
from .models import Message
from .forms import MessageForm


def board(request):
    messages = Message.objects.order_by('-date')
    if request.method == "POST":
        form = MessageForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('board')
    else:
        form = MessageForm()
    return render(request, 'msgboard/board.html', {
        'messages': messages,
        'form': form,
    })
```

The new function code checks whether the incoming HTTP request contains form POST data. If so, it validates the data and stores it in the database, otherwise it just displays an empty form.

Note the use of the `form.save()` method, which saves us the trouble of creating the model object and filling it with form data on our own.

Displaying the form on the page

To display the form on our message board page, all we need to do is add the following code to the `msgboard/templates/msgboard/board.html` template file. The code should be added after the `<h1>` line:

```
<div class="newfrm">
    <p class="newmsg">New message:</p>
    <form method="POST">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Post">
    </form>
</div>
```

Testing our changes

When we refresh the browser after making our changes, we should see a “new message” form displayed at the top of our message board page. If we fill-in the form fields and click “Post”, we should see our new message displayed on the page. If we try to click on “Post” without filling-in the required details, we should see an error message displayed.

Depending on how you edited your code, the test server may crash while you work, and you will get a browser error page when you try to refresh. See the comment near the end of Task 7 above for an explanation about how to bring it back.

Task 10: Checking our code

Before sending our new code to GitHub, we should check it to find any issues. The first check we can do is to run the Flake8 tool:

```
pipenv run flake8 --max-line-length 120
```

The tool might find a couple of issues in the files Django generated for us, we can resolve those for now by commenting-out the unused import lines that cause the issue.

```
./msgboard/tests.py:1:1: F401 'django.test.TestCase' imported but unused
./msgboard/admin.py:1:1: F401 'django.contrib.admin' imported but unused
```

Another test we should perform is to destroy our Vagrant VM, bring it up again and make sure the app and all its features work. That way we can be sure we did not ruin the work environment for other developers.

```
vagrant halt
Vagrant destroy
vagrant up
```

Note: To properly test everything you need to also delete the `db.sqlite3` file before bringing the VM up again. Otherwise the same file will be reused by the newly started VM, and the migrations will be skipped.

When everything works well, the `vagrant up` output should show the migrations running before the application server is started.

Task 11: Committing and pushing our changes

It is now time to commit and push all the changes we've made. The resulting commit and PR are going to be quite big. When developing in a team, it's a good idea to split such large changes into smaller ones so that the review process may be more efficient.

One way to split apart such a big change is along the lines by which we divided it into tasks:

1. First create and push the data model classes and the migrations
2. Then create some read-only view functions and templates to show how the data would ultimately be presented
3. Finally add the ability to add or alter the data.

Each separate set of changes should be sent as a separate PR, reviewed and merged before the next one is sent.