INDEX

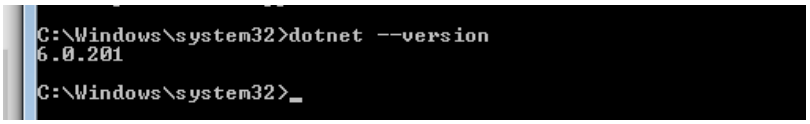
Practical 1

Install .NET Core and create real time microservice with ASP.Net

Solution:

1) **Download link: https://download.visualstudio.microsoft.com/download/pr/deb4711b-7bbc-4afa-8884-9f2b964797f2/fb603c451b2a6e0a2cb5372d33ed68b9/dotnet-sdk-6.0.300-win-x64.exe**
2) **Check everything installed correctly**

Once you've installed, open a **new** command prompt and run the following command:

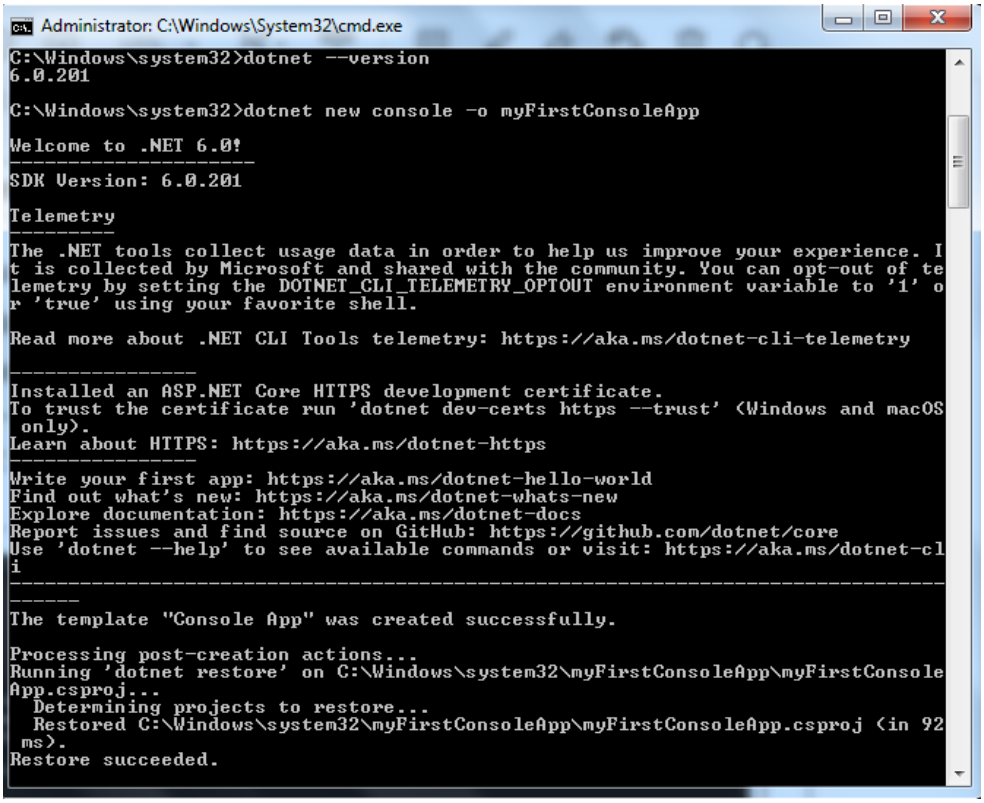


If the above command works, then the basic requirements for .NET Core are installed on the workstation. In a Windows machine, you should be able to find the .NET Core installed runtimes in the following directory:

"C:\Program Files (x86)\ dotnet\shared\Microsoft.NETCore.App\3.1.4".

3) **Building a Console App**

Open the command prompt and go to the desired folder and type the dotnet new console command as shown in Fig. The dotnet new console command is used to generate the standard console application.



The folder myFirstConsoleApp consists of two files: the project file which defaults to <directory name>.csproj which in our case is called myFirstConsoleApp.csproj and Program.cs.

4) The .csproj extension file represents a C# project file that contains the list of files included in a project along with the references to system assemblies. The myFirstConsoleApp.csproj file is shown in Fig.

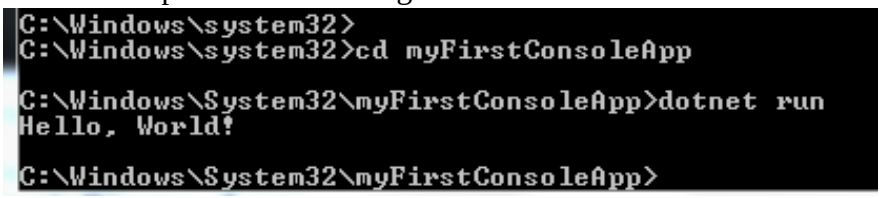
```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

The Program.cs file contains the method Main, which is the entry point of the ASP.NET Core applications. All the .NET Core applications basically designed as console applications. The Program.cs file is shown in Fig.

Change the directory to the newly created project folder and run the dotnet run command to view the output as shown in Fig.



Practical 2

Building ASP.NET Core App

Solution:

1. **continue practical 1, Go to the folder myFirstApp and open the file Program.cs in notepad and make the changes in the file as shown**

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Http;
using System.Configuration;

namespace myFirstConsoleApp
{
        class Program
        {
        static void Main(string[] args)
{
var config=new ConfigurationBuilder().AddCommandLine(args).Build();
var host=new
WebHostBuilder().UseKestrel().UseStartup<Startup>().UseConfiguration(config).Build();
host.Run();
}
        }
}
```

2. **Go to the folder myFirstApp and create the file Startup.cs in notepad and type the code as shown**

```
using System;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Http;
using System.Configuration;

namespace myFirstConsoleApp
{
        public class Startup
        {
        public Startup(IHostingEnvironment env)
        {
        }
```

```
public void Configure(IApplicationBuilder app,IHostingEnvironmentenv,ILoggerFactory lf)
{
app.Run(async (context)=>{await context.Response.WriteAsync("Hello World!");});
        }
        }
}
```

3. **Go to the command prompt and execute the following commands.**

```
dotnet add package Microsoft.AspNetCore.Mvc
dotnet add package Microsoft.AspNetCore.Server.Kestrel
dotnet add package Microsoft.Extensions.Logging
dotnet add package Microsoft.Extensions.Logging.Console
dotnet add package Microsoft.Extensions.Logging.Debug
dotnet add package Microsoft.Extensions.Configuration.CommandLine
```

4. **Depicts the file myFirstApp.csproj.**

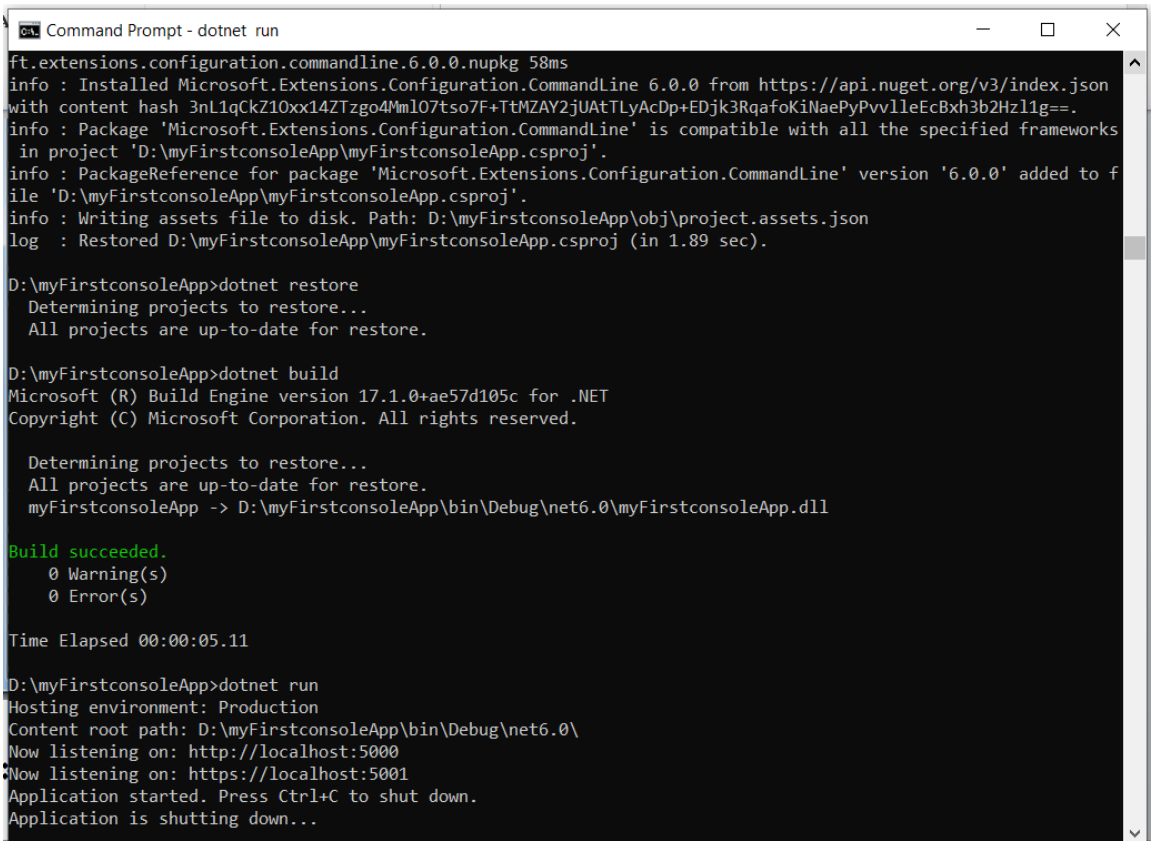
```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net6.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
         <Nullable>enable</Nullable>
 </PropertyGroup>
 <ItemGroup>
        <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
        <PackageReference  Include="Microsoft.AspNetCore.Server.Kestrel"  Version="2.2.0"
        />
        <PackageReference Include="Microsoft.Extensions.Configuration.CommandLine"
        Version="6.0.0" />
        <PackageReference Include="Microsoft.Extensions.Logging" Version="6.0.0" />
        <PackageReference Include="Microsoft.Extensions.Logging.Console" Version="6.0.0"
        />
        <PackageReference  Include="Microsoft.Extensions.Logging.Debug"  Version="6.0.0"
        />
 </ItemGroup>

</Project>
```

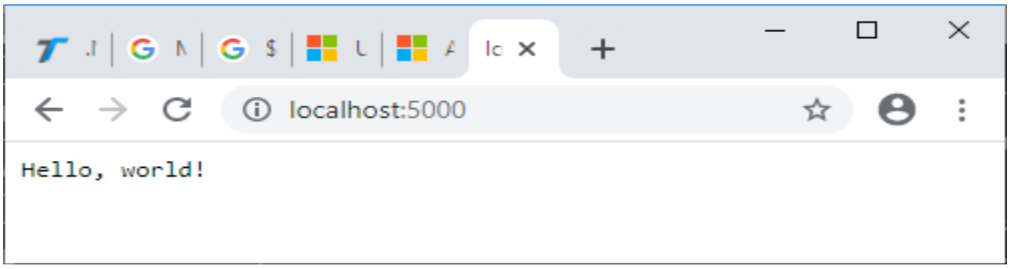
5. **Go to the command prompt and execute the following commands.**

    1. dotnet restore
    2. dotnet build
    3. dotnet run

    dotnet restore is used to restore or download dependencies for the .NET application. dotnet build is used to compile the application. dotnet run is used to executing the application. Resolve any errors if any shows the command prompt after the execution of dotnet run command. This indicates the server has started executing.



6. **Go to the browser and type localhost:5000 in the address bar and hit enter to view the output as shown**



Practical 3

Building an ASP.NET MVC Core rest API

Solution:

Create MVC application type:

$ dotnet new mvc --auth none

We can simply indicate that we want to use the Web SDK (Microsoft.NET. Sdk.Web) at the opening of the project file, and that saves us from having to explicitly declare certain dependencies: we initially get a *Program.cs* file that contains the following code after we issue a dotnet new console command:

```
public class Program
{
public static void Main(string[] args)
{
Console.WriteLine("Hello World!");
}
}
```

We then modified the *Program.cs* file to add configuration support as well as enable the Kestrel web server, as shown here:

```
public static void Main(string[] args)
{
var config = new ConfigurationBuilder().AddCommandLine(args).Build();

var host = new WebHostBuilder().UseContentRoot(Directory.GetCurrentDirectory())
.UseKestrel().UseStartup<Startup>() .UseConfiguration(config).Build();
host.Run();
}
```

Note the use of the UseContentRoot method. We have to do this so that when the application starts it can find all of the supporting files, like the *.cshtml* files for views. Next we added a Startup class that configures the default middleware that responds with "Hello, world!" to all HTTP requests:

```
public class Startup
{
public Startup(IHostingEnvironment env)
{
}
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
        {
        app.Run(async (context) =>{await context.Response.WriteAsync("Hello, world!"); });
        }
}
```

We also added the following NuGet packages as dependencies for our project:

- Microsoft.AspNetCore

The basic building blocks for all ASP.NET applications.

- Microsoft.AspNetCore.Server.Kestrel
- The Kestrel web server.
- Microsoft.Extensions.Configuration.CommandLine

Extensions for parsing command-line parameters. This will be required to change the port number on which our application runs via command-line argument. At this point, we technically have a functioning ASP.NET web application, but it is really just simple middleware that does nothing of value. While we've already got plenty of experience with controller routing for our microservices, we're going to finally delve into the "M" and "V" aspects of MVC: the *model* and *view*.

With the simplified syntax of the project file, we can simply indicate that we want to use the Web SDK (Microsoft.NET.Sdk.Web) at the opening of the project file, and that saves us from having to explicitly declare certain dependencies:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>
<TargetFramework>netcoreapp1.1</TargetFramework>
</PropertyGroup>

<ItemGroup>
<PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
<PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
<PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
<PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1" />
<PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="1.1.0" />
<PackageReference Include="Microsoft.Extensions.Configuration" Version="1.1.1"/>
<PackageReference
Include="Microsoft.Extensions.Options.ConfigurationExtensions" Version="1.1.1"/>
<PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="1.1.1"/>
<PackageReference Include="Microsoft.Extensions.Configuration.CommandLine"
Version="1.1.1"/>
</ItemGroup>

</Project>
```

Adding ASP.NET MVC Middleware

Let's enhance our existing sample by adding support for the MVC framework with the default routing scheme that we're familiar with. To do this, we simply replace the app.Use middleware configuration with the UseMvc extension in the Startup class, as shown in our new class.

Startup.cs

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
namespace StatlerWaldorfCorp.WebApp
{
public class Startup
{
public Startup(IHostingEnvironment env)
{
}

public void ConfigureServices(IServiceCollection services)
{
        services.AddMvc();
}

public void Configure(IApplicationBuilder app,
IHostingEnvironment env, ILoggerFactory loggerFactory)
{
app.UseMvc(routes=>{routes.MapRoute("default",template:"{controller=Home}/{action=Inde
x}/{id?}");});
}

}
}
```

For this to work, we'll also need to add a dependency on the NuGet package

- Microsoft.AspNetCore.Mvc.

The default route that we added should look familiar to you if you've done any ASP.NET MVC development in the past. Go ahead and run this application with the usual command-line tools (dotnet restore, dotnet run) and see what happens. You should simply get 404s on every possible route because we have no controllers.

Adding a Controller

Controllers should do the following and *nothing more*:
1. Accept input from HTTP requests.
2. Delegate the input to service classes that are written without regard for HTTP transport or JSON parsing.
3. Return an appropriate response code and body.

In other words, our controllers should be very, very small. They should do little more than wrap highly tested components that can operate outside the context of a web request if necessary. To add a controller to our project, let's create a new folder called *Controllers* and put a class in it called HomeController

HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace StatlerWaldorfCorp.Controllers
{
public class HomeController : Controller
{
public string Index()
{
return "Hello World";
}
}
}
```

With the simple addition of this file, the route we created earlier will automatically pick up the existence of this controller and let us use it. If you run the app from the command line and hit the home URL (e.g.,

http://localhost:5000 or whatever port you're running on) you'll see the text "Hello World" in your browser.

Adding a Model

The role of the model is, as you might have guessed, to represent the data required by the controller and the view to present some form of interaction between the user and the application. This isn't a book on building ASP.NET MVC web applications (there are far more detailed references available), so we won't go into all of the things that you can do with models, like automatic validation and so on.

StockQuote.cs

```
namespace StatlerWaldorfCorp.WebApp.Models
{
public class StockQuote
{
public string Symbol { get; set; }
public int Price { get; set; }
}
}
```

Adding a View

Now that we've got a controller and a model, let's build a view to render that data to the user via server-side templating. Just like with the controller and the model, there is a default convention for locating the views that correspond to controllers. For example, if we wanted to create a view for the HomeController's Index method, we would store that view as *Index.cshtml* in the *Views/Home* directory.

Views/Home/Index.cshtml

```
<html>
<head>
<title>Hello world</title>
</head>
<body>
<h1>Hello World</h1>
<div>
<h2>Stock Quote</h2>
<div>
Symbol: @Model.Symbol<br/>
Price: $@Model.Price<br/>
</div>
</div>
</body>
</html>
```

Now we can modify our home controller to render a view instead of returning simple text:

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using webapp.Models;
namespace webapp.Controllers

{
public class HomeController : Controller
{
public async Task<IActionResult> Index()
{
var model = new StockQuote { Symbol = "HLLO", Price = 3200 };
return View(model);
}
}
}
```

UseDeveloperExceptionPage method to our Startup class, in the Configure method. Here is our new and complete Startup class:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
namespace StatlerWaldorfCorp.WebApp
{
public class Startup
{
public Startup(IHostingEnvironment env)
{
var builder = new
ConfigurationBuilder().SetBasePath(env.ContentRootPath).AddEnvironmentVariables();
Configuration = builder.Build();
}
```

```
public IConfiguration Configuration { get; set; }

public void ConfigureServices(IServiceCollection services)
{
services.AddMvc();
}

public void Configure(IApplicationBuilder app,IHostingEnvironment env, ILoggerFactory
loggerFactory)
{
loggerFactory.AddConsole();
loggerFactory.AddDebug();
app.UseDeveloperExceptionPage();
app.UseMvc(routes
=>{routes.MapRoute("default",template:"{controller=Home}/{action=Index}/{id?}");});
app.UseStaticFiles();
}
}
}
```

Practical 4

Install Docker Desktop on Windows, verify installation create your first repository.

Solution:

1. To install Docker on Windows OS Docker Desktop software can be used. Docker Desktop is the Community version of Docker for Microsoft Windows. Docker Desktop for Windows can be downloaded from Docker Hub and installed or follow the instructions on the website

https://docs.docker.com/docker-for-windows/install/ for complete installation process.

2. System requirements

Your Windows machine must meet the following requirements to successfully install Docker Desktop.

WSL 2 backend and Hyper-V

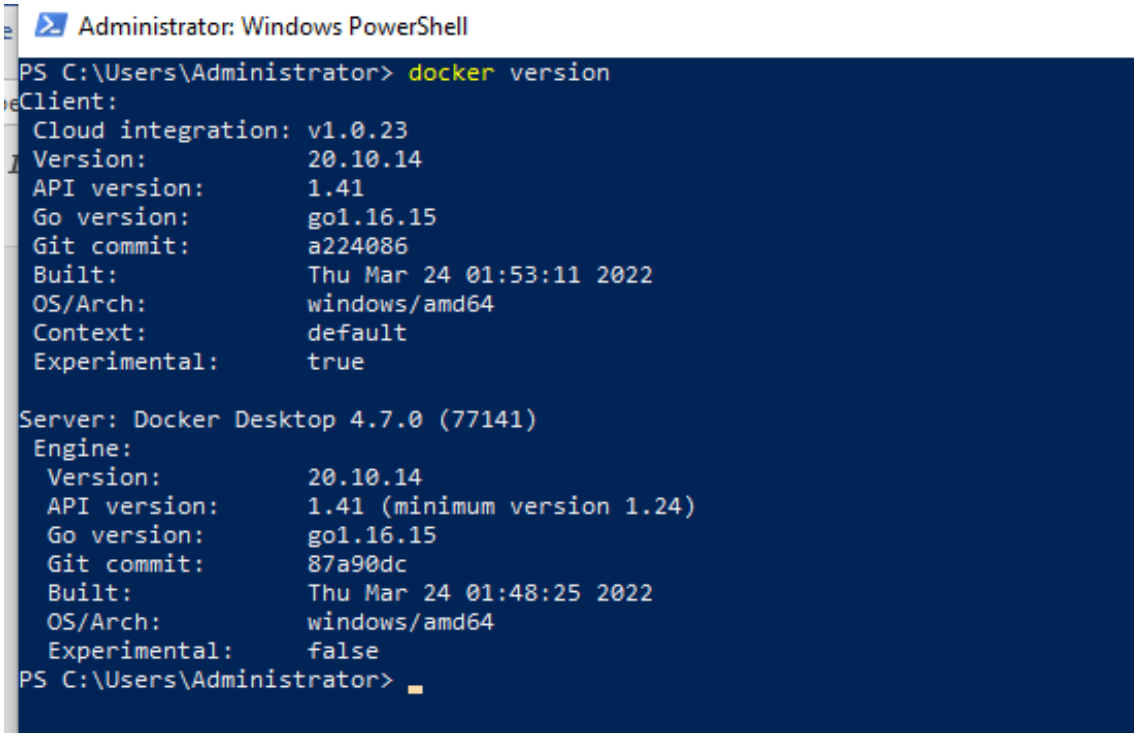
- Windows 11 64-bit: Home or Pro version 21H2 or higher, or Enterprise or Education version 21H2 or higher.
- Windows 10 64-bit: Home or Pro 2004 (build 19041) or higher, or Enterprise or Education 1909 (build 18363) or higher.
- Enable the WSL 2 feature on Windows. For detailed instructions, refer to the Microsoft documentation.
- Hyper-V and Containers Windows features must be enabled.
- The following hardware prerequisites are required to successfully run WSL 2 and Client Hyper-V on Windows 10 or Windows 11:
    - 64-bit processor with Second Level Address Translation (SLAT)
    - 4GB system RAM
    - BIOS-level hardware virtualization support must be enabled in the BIOS settings.

Note: Docker only supports Docker Desktop on Windows for those versions of Windows 10

1. Double-click **Docker Desktop Installer.exe** to run the installer.
   If you haven't already downloaded the installer (Docker Desktop Installer.exe), you can get it from **Docker Hub**. It typically downloads to your ownloads folder, or you can run it from the recent downloads bar at the bottom of your web browser.
2. When prompted, ensure the **Enable Hyper-V Windows Features** or the **Install required Windows components for WSL 2** option is selected on the Configuration page.
3. Follow the instructions on the installation wizard to authorize the installer and proceed with the install.
4. When the installation is successful, click **Close** to complete the installation process.
5. If your admin account is different to your user account, you must add the user to the **docker-users** group. Run **Computer Management** as an **administrator** and navigate to **Local Users and Groups** > **Groups** > **docker-users**. Right-click to add the user to the group. Log out and log back in for the changes to take effect.

3. Testing Your Docker Install

Run docker version to check the basic details of your deployment. You should see "Windows" listed as the operating system for the Docker client and the Docker Engine:



Docker Hub is a service provided by Docker for finding and sharing container images with your team. It is the world's largest repository of container images with an array of content sources including container community developers, open source projects and independent software vendors (ISV) building and distributing their code in containers.

Step 1: Sign up for a Docker account

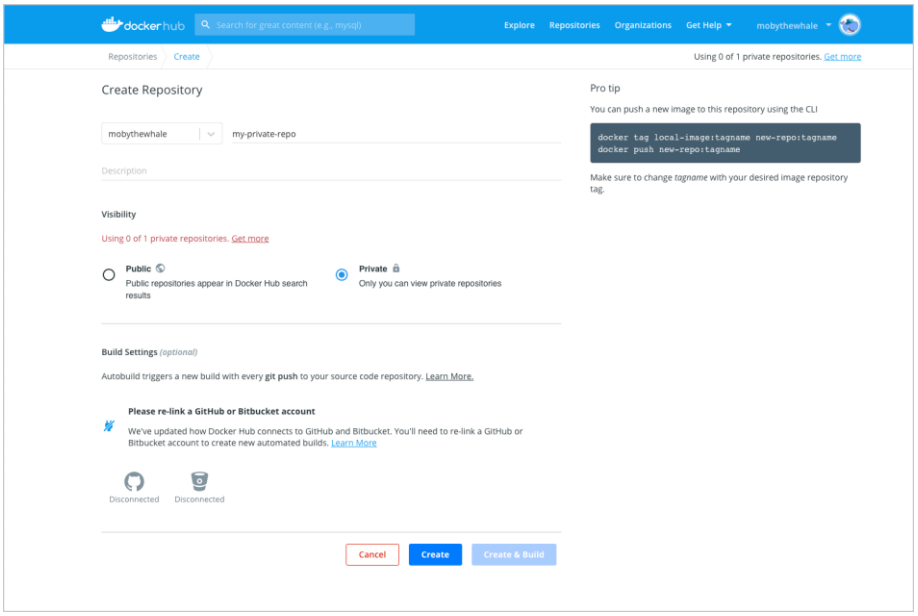
Let's start by creating a Docker ID.

A Docker ID grants you access to Docker Hub repositories and allows you to explore images that are available from the community and verified publishers. You'll also need a Docker ID to share images on Docker Hub.

Step 2: Create your first repository

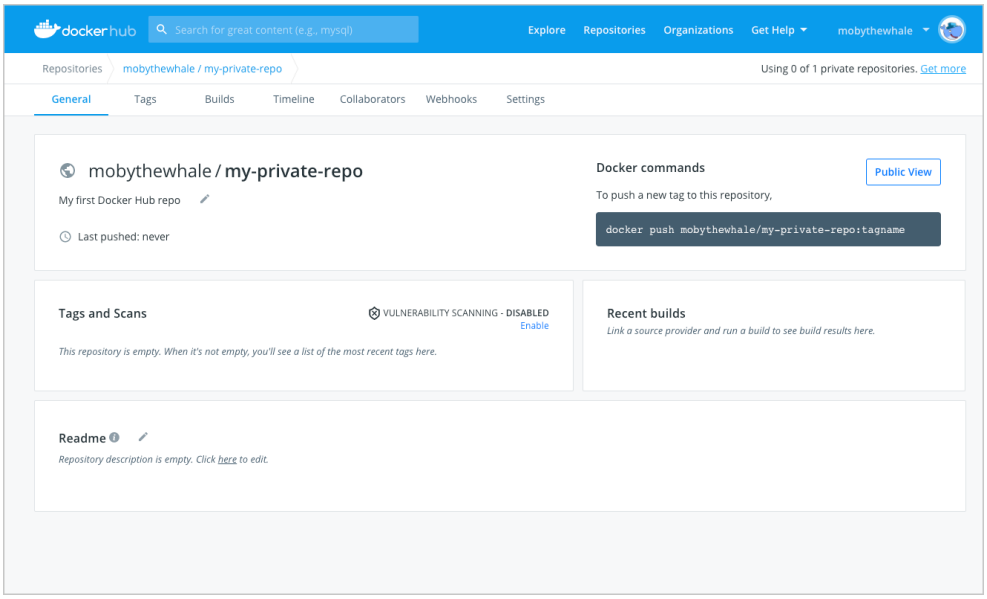
To create a repository:

1. Sign in to Docker Hub.
2. Click **Create a Repository** on the Docker Hub welcome page.
3. Name it **<your-username>/my-private-repo**.
4. Set the visibility to **Private**.



1. Click **Create**.

   You've created your first repository. You should see:



Practical 5

Built a Docker container image on your computer and pushed it to Docker Hub.

Step 1: Download and install Docker Desktop
We'll need to download Docker Desktop to build and push a container image to Docker Hub.

1. Download and install Docker Desktop.
2. Sign in to the Docker Desktop application using the Docker ID you created in Step 1.

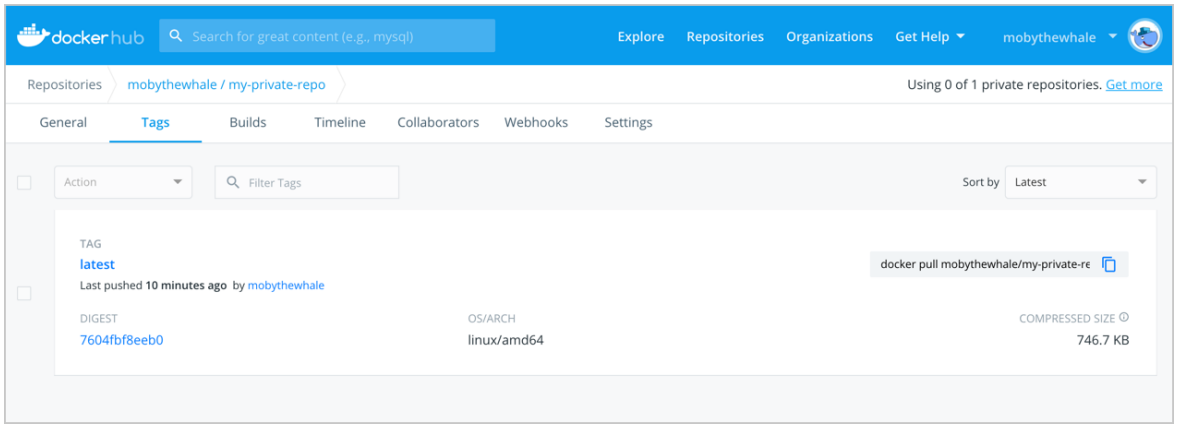
Step 2: Build and push a container image to Docker Hub from your computer

1. Start by creating a Dockerfile to specify your application as shown below:
2. # syntax=docker/dockerfile:1
3. FROM busybox
4. CMD echo "Hello world! This is my first Docker image."
5. Run docker build -t <your_username>/my-private-repo . to build your Docker image.
6. Run docker run <your_username>/my-private-repo to test your Docker image locally.
7. Run docker push <your_username>/my-private-repo to push your Docker image to Docker Hub. You should see output similar to:

```
[redacted] cat > Dockerfile <<EOF
FROM busybox
CMD echo "Hello world! This is my first Docker image."
EOF
[redacted] docker build -t mobythewhale/my-private-repo .
[+] Building 1.2s (5/5) FINISHED
 => [internal] load build definition from Dockerfile                    0.0s
 => => transferring dockerfile: 110B                                    0.0s
 => [internal] load .dockerignore                                       0.0s
 => => transferring context: 2B                                         0.0s
 => [internal] load metadata for docker.io/library/busybox:latest       1.2s
 => CACHED [1/1] FROM docker.io/library/busybox@sha256:a9286defaba7b3a519  0.0s
 => exporting to image                                                  0.0s
 => => exporting layers                                                 0.0s
 => => writing image sha256:dcdb1fd928bfb257bfc0122ea47accd911a3a386ce618  0.0s
 => => naming to docker.io/mobythewhale/my-private-repo                 0.0s
[redacted] docker run mobythewhale/my-private-repo
Hello world! This is my first Docker image.
[redacted] docker push mobythewhale/my-private-repo
The push refers to repository [docker.io/mobythewhale/my-private-repo]
d2421964bad1: Layer already exists
latest: digest: sha256:7604fbf8eeb03d866fd005fa95cdbb802274bf9fa51f7dafba6658294
efa9baa size: 526
[redacted]
```

Having trouble pushing? Remember, you must be signed into Docker Hub through Docker Desktop or the command line, and you must also name your images correctly, as per the above steps.

8. Your repository in Docker Hub should now display a new latest tag under **Tags**:



Congratulations! You've successfully:

- Signed up for a Docker account
- Created your first repository
- Built a Docker container image on your computer
- Pushed it successfully to Docker Hub

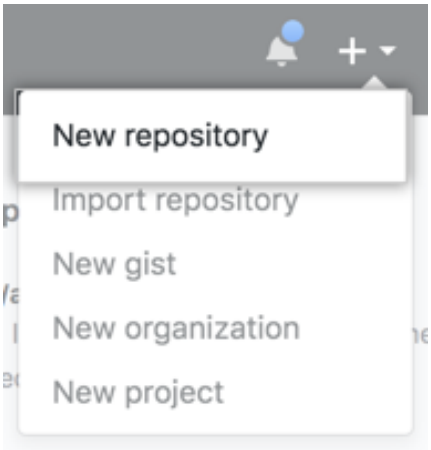
Practical 6

Create repository and branches on Github

Solution:

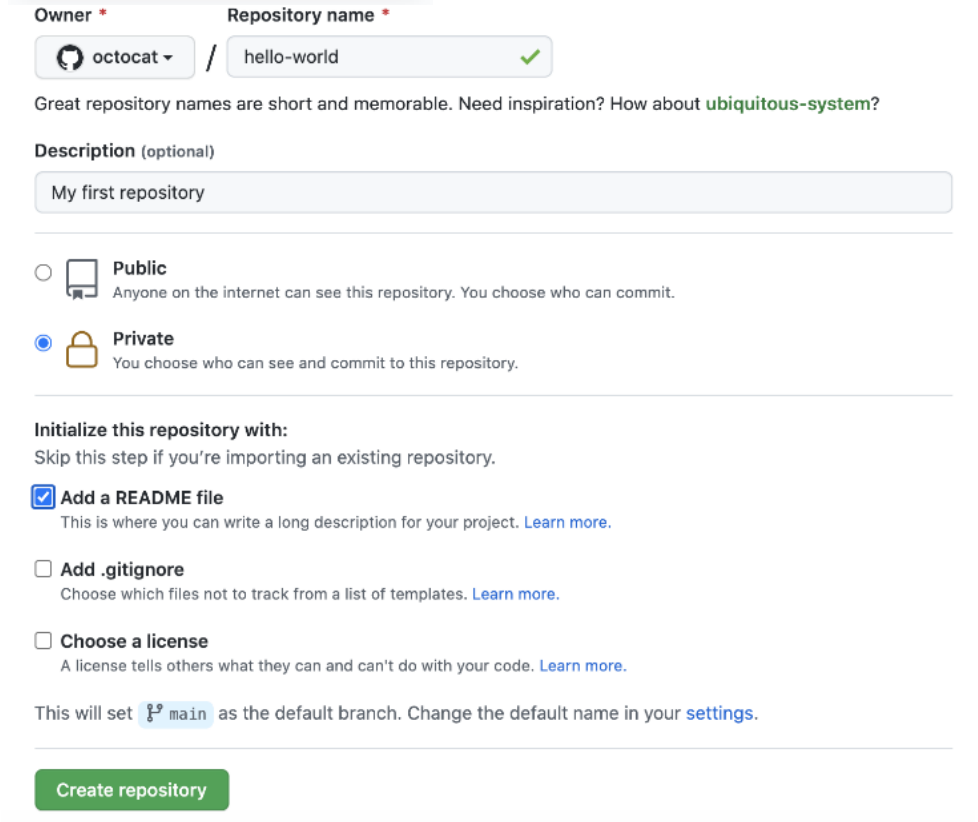
GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere. You need a GitHub account and Internet access.

Creating a repository



A repository is usually used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets -- anything your project needs. Often, repositories include a *README* file, a file with information about your project. *README* files are written in the plain text Markdown language.

Your hello-world repository can be a place where you store ideas, resources, or even share and discuss things with others. 1. In the upper-right corner of any page, use the drop-down menu, and select **new repository**.



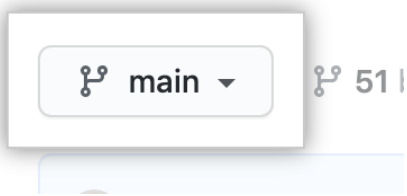
In the **Repository name** box, enter hello-world.

2. In the **Description** box, write a short description.
3. Select **Add a README file**.
4. Select whether your repository will be **Public** or **Private**.
5. Click **Create repository**.

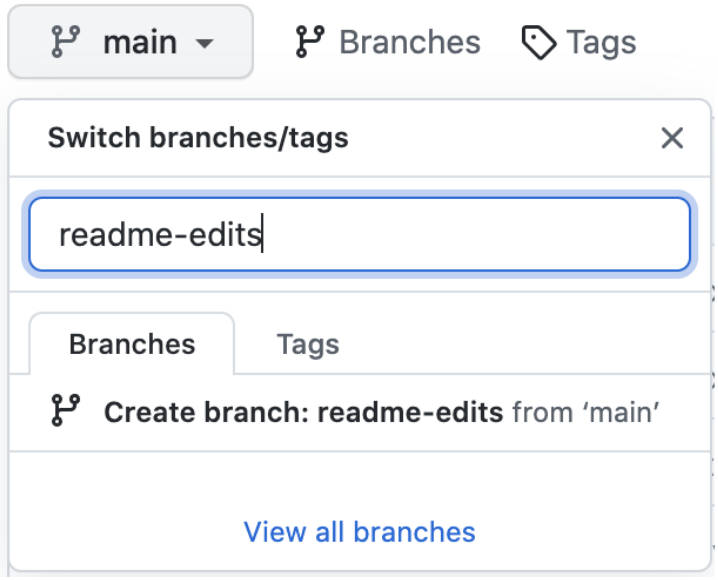
Creating a branch

Branching lets you have different versions of a repository at one time. By default, your repository has one branch named main that is considered to be the definitive branch. You can create additional branches off of main in your repository. You can use branches to have different versions of a project at one time. This is helpful when you want to add new features to a project without changing the main source of code.

1. Click the **Code** tab of your hello-world repository.



2. Click the drop down at the top of the file list that says **main**.
3. Type a branch name, readme-edits, into the text box.
4. Click **Create branch: readme-edits from main**.



Now you have two branches, main and readme-edits. Right now, they look exactly the same. Next you'll add changes to the new branch.

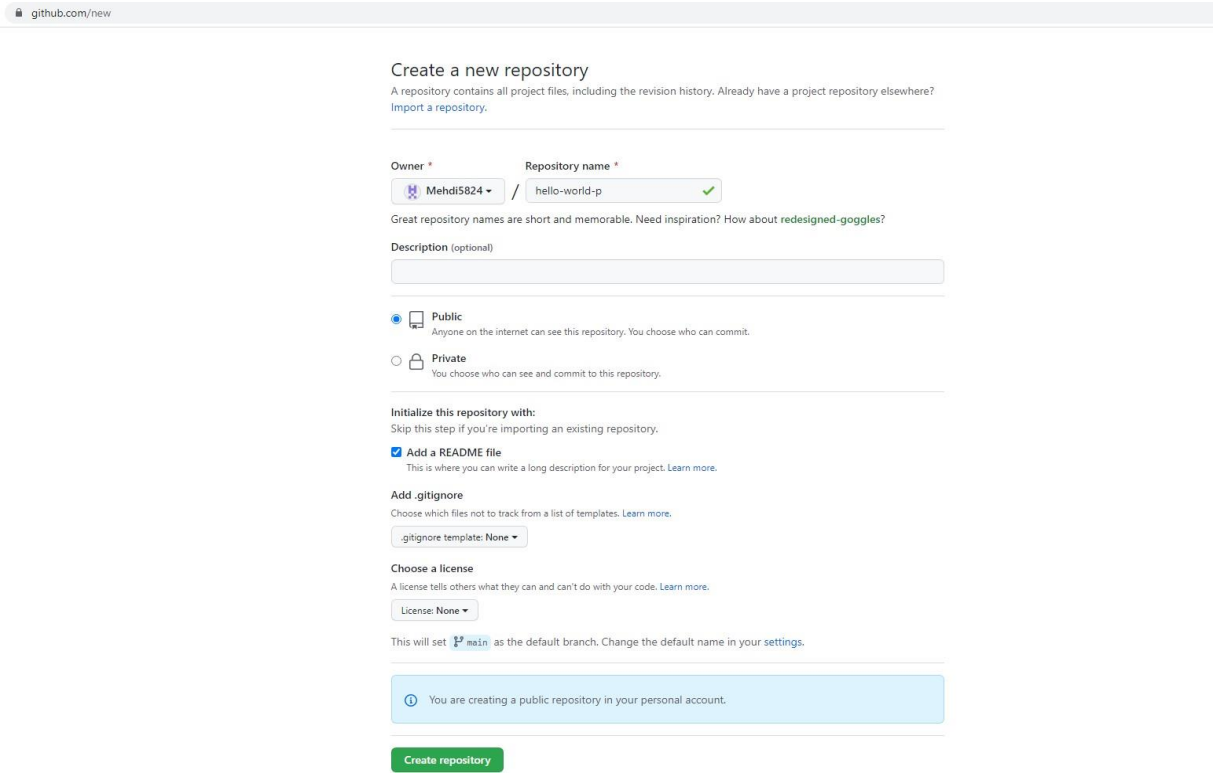
Practical 7

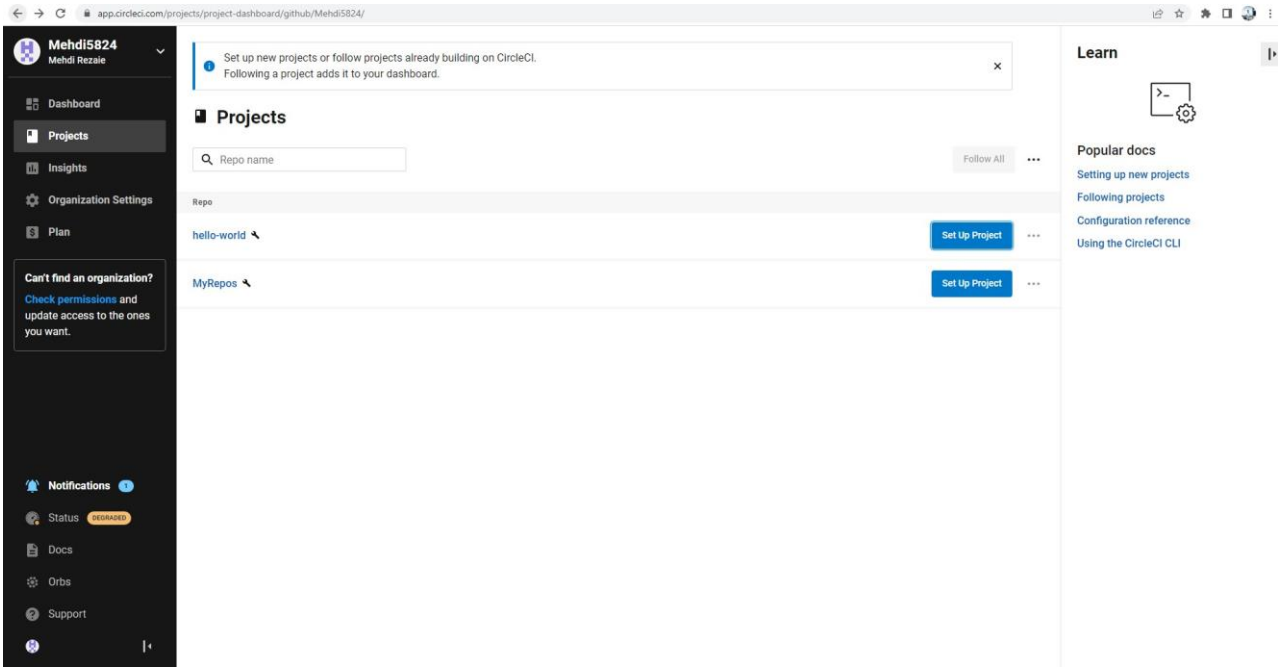
Working with Circle CI for continuous integration

Step 1 - Create a repository

1. Log in to GitHub and begin the process to create a new repository.
2. Enter a name for your repository (for example, hello-world).
3. Select the option to initialize the repository with a README file.
4. Finally, click Create repository.
5. There is no need to add any source code for now.

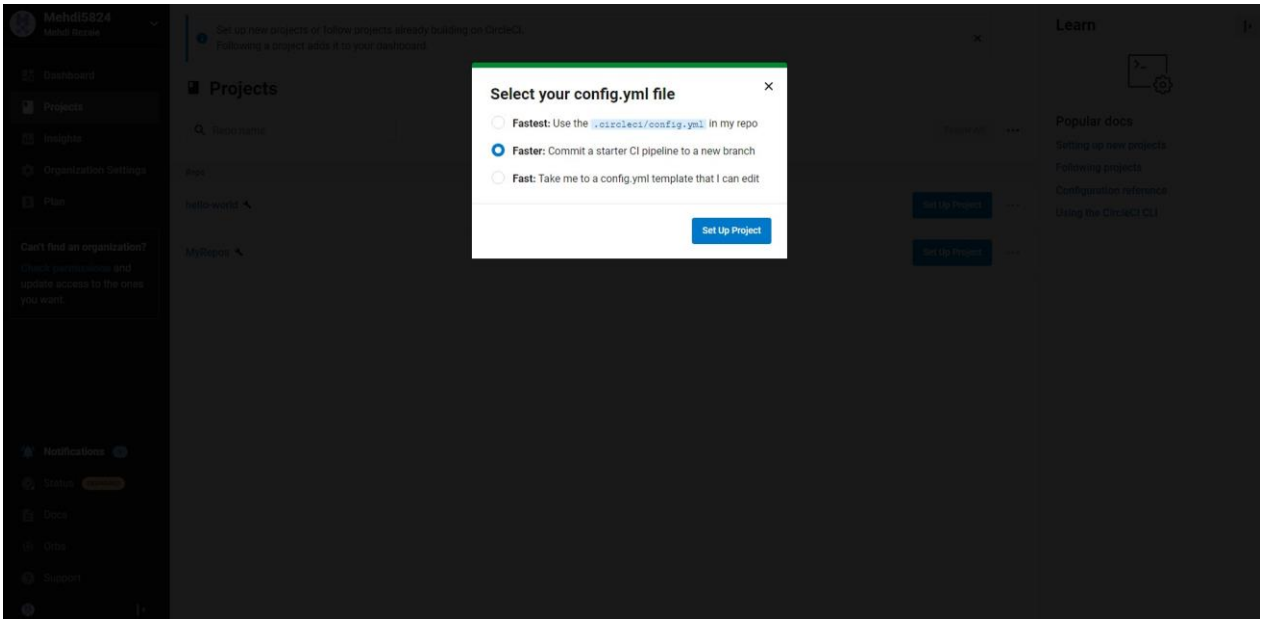
Login to Circle CI https://app.circleci.com/ Using GitHub Login, Once logged in navigate to Projects.





Step 2 - Set up CircleCI

1. Navigate to the CircleCI Projects page. If you created your new repository under an organization, you will need toselect the organization name.
2. You will be taken to the Projects dashboard. On the dashboard, select the project you want to set up (hello-world).
3. Select the option to commit a starter CI pipeline to a new branch, and click Set Up Project. This will create a file .circleci/config.yml at the root of your repository on a new branch called circleci-project-setup.



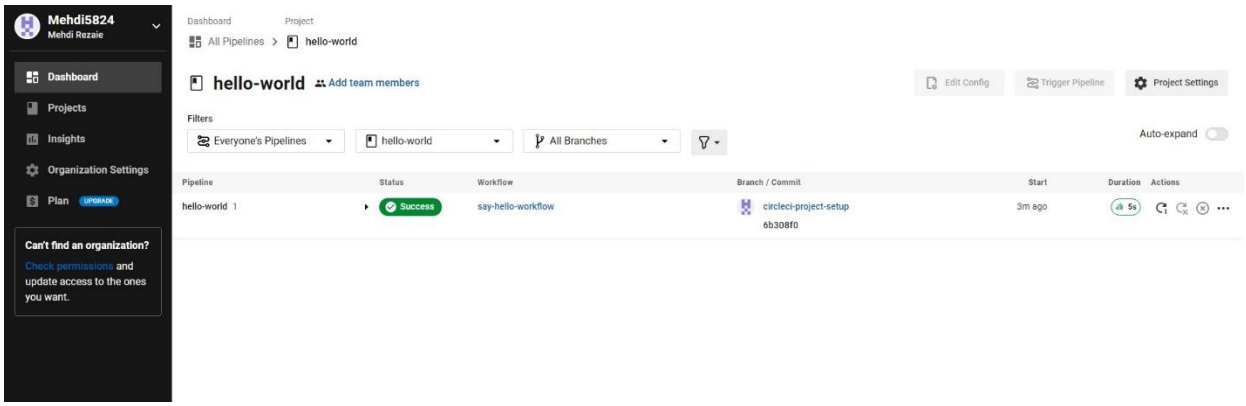
Step 3 - Your first pipeline

On your project's pipeline page, click the green Success button, which brings you to the workflow that ran (say-hello-workflow).

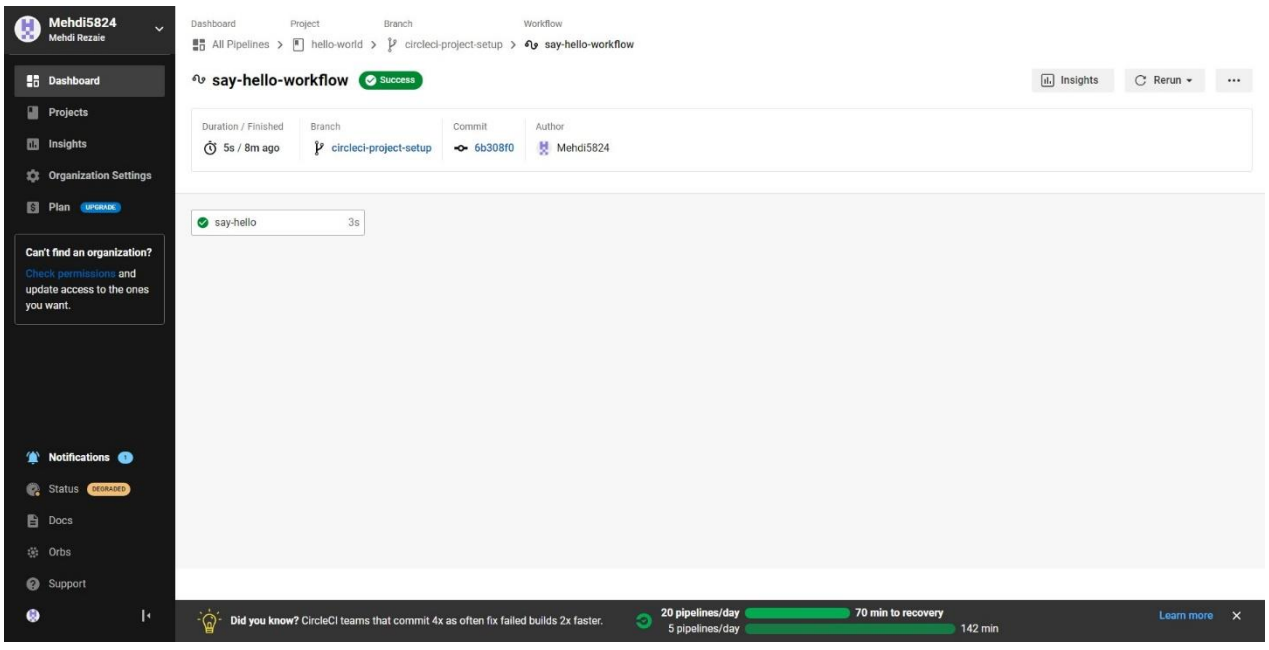
Within this workflow, the pipeline ran one job, called say-hello. Click say-hello to see the steps in this job:

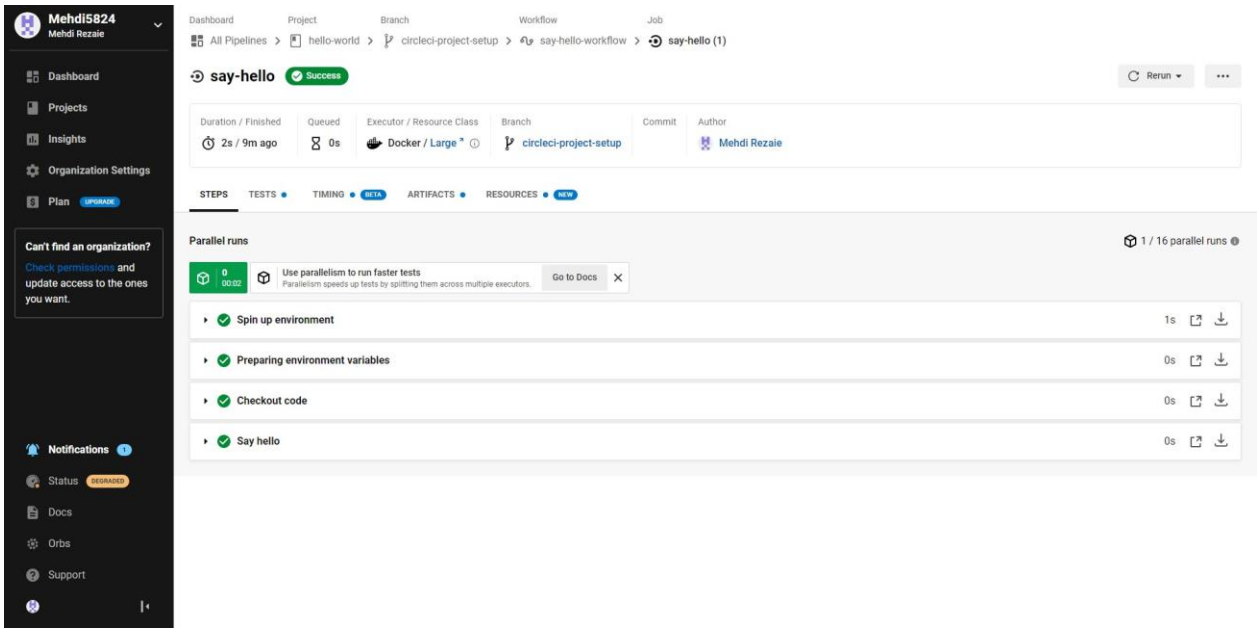
a. Spin up environment
b. Preparing environment variables
c. Checkout code
d. Say hello

Now select the "say-hello-workflow" to the right of Success status column

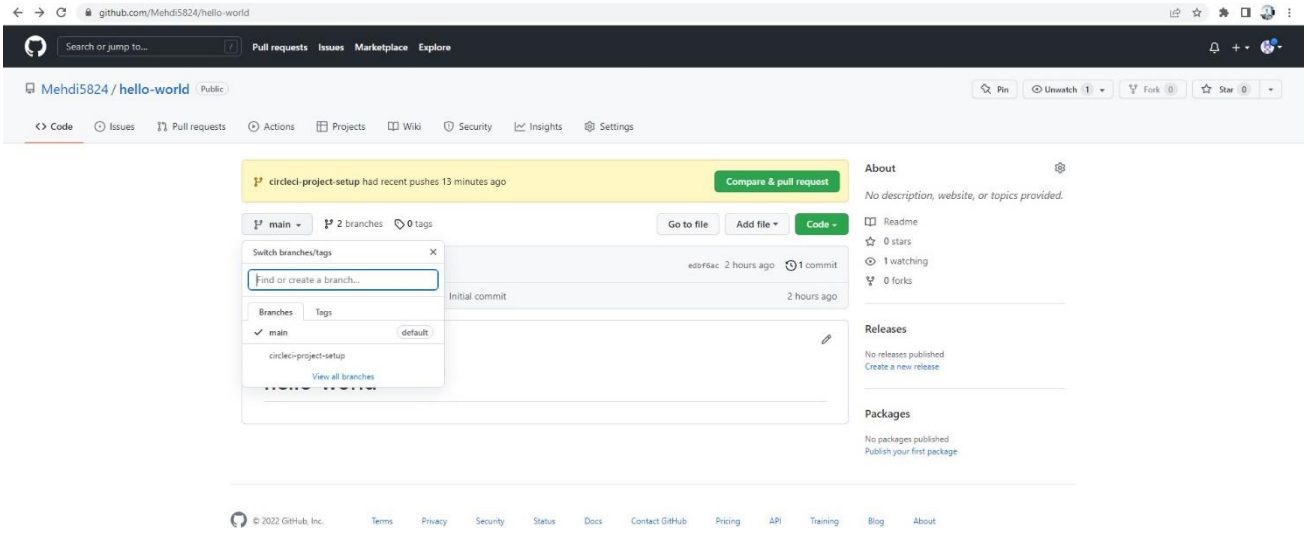


Select "say-hello" Job with a green tick



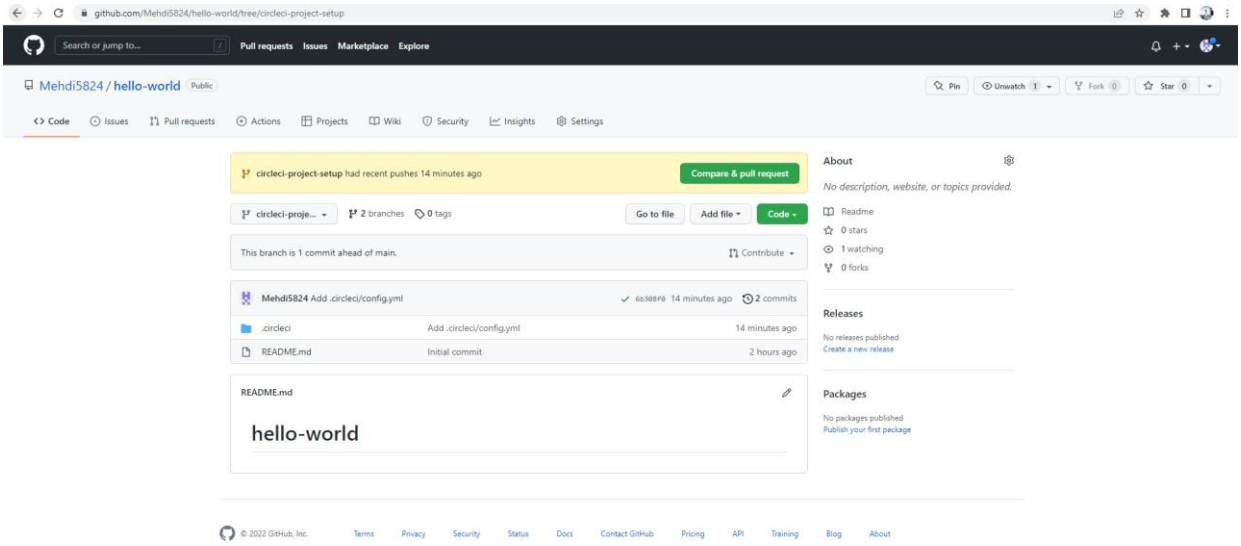


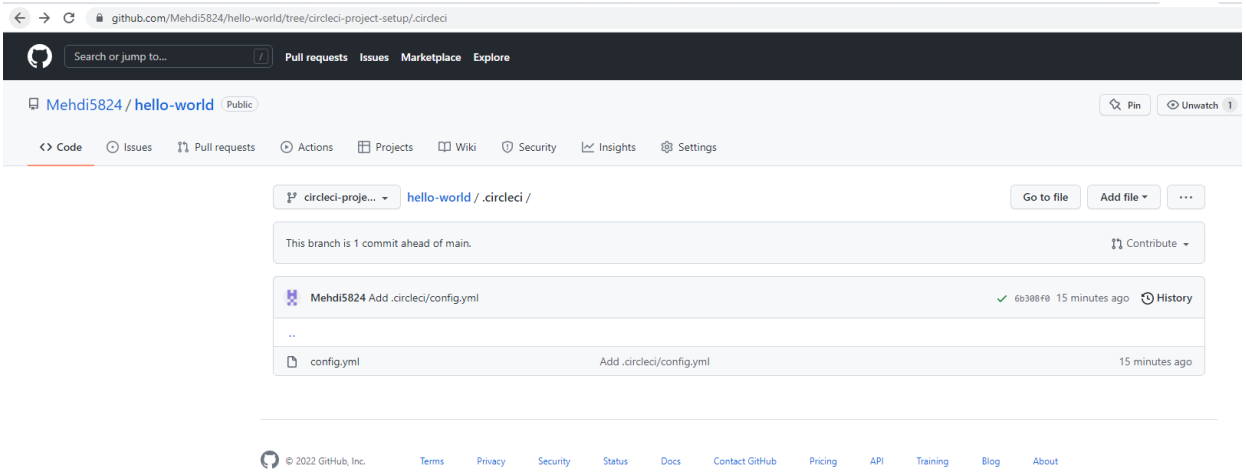
Select Branch and option circleci-project-setup

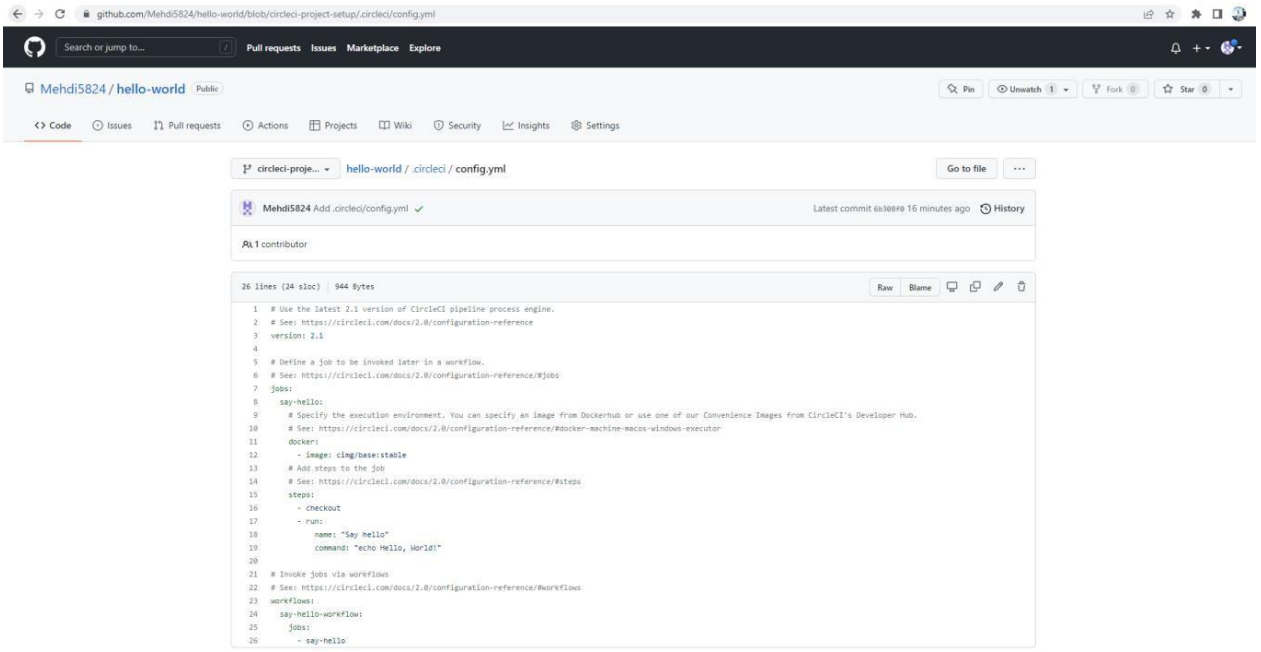


Step 4 - Break your build

In this section, you will edit the .circleci/config.yml file and see what happens if a build does not complete successfully. It is possible to edit files directly on GitHub.


github.com/Mehdi5824/hello-world/tree/circleci-project-setup
Search or jump to...
Pull requests Issues Marketplace Explore
Mehdi5824 / hello-world Public
Pin Unwatch 1 Fork 0 Star 0
Code Issues Pull requests Actions Projects Wiki Security Insights Settings
circleci-project-setup had recent pushes 14 minutes ago
Compare & pull request
circleci-proje... 2 branches 0 tags
Go to file Add file Code
This branch is 1 commit ahead of main.
Contribute
Mehdi5824 Add .circleci/config.yml
6b308f0 14 minutes ago 2 commits
.circleci Add .circleci/config.yml 14 minutes ago
README.md Initial commit 2 hours ago
README.md
hello-world
About
No description, website, or topics provided.
Readme
0 stars
1 watching
0 forks
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
© 2022 GitHub, Inc.
Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About

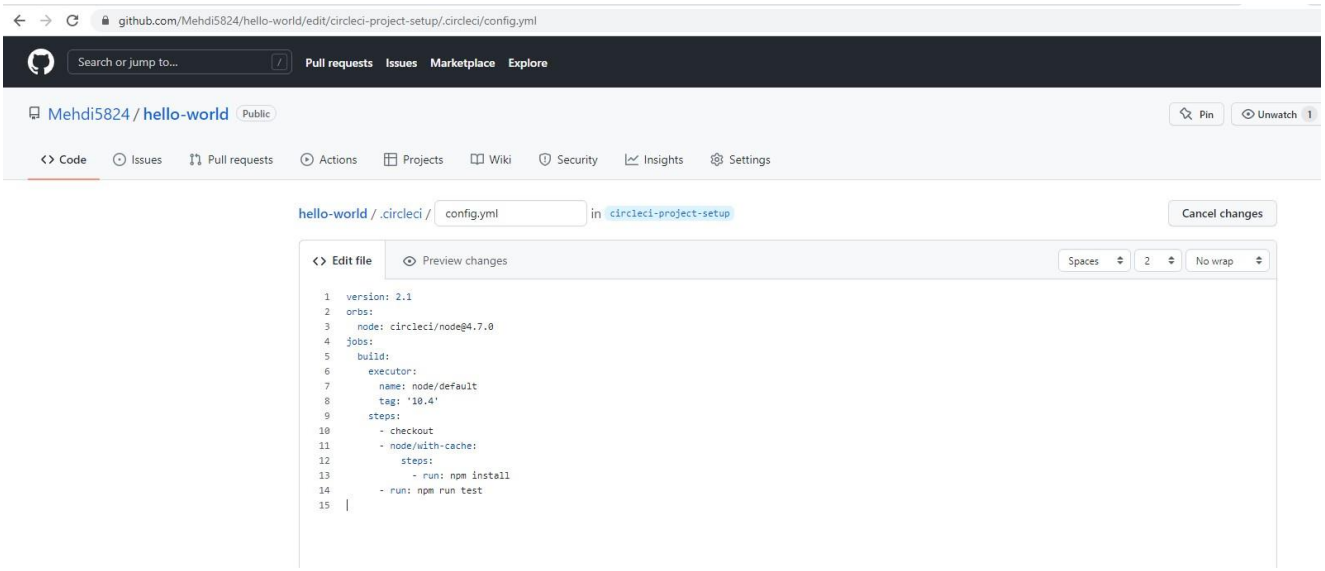

github.com/Mehdi5824/hello-world/tree/circleci-project-setup/.circleci
Search or jump to...
Pull requests Issues Marketplace Explore
Mehdi5824 / hello-world Public
Pin Unwatch 1
Code Issues Pull requests Actions Projects Wiki Security Insights Settings
circleci-proje... hello-world / .circleci /
Go to file Add file
This branch is 1 commit ahead of main.
Contribute
Mehdi5824 Add .circleci/config.yml
6b308f0 15 minutes ago History
..
config.yml Add .circleci/config.yml 15 minutes ago
© 2022 GitHub, Inc.
Terms Privacy Security Status Docs Contact GitHub Pricing API Training Blog About


github.com/Mehdi5824/hello-world/blob/circleci-project-setup/.circleci/config.yml
Search or jump to...
Pull requests Issues Marketplace Explore
Mehdi5824 / hello-world Public
Code Issues Pull requests Actions Projects Wiki Security Insights Settings
circleci-proje... hello-world / .circleci / config.yml
Go to file
Mehdi5824 Add .circleci/config.yml
Latest commit 6b308f0 16 minutes ago History
1 contributor
26 lines (24 sloc) 944 Bytes
Raw Blame
Use the latest 2.1 version of CircleCI pipeline process engine.
See: https://circleci.com/docs/2.0/configuration-reference
version: 2.1
Define a job to be invoked later in a workflow.
See: https://circleci.com/docs/2.0/configuration-reference/#jobs
jobs:
say-hello:
Specify the execution environment. You can specify an image from Dockerhub or use one of our Convenience Images from CircleCI's Developer Hub.
See: https://circleci.com/docs/2.0/configuration-reference/#docker-machine-macos-windows-executor
docker:
- image: cimg/base:stable
Add steps to the job
See: https://circleci.com/docs/2.0/configuration-reference/#steps
steps:
- checkout
- run:
name: "Say hello"
command: "echo Hello, World!"
Invoke jobs via workflows
See: https://circleci.com/docs/2.0/configuration-reference/#workflows
workflows:
say-hello-workflow:
jobs:
- say-hello

Let's use the Node orb. Replace the existing config by pasting the following code:

```
version: 2.1
orbs:
  node: circleci/node@4.7.0
jobs:
  build:
    executor:
      name: node/default
      tag: '10.4'
    steps:
      - checkout
      - node/with-cache:
          steps:
            - run: npm install
      - run: npm run test
```

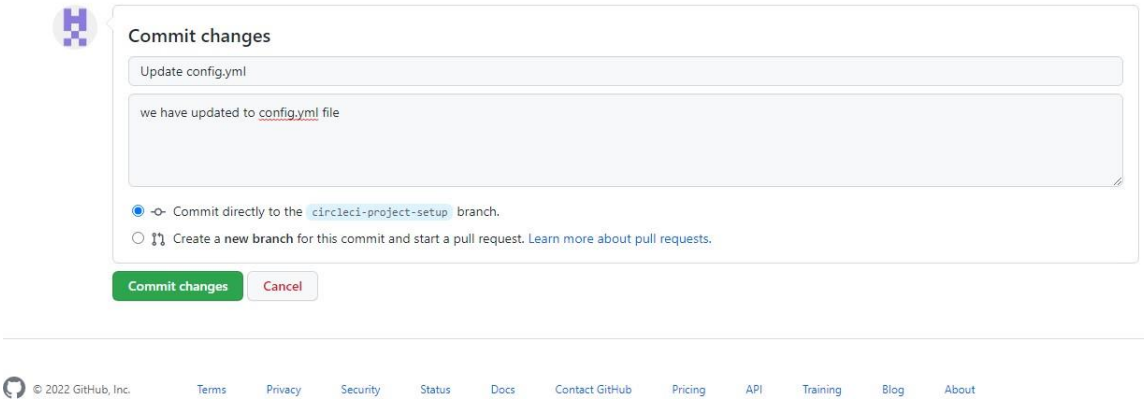
The GitHub file editor should look like this



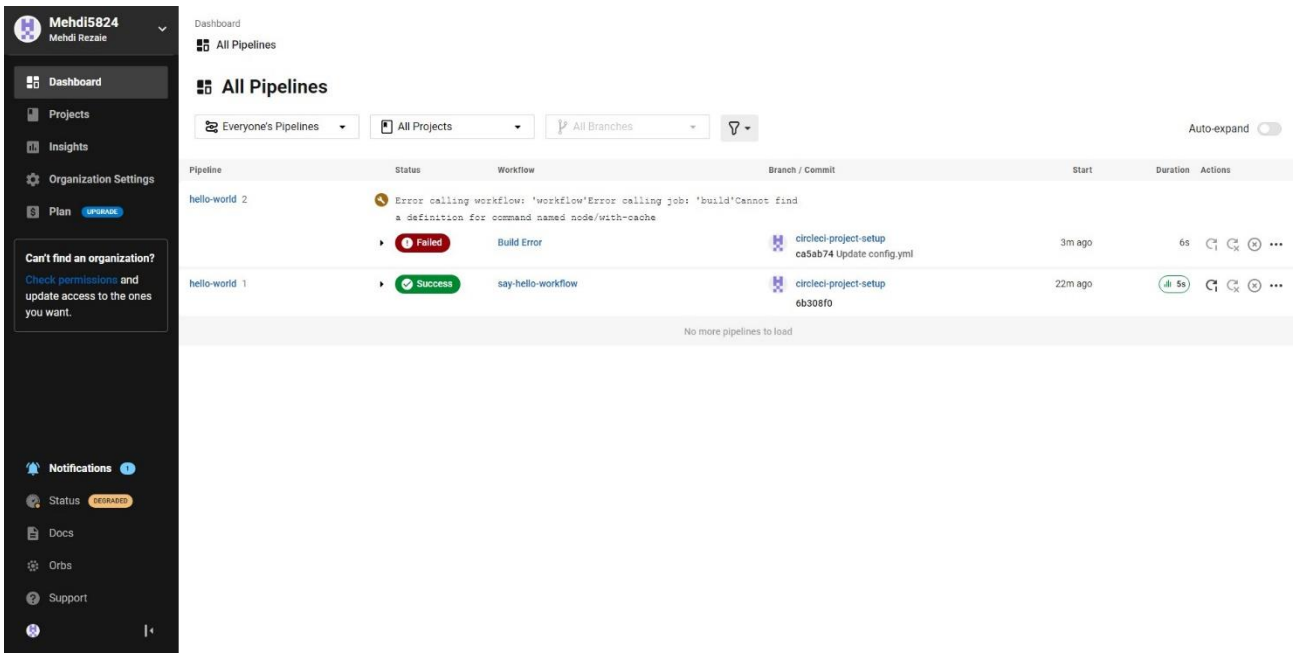
Scroll down and Commit your changes on GitHub

```
      - checkout
      - node/with-cache:
          steps:
            - run: npm install
      - run: npm run test

```



After committing your changes, then return to the Projects page in CircleCI. You should see a new pipeline running… and itwill fail! What's going on? The Node orb runs some common Node tasks. Because you are working with an empty repository, running npm run test, a Node script, causes the configuration to fail. To fix this, you need to set up a Node project in your repository.



Step 5 – Use Workflows

You do not have to use orbs to use CircleCI. The following example details how to create a custom configuration that alsouses the workflow feature of CircleCI.

1) Take a moment and read the comments in the code block below. Then, to see workflows in action, edityour **.circleci/config.yml file** and copy and paste the following text into it.

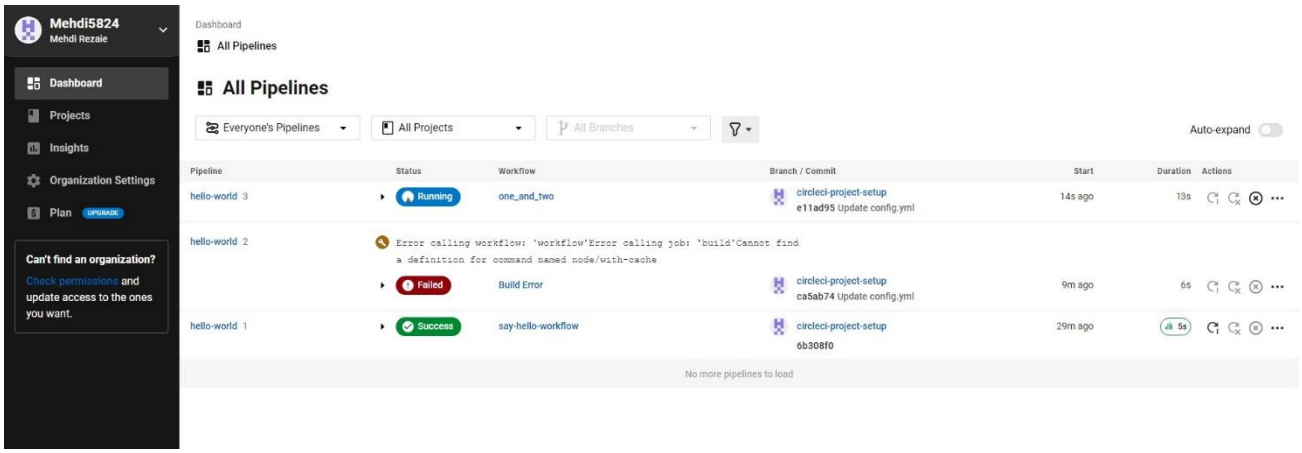
You don't need to write the comments which are the text after #

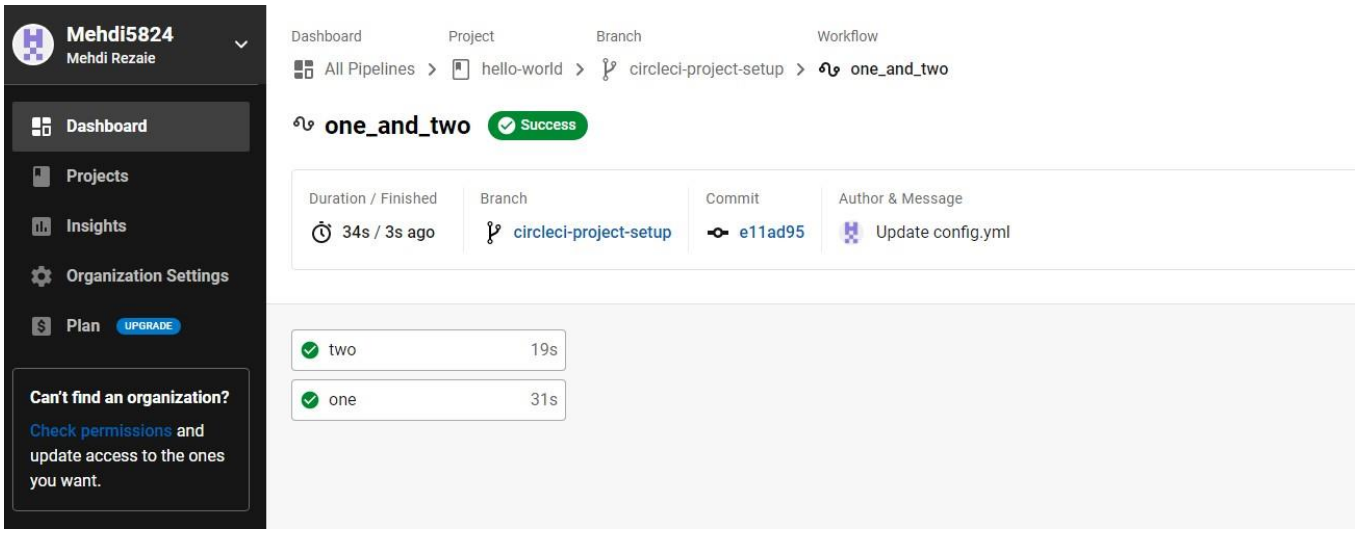
```
version: 2
jobs: # we now have TWO jobs, so that a workflow can coordinate them!
  one: # This is our first job.
    docker: # it uses the docker executor
      - image: cimg/ruby:2.6.8 # specifically, a docker image with ruby 2.6.8
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
    # Steps are a list of commands to run inside the docker container above.
    steps:
      - checkout # this pulls code down from GitHub
      - run: echo "A first hello" # This prints "A first hello" to stdout.
      - run: sleep 25 # a command telling the job to "sleep" for 25 seconds.
  two: # This is our second job.
    docker: # it runs inside a docker image, the same as above.
      - image: cimg/ruby:3.0.2
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
    steps:
      - checkout
      - run: echo "A more familiar hi" # We run a similar echo command to above.
      - run: sleep 15 # and then sleep for 15 seconds.
# Under the workflows: map, we can coordinate our two jobs, defined above.
workflows:
  version: 2
  one_and_two: # this is the name of our workflow
    jobs: # and here we list the jobs we are going to run.
      - one
      - two
```

2) Commit these changes to your repository and navigate back to the CircleCI Pipelines page. You should see your pipelinerunning.

```
version: 2
jobs: # we now have TWO jobs, so that a workflow can coordinate them!
  one: # This is our first job.
    docker: # it uses the docker executor
      - image: cimg/ruby:2.6.8 # specifically, a docker image with ruby 2.6.8
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
    # Steps are a list of commands to run inside the docker container above.
    steps:
      - checkout # this pulls code down from GitHub
      - run: echo "A first hello" # This prints "A first hello" to stdout.
      - run: sleep 25 # a command telling the job to "sleep" for 25 seconds.
  two: # This is our second job.
    docker: # it runs inside a docker image, the same as above.
      - image: cimg/ruby:3.0.2
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
    steps:
      - checkout
      - run: echo "A more familiar hi" # We run a similar echo command to above.
      - run: sleep 15 # and then sleep for 15 seconds.
# Under the workflows: map, we can coordinate our two jobs, defined above.
workflows:
  version: 2
  one_and_two: # this is the name of our workflow
    jobs: # and here we list the jobs we are going to run.
      - one
      - two
```

3) Click on the running pipeline to view the workflow you have created. You should see that two jobs ran (or are currentlyrunning!) concurrently.





Step 5 – Add some changes to use workspaces

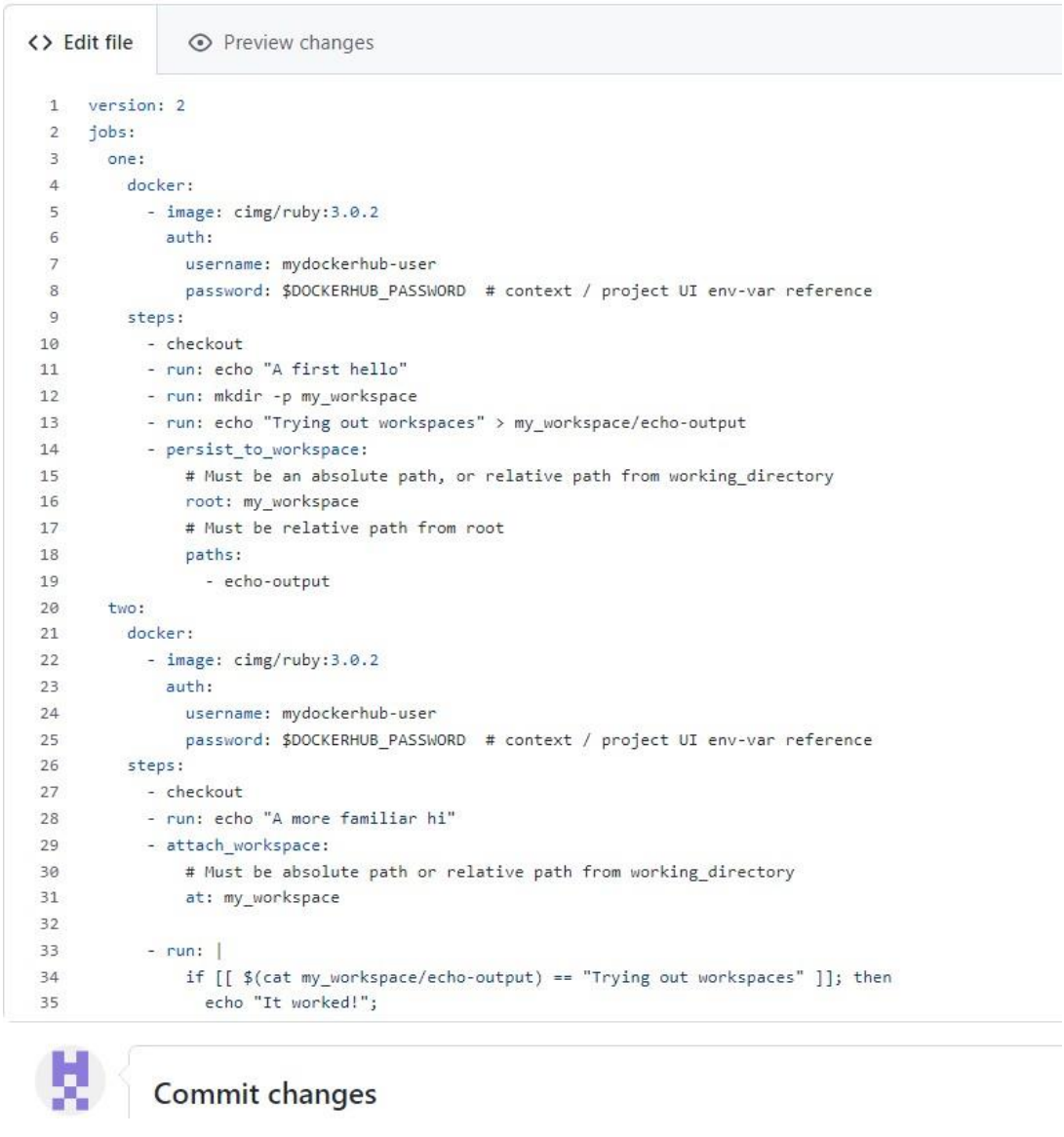
Each workflow has an associated workspace which can be used to transfer files to downstream jobs as the workflow progresses. You can use workspaces to pass along data that is unique to this run and which is needed for downstream jobs. Try updating config.yml to the following:

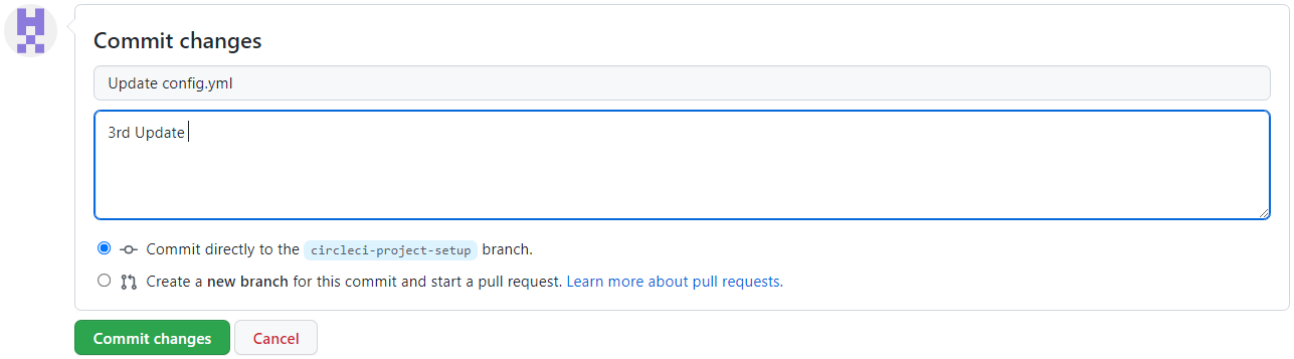
```
version: 2
jobs:
  one:
    docker:
      - image: cimg/ruby:3.0.2
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
    steps:
      - checkout
      - run: echo "A first hello"
      - run: mkdir -p my_workspace
      - run: echo "Trying out workspaces" > my_workspace/echo-output
      - persist_to_workspace:
          # Must be an absolute path, or relative path from working_directory
          root: my_workspace
          # Must be relative path from root
          paths:
            - echo-output
  two:
    docker:
      - image: cimg/ruby:3.0.2
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
    steps:
      - checkout
      - run: echo "A more familiar hi"
      - attach_workspace:
          # Must be absolute path or relative path from working_directory
          at: my_workspace
```

```
32
33        - run: |
34            if [[ $(cat my_workspace/echo-output) == "Trying out workspaces" ]]; then
35              echo "It worked!";
36            else
37              echo "Nope!"; exit 1
38            fi
39  workflows:
40    version: 2
41    one_and_two:
42      jobs:
43        - one
44        - two:
45            requires:
46              - one
```

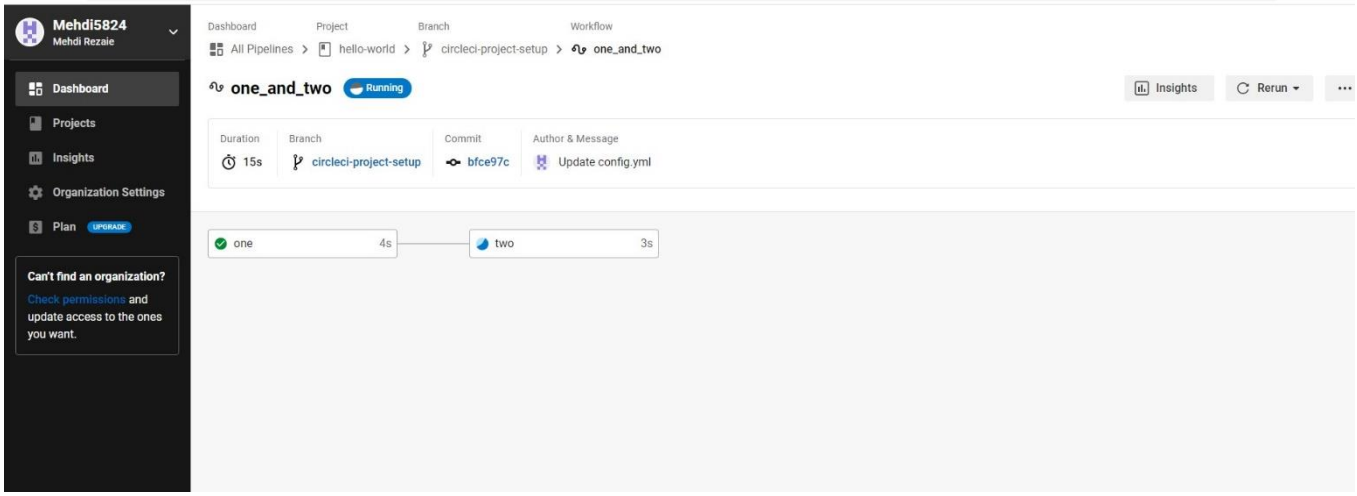
Updated config.yml in GitHub file editor should be updated like this

hello-world / .circleci / config.yml in circleci-project-setup





Finally your workflow with the jobs running should look like this



Practical 8

Working with Kubernetes

Kubernetes is a container-based platform for managing cloud resources and developing scalable apps. It is widely regarded as the most common platform for automating, deploying, and scaling the entire cloud infrastructure. The platform runs on all major operating systems and is the most widely used open-source cloud tool.

Prerequisites

For installing Kubernetes in your system, here are a few prerequisites that need special attention. The hardware and software requirements are discussed below:

Hardware requirements

- Master node with at least 2 GB memory. (Additional will be great)
- Worker node with 700 MB memory capacity.
- Your Mouse/Keyboard (monitor navigation)

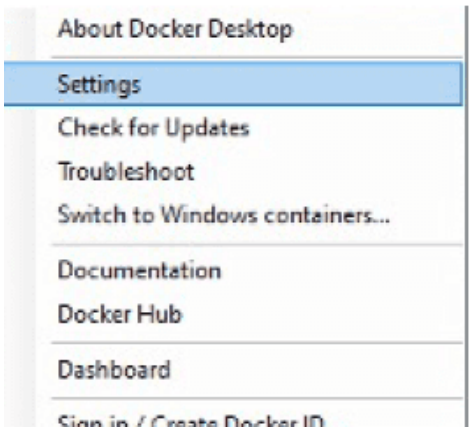
Software requirements

- Hype-V
- Docker Desktop
- Unique MAC address
- Unique product UUID for every node

Docker includes a graphical user interface (GUI) tool that allows you to change some settings or install and enable Kubernetes.

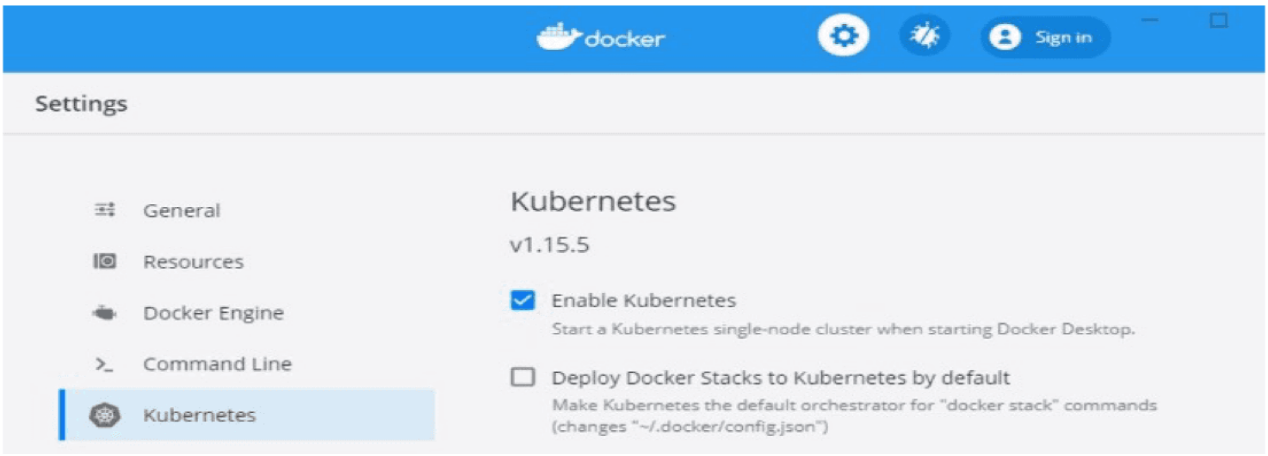
To install Kubernetes, simply follow the on-screen instructions on the screen:

1. Right-click the Docker tray icon and select Properties.
2. Select "Settings" from the drop-down menu.



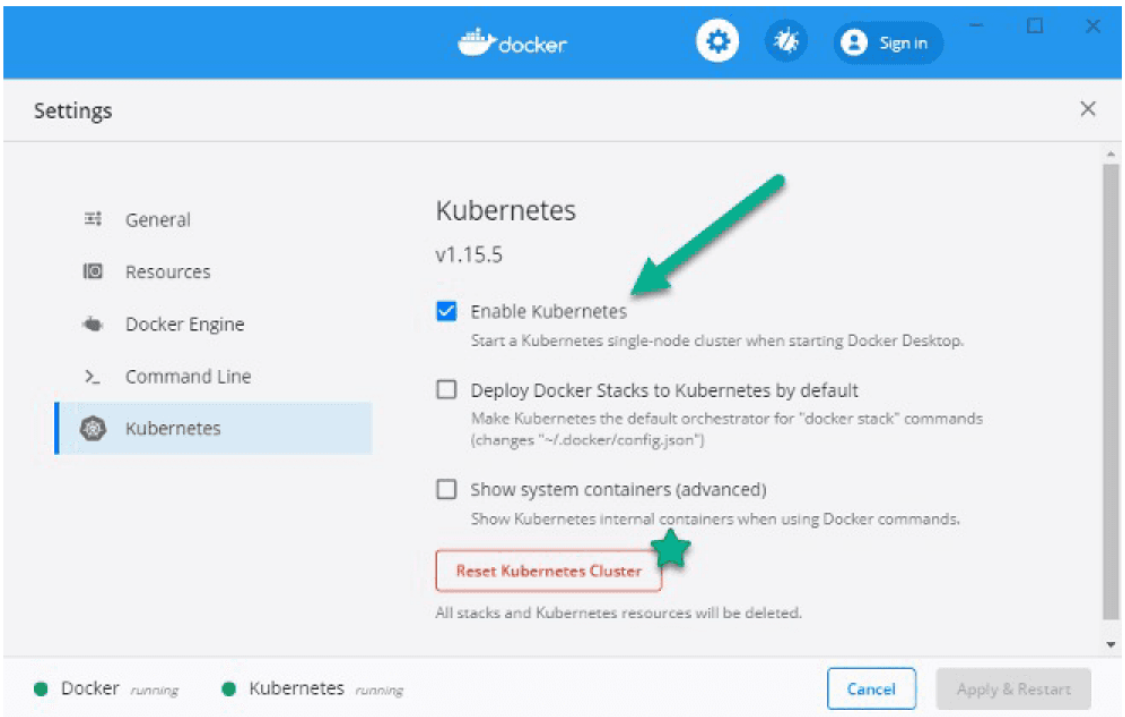
3. Select "Kubernetes" from the left panel.

4. Check Enable Kubernetes and click "Apply"



Docker will install additional packages and dependencies during the installation process. It may take between 5 and 10 minutes to install, depending on your Internet speed and PC performance. Wait until the message 'Installation complete!' appears on the screen. The Docker app can be used after Kubernetes has been installed to ensure that everything is working properly. Both icons at the bottom left will turn green if both services (Docker and Kubernetes) are running successfully and without errors.

Example.



Step 4: Install Kubernetes Dashboard

The official web-based UI for managing Kubernetes resources is Kubernetes Dashboard. It isn't set up by default. Kubernetes applications can be easily deployed using the cli tool kubectl, which allows you to interact with your cloud and manage your Pods, Nodes, and Clusters. You can easily create or update Kubernetes resources by passing the apply argument followed by your YAML configuration file.

Use the following commands to deploy and enable the Kubernetes Dashboard.

1. Get the yaml configuration file from here.
2. Use this to deploy it

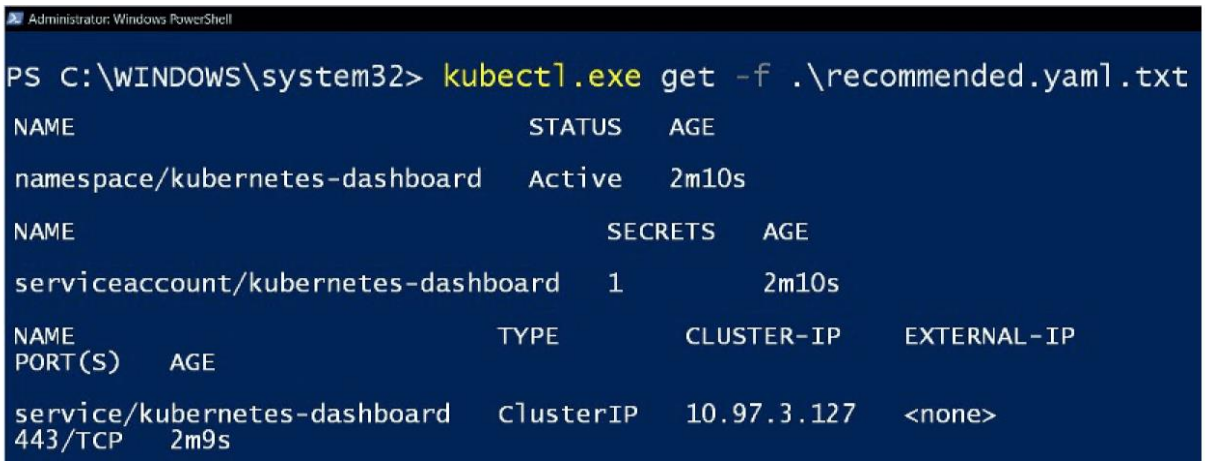
. kubectl apply -f .\recommended.yaml

Copy Code

3. Run the following command to see if it's up and running.:

kubectl.exe get -f .\recommended.yaml.txt

Copy Code



Step 5: Access the dashboard

The dashboard can be accessed with tokens in two ways: the first is by using the default token created during Kubernetes installation, and the second (more secure) method is by creating users, giving them permissions, and then receiving the generated token. We'll go with the first option for the sake of simplicity.

1. Run the following command PowerShell (not cmd)

((kubectl -n kube-system describe secret default | Select-String "token:") -split " +")[1]

Copy Code

2. Copy the generated token
3. Run

kubectl proxy.

Copy Code

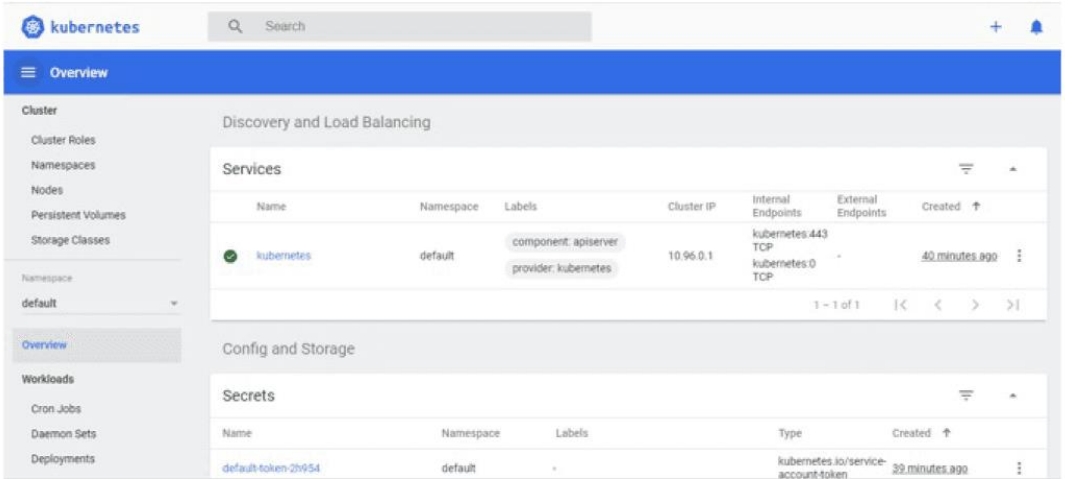
4. Open the following link on your browser:

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

Copy Code

5. Select
Token & paste the generated token
6. Sign In

Finally



You'll be able to see the dashboard and your cloud resources if everything is set up correctly. You can then do almost all of the "hard" work without having to deal with the CLI every time. You may occasionally get your hands dirty with the command line, but if you don't understand Docker and Kubernetes or don't have the time to manage your own cloud, it's better to stick with some PaaS providers that can be quite expensive.