Front end, back end, and project development environment setup

Setting up a development environment is essential in order to start building any application whether it is a web, desktop, or mobile based application. For doing so in a detailed and meticulous manner will be the foundation of starting or building any project which will in turn be deployed as an application to be used by everyone. So it is important that not only the developer can access and use the built application but so can everyone which means all the files in the developers system when moved or downloaded to and in other individuals devices must still serve the same functionality that it had done previously in the developer's device where it was built in the first place

Essentially because most applications have a front end or the UI (user interface) which the user sees, the back end which the allows the front end to request data from the server, and the database which stores the data, each of these have their diverse set of respective frameworks depending on the one you will be desiring to use that need to be installed in your system as a developer

Front end setup

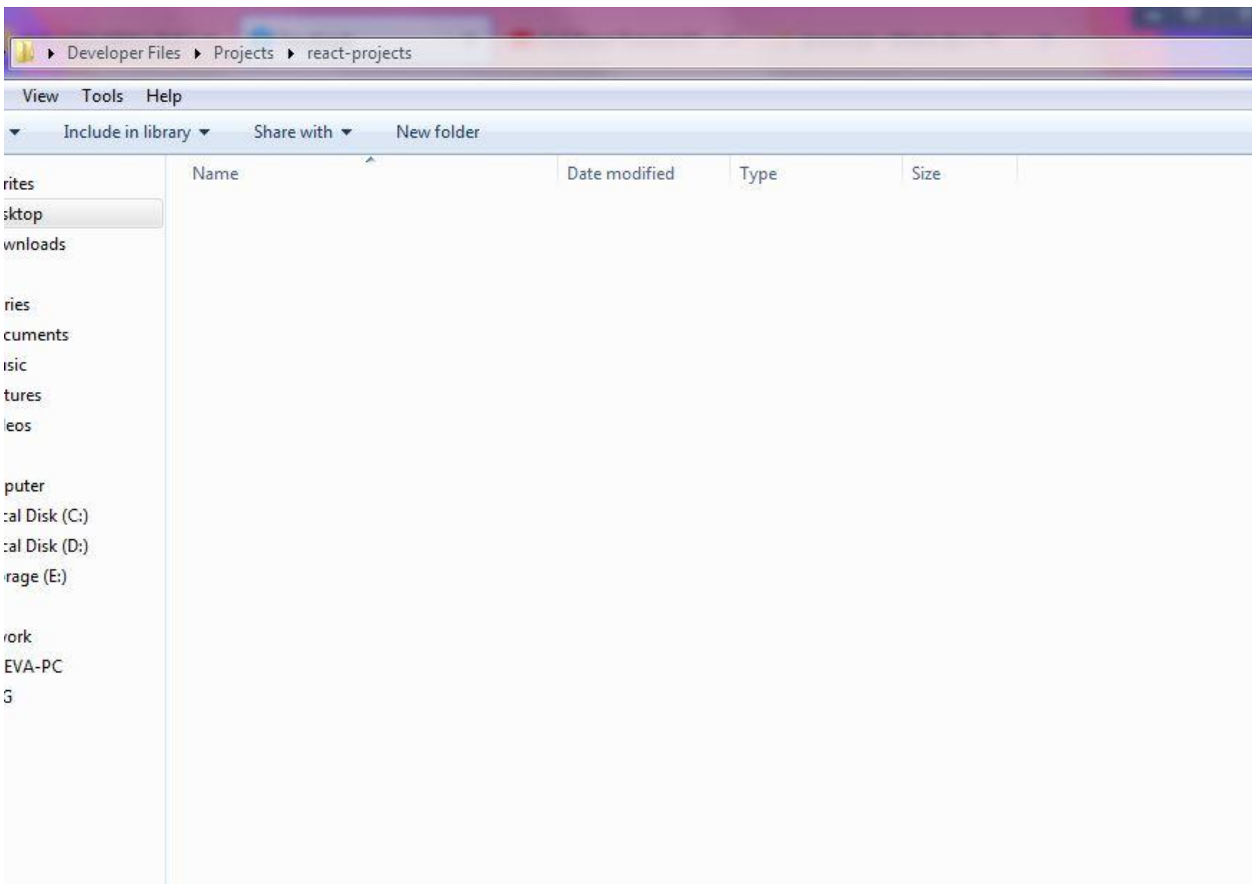
In order to first set up our front end development environment we need to add the node.js path to our environment variables so that we are able to use the keywords `npm` and `node` in our command line to execute `npm` and `node` related commands to set up our front end development environment. The path to or location of node.js is usually located in `C:\Users\<User>\AppData\Roaming\npm` once the installer program of node.js is done installing node.js in our device/system. Now that node.js is installed and added to our path, we can now run all the `node` and `npm` related commands. One way of testing this is by typing the command `node –v` or `node --version` in our command line or terminal which will output the version of node.js we are using.

Note that `–v` and `--version` and any command that has a single hyphen and a single letter are only mere short hands to the original long hand versions of the command which use a double hyphen then the command name. We can also compile or rather interpret and run our JavaScript files with the command `node <nameoffile>.js`

Creating the package.json file

It is essential to know that when we create front end projects or applications and use source code made by other developers which we can download and install and use in our application we basically must make a file called `package.json` in order to keep track of all these source code or from now on modules that we use in our project, that when we do deploy our project/application everyone can use it and make their own changes to it without ever running to the error that the used packages/modules in the application are missing. Why the `package.json` is important is because it contains information of all the moduels we used inour project and of course when we deploy our project it doesn't make sense to also deploy every time along with our project, files that are too large, which `package.json` solves automatically because once this single file containing all the information of our used modules is deployed along with our application, the developer/user can use this single file to install all of the modules we used in our application in their own system/device so the developer/user can have no trouble using the application we deployed in their own system/device.

Note that when starting a project all the up and coming commands below need to be used in a specific folder where we will place our project and other project files, this means that in our command line we have to navigate to our desired location which is also our project folder. To create the `package.json` we run the command `npm init` in our command line with the location set to our desired location, which in this case is `C:\Users\<User>\Desktop\Developer Files\Projects\react-projects`


Developer Files ▸ Projects ▸ react-projects
View Tools Help
Include in library ▾ Share with ▾ New folder
Name Date modified Type Size
rites
sktop
wnloads
ries
cuments
isic
tures
leos
puter
:al Disk (C:)
:al Disk (D:)
rage (E:)
ork
EVA-PC
G

In our command line once we have navigated to our desired location where to create the `package.json` file and use the `npm init` command we will get the following questions:

```
C:\Users\Cueva\Desktop\Developer Files\Projects\react-projects>npm init

This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sensible defaults.


See `npm help json` for definitive documentation on these fields

and exactly what they do.


Use `npm install <pkg>` afterwards to install a package and

save it as a dependency in the package.json file.


Press ^C at any time to quit.

package name: (react-projects)

version: (1.0.0)

description:

entry point: (index.js)
```

```
test command:

git repository:

keywords:

license: (MIT)

About to write to C:\Users\Cueva\Desktop\Developer Files\Projects\react-projects


{

  "name": "react-projects",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "author": "Michael Cueva",

  "license": "MIT"

}


Is this OK? (yes)
```

This is what is asked by node when we use the specific command `npm init` before we create our `package.json` file. It asks what values to put in the package-name, version, description, entry point, test command, git repository, keywords, and license keys (since these are JavaScript object/dictionary keys), but we can skip all these questions by simply running an alternative command which is `npm init –y` or `npm init --yes` which creates the `package.json` file and uses now only the default values for each key. Should we change however these default values we can always use the command `npm config set init-<key> "<newdefaultvalue>"` which if we use for example the author-name, or main, or license for `<key>` then it assigns new default values for these keys in the `package.json` file. And once we assign these new default values we can then run `npm init –y` and we will see these new default values assigned to each key the object declared in our `package.json` file.

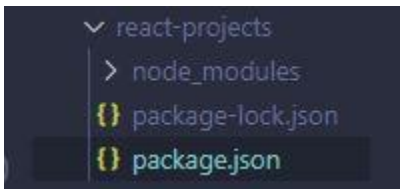
`npm config set init-author-name "<author name>"` - sets the default author name

`npm config set init-license "<e.g. MIT>"` – sets the default license

`npm config set init-main "<nameofindexjsfile>.js"` – sets the name of our main project file e.g. `"index.js"` or `"main.js"`

And finally like `npm config set init-<key>` we also have `npm config delete init-<key>` which deletes the default value of the author name in the `package.json` file

Moving on from the previous step we need to install now the necessary modules we will need in our project. There are two ways to install a package or module in our system using node, one is by global installation and the other by local installation. Whenever we use the command `npm –g install <packagename>` or `npm --global install <packagename>` node is essentially in this order creating a folder named `node_modules`, placed in, recall, our path that we used to add node to our environment variables which is located in …`/AppData/Roaming/npm`, and then installing and placing all the module source code in this `node_modules` folder. This is what is meant by global of a package, but for a mere local installation of our package to be used by our project placed locally in our system, we need to use the `npm install <packagename>` in order to install the `node_modules` folder along with our desired package to be installed. Note that in using this local way we must be in our project folder.


react-projects
node_modules
package-lock.json
package.json

However the mere command `npm install <packagename>` is not enough to let node know that our `package.json` uses and depends on this specific package in order to eventually use the project/application. Therefore we must use another command which is `npm install <packagename> --save` that will save essentially the module name that we installed and will be using in our project/application, as information in our `package.json` file. Using this command will specifically add a `dependencies` key in the `package.json` file object and populate it with all the modules/packages our project/application uses and will depend on in order to use such an application.

```
{
  "name": "react-projects",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Michael Cueva",
  "license": "MIT",
  "dependencies": {
    "lodash": "^4.17.21"
  }
}
```

It should be noted that should this `dependencies` key be removed in the `package.json file`, the application/project along with this `package.json` file when moved to another system might not be able to work when uploaded to this system or a foreign location like a github repository. Since `package.json` is basically used as the file, node will base on when the command `npm install` is ran in the folder containing this `package.json` file (because to run other developers projects/applications we need to install the dependencies that is in the package.json file through the `npm install` command which must be ran inside this folder containing the `package.json` file), this `package.json` file because it virtually has no information with regards to all the modules/packages the project/application uses, the `npm install` command will run yes but no packages will be installed in this different system since the `dependencies` key is missing

`npm install` also installs all the modules/packages under the `devDependencies` key which are the modules only used in a development setting, and are not necessarily deployed for marketing and use unlike the `dependencies` key. For other commands like npm install --production or `npm install –p` it installs only the packages under the `dependencies` key and not the `devDependencies` key in the `package.json` file.

To use an installed package/module inside a JavaScript file this file must be inside the folder containing the `node_modules` folder and `package.json` file. Note that the name of this JavaScript file must be the same name as the value of the `main` key in the `package.json` file which is usually named `index.js` as its default value. To import the node package/module that has been installed we only need to use the `require()` function which takes in a string of the name of the module installed

```
index.js

const _ = require('lodash'); // const _ = require('<nameofmoduleinstalled>');

const arr = [4, 1, 9324, 23, 1];


_.forEach(arr, (num, index) => {

    console.log(`${num} at index ${index}`);

});


// cannot be used

import forEach from 'lodash';


forEach(arr, (num, index) => {

    console.log(`${num} at index ${index}`);

});
```

Finally for the uninstallation process the same concept of installing packages saved in the `dependencies` and `devDependencies` key can be applied here, which means `npm uninstall <packagename> --save` will uninstall or remove the package in the `node_modules` folder as well as remove the package name saved as a string under the `dependencies` key or `devDependencies` key should one use `npm uninstall <package> --save-dev`. Some other commands also include `npm install <packagename@version (e.g. 1.5.1)> --save` and `npm update <packagename>`

When installing a pre-made react project template why `npm install –g create-react-app` is done first is because to use `npx create-react-app <projectname>` the `create-react-app` package must be installed first globally which will be placed in a node_modules folder located in the location that we used to add to the path in our environment variables which is , or the global folder to put it more simply, so we can access this npx command even at other locations not only at a local location where the node_modules and the create-react-app package inside it exists

npm install <package> used in a specific folder will install the package and create a node_modules folder to place this package inside this folder so that when installation is finished this node_modules folder will be inside this specific folder

npm uninstall <package> used in a specific folder will uninstall the package in the node_modules folder located in the same specific folder

installing packages outside of the location of the already existing node_modules folder will just create another node_modules folder that contains the packages we will install and will be separate to this already existing node_modules folder

uninstalling packages outside of the location of the already existing node_modules folder will not uninstall anything especially if no node_modules folder exists in the location where the npm uninstall <package> command was used

Versioning

Usually a node package/module and perhaps even other packages of other language will have 3 digits separated by dots that represent the current version the package identifies with. The first digit being the major version simply means that if a package is updated in a major manner in the package.json file, some packages that depend on it might break the project/application or even the other packages that depend on this package/module, entirely. The second digit being the minor version simply means that new features have been added to the package/module and won't break the project/application or any other package that may depend on this updated package. And lastly the third digit being the patch or bug fixes of a package/module simply means that if bugs are found and fixed in the package then no possibility of breaking a project/application or any other package that depends on this updated package will occur.

One may also notice in the package.json file under the dependencies and devDependencies key the exponent, tilde, or a "no character" character before the version number of the package. This simply means that should the project/application be deployed/moved somewhere in another system or repository, and used by another user, and this user installs the list of packages in the package.json file, the package that uses an exponent node will essentially update the package in a minor way when the package is installed. If the package uses a tilde it essentially uses a patch or bug fix update, and finally if no character is used when the package is installed the exact version of the package is kept.

Setting up a react project

Essentially there are two ways to setup a react project one is simply by using the series of commands `npm install create-react-app` or `npm i create-react-app` then the `npx create-react-app <projectname>` command in the location where the previous command was used

If we install `create-react-app` globally through the command `npm –g install create-react-app` then we are able to use the `npx create-react-app <projectname>` anywhere even in locations where the `node_modules` folder that must contain the `create-react-app` module isn’t installed

However, using `npx create-react-app <project folder name>` is a time consuming as well as storage consuming task because this command for lack of a better term installs node modules that we aren't really going to need in our project unless we are going to build a full-fledged application that is on par with what companies build every day, which we obviously would not do more often than not as perhaps lone developers. So to setup our own react environment, we would need to install the ff. using `npm install`: `react`, `react-dom`. This is merely for our main dependencies but for our development dependencies we would run `npm install <pkg> --save-dev` which would be the ff: **`webpack`** for bundling our modules and so we can able to access the set of commands for our react project like run build and start, **`webpack-cli`**, **`webpack-dev-server`** for when our code is changed live server will automatically update the DOM and load the page again, **`@babel/core`** to transpile our code, **`babel-loader`** to preprocess imported media and stylesheet files other than **`.js`** or **`.jsx`** in our react app like **`.png`**, **`.jpg`**, **`.jpeg`**, **`.gif`**, **`.css`**, **`.scss`**, or **`.sass`** files, **`@babel/preset-react`** which contains presets for all react plugins and supports language features like **`.jsx`** files and ES5 & ES6 JavaScript, **`@babel/preset-env`** to make our compiled **`.js`** files smaller when we do run npm run build , **`html-webpack-plugin`**, **`css-loader`** to be able to import **`.css`** files, **`sass-loader`** to import **`.sass`** and **`.scss`** files, and lastly **`file-loader`** to be able to import **`.png`**, **`.jpg`**, **`.jpeg`**, and **`.gif`** files to name a few.

Generally, the file structure we have to follow to ease our environment set up would be the following:

```
<project folder name>
├── build
├── node_modules
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
├── src
│   ├── assets
│   │   └── images
│   │       └── logo.svg
│   │   └── stylesheets
│   │       ├── sample.css
│   │       ├── sample.scss
│   │       └── sample.scss
│   ├── components
│   │   └── Home.jsx
│   ├── pages
│   ├── context
│   ├── utilities
│   ├── index.js
│   ├── App.js
│   └── service-worker.js
├── .gitignore
├── .env
├── webpack.config.js
├── package.json
└── README.md
```

And In our `webpack.config.js` file we write the following configuration code

```
const path = require('path');

const HTMLWebpackPlugin = require('html-webpack-plugin');



module.exports = {
    // tell webpack the entry point of our app, to create dependency graph
    entry: './src/index.js',
    // tell webpack name and location of bundled file
    output: {
        // __dirname currnetly ../sandigan so when joined to /dist
        // resulting path is ../sandigan/dist
        path: path.join(__dirname, '/build'),
        filename: 'bundle.js'
    },

    plugins: [
        new HTMLWebpackPlugin({
            // path of our main html file where we render our react code
            template: './public/index.html'
        })
    ],
    resolve: {
        modules: [__dirname, "src", "node_modules"],
        //
        extensions: [".*", ".js", ".jsx", ".tsx", ".ts"],
    },

    module: {
        rules: [
            {
                // transpiles all our files that end both with .js or .jsx by
default, but since
                // extensions array where *, .js, .tsx, and .ts is included
aside from
                // .jsx it will compile also files with these extensions
                test: /\.(js|jsx)$/,
                // we want to also exclude folders liek node moduels when babel
transpiles them
                exclude: /node_modules/,
                use: {
                    // transpiles our .js, .jsx files with preset-env and
preset-react
                    loader: 'babel-loader',
                    options: {
                        presets: [
                            '@babel/preset-env',
                            '@babel/preset-react'
                        ]
                    }
                }
            },
            {
                // transpiles all our stylesheet files that have extensions
.scss, .sass, or css
                // node this needs style-loader, css-loader, sass-loader, and
sass to be installed
                // if css only then only css-loader and style-loader will be
needed in devDependencies
```

```
                test: /\.(s[ac]ss|css)$/i,

                use: ['style-loader', 'css-loader', 'sass-loader']

            },

            {

                // transpiles all our media files that have extensions .png,
.jpeg, .jpg, .gif, and .svg

                test: /\.(png|jpe?g|gif|svg)$/i,

                use: [

                    {

                        loader: 'file-loader',

                    }

                ]

            }

        ]

    }
```

}From here we just copy the typical react code we use in our index.js, App.js, and in our initial .jsx file which in this case is our Home.jsx file

package-lock.json is the file that locks on to the current version of the node packages installed

are things that you should never ever edit manually

package.json if npm install react and npx create-react-app is used and is also the manifest file that holds the information of our project/application like its name, the version, author, license and especially the dependencies or the modules that our application depends on whenever we make an import in our application

```
{

  "name": "django-react-project",

  "version": "0.1.0",

  "private": true,

  "dependencies": {

    "@material-ui/core": "^4.12.2",

    "@testing-library/jest-dom": "^4.2.4",

    "@testing-library/react": "^9.5.0",

    "@testing-library/user-event": "^7.2.1",

    "antd": "^4.15.1",
    "react": "^16.13.1",
    "react-addons-pure-render-mixin": "^15.6.3",
    "react-dom": "^16.13.1",
    "react-native": "^0.63.3",
    "react-router-dom": "^5.2.0",
    "react-scripts": "3.4.3"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Adding scripts

```
{
  "name": "react-projects",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "Michael Cueva",
  "license": "MIT",
  "dependencies": {
    "lodash": "^4.17.21"
  }
}
```

We can copy the aboce scripts but we can use the the following scripts in the form of key value pairs

Node version manager

Setting up environment for Svelte.js development

Assuming npm has already been installed in our system using node we type in our command prompt npm create vite@latest, which run and then prompt us with the

```
D:\Projects\To Github>npm create vite@latest
Need to install the following packages:
  create-vite@latest
Ok to proceed? (y) y
√ Project name: ... project-alexander-test
√ Select a framework: » Svelte
√ Select a variant: » JavaScript

Scaffolding project in D:\Projects\To Github\project-alexander-test...

Done. Now run:

  cd project-alexander-test
  npm install

  npm run dev
```

Which will build a file structure like this:

```
project-alexander-test

      | - node_modules

      | - public

      | - src

            | - assets

            | - fonts

            | - mediafiles

            | - stylesheets (this is where we can add the custom.scss file to
be able to compile it to the custom.css file and then add it to our index.html
file in the base directory of our project)

            | - components

                  | - <component we have>.svelte

            | - App.svelte

            | - main.js

      | - index.html

      | - jsconfig.json

      | - package-lock.json

      | - package.json

      | - README.md

      | - .gitignore

      | - svelte.config.js

      | - vite.config.js
```

Now we can run in here our local development server like we would do in a React.js project where we run whatever command that our .config files have.

Setting up environment for python development

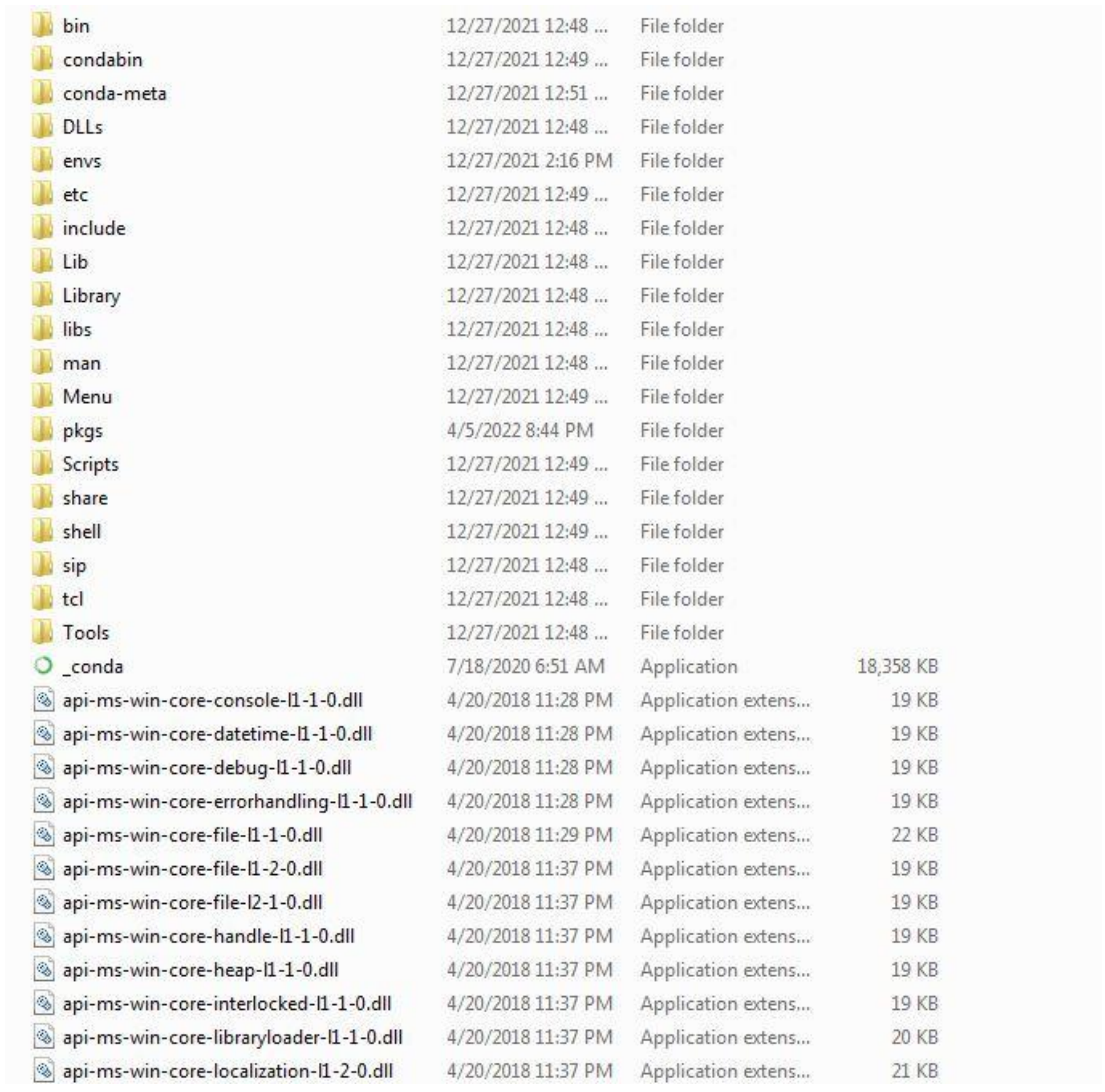
There are basically two interpreters for python one that is downloaded in https://www.python.org/downloads/ which is the official python interpreter and the other alternative interpreter mainly used in higher level programming like in the fields of machine learning and AI, named conda which can be downloaded here https://www.anaconda.com/products/distribution or https://repo.anaconda.com/archive/ to pick out a specific version of conda.

Remember there are other python versions that will not be compatible with the other interpreter conda installing interpreters and compilers

always add to path that the commands that comes with these interpreters or compilers can be used for the

Installation conda and python and adding its locations to PATH

Once anaconda is downloaded when installed it is good practice to select the option of not adding conda to our path in our environment variables automatically via the anaconda installer. Once installed now we add conda to our path in our environment variables. Basically a folder called `Anaconda3` will be created usually in the locations `C:\Users\<Users>\AppData\Local or C:\ProgramData` when conda is installed. We then navigate inside this `Anaconda3` folder and we copy our current location since this location will contain the interpreter application itself for us to use in interpreting our python code. It should be noted that in other cases the `Anaconda3` folder may be a different name depending if you've renamed this default folder name to another name in the installation process for example `Anaconda3_`. Once one has navigated in the location where the interpreter is in we copy this location which will be `C:\Users\<User>\AppData\Local\Anaconda3` or `C:\ProgramData\Anaconda3` and add it to the path in our environment variables with a semi colon at the end, `C:\Users\<User>\AppData\Local\Anaconda3;` or `C:\ProgramData\Anaconda3;`. However not only the location of this folder is needed to be added to our path in our environment variables.



bin	12/27/2021 12:48 ...	File folder	
condabin	12/27/2021 12:49 ...	File folder	
conda-meta	12/27/2021 12:51 ...	File folder	
DLLs	12/27/2021 12:48 ...	File folder	
envs	12/27/2021 2:16 PM	File folder	
etc	12/27/2021 12:49 ...	File folder	
include	12/27/2021 12:48 ...	File folder	
Lib	12/27/2021 12:48 ...	File folder	
Library	12/27/2021 12:48 ...	File folder	
libs	12/27/2021 12:48 ...	File folder	
man	12/27/2021 12:48 ...	File folder	
Menu	12/27/2021 12:49 ...	File folder	
pkgs	4/5/2022 8:44 PM	File folder	
Scripts	12/27/2021 12:49 ...	File folder	
share	12/27/2021 12:49 ...	File folder	
shell	12/27/2021 12:49 ...	File folder	
sip	12/27/2021 12:48 ...	File folder	
tcl	12/27/2021 12:48 ...	File folder	
Tools	12/27/2021 12:48 ...	File folder	
_conda	7/18/2020 6:51 AM	Application	18,358 KB
api-ms-win-core-console-l1-1-0.dll	4/20/2018 11:28 PM	Application extens...	19 KB
api-ms-win-core-datetime-l1-1-0.dll	4/20/2018 11:28 PM	Application extens...	19 KB
api-ms-win-core-debug-l1-1-0.dll	4/20/2018 11:28 PM	Application extens...	19 KB
api-ms-win-core-errorhandling-l1-1-0.dll	4/20/2018 11:28 PM	Application extens...	19 KB
api-ms-win-core-file-l1-1-0.dll	4/20/2018 11:29 PM	Application extens...	22 KB
api-ms-win-core-file-l1-2-0.dll	4/20/2018 11:37 PM	Application extens...	19 KB
api-ms-win-core-file-l2-1-0.dll	4/20/2018 11:37 PM	Application extens...	19 KB
api-ms-win-core-handle-l1-1-0.dll	4/20/2018 11:37 PM	Application extens...	19 KB
api-ms-win-core-heap-l1-1-0.dll	4/20/2018 11:37 PM	Application extens...	19 KB
api-ms-win-core-interlocked-l1-1-0.dll	4/20/2018 11:37 PM	Application extens...	19 KB
api-ms-win-core-libraryloader-l1-1-0.dll	4/20/2018 11:37 PM	Application extens...	20 KB
api-ms-win-core-localization-l1-2-0.dll	4/20/2018 11:37 PM	Application extens...	21 KB

This is because we need to also add the other locations `C:\Users\<User>\AppData\Local\Anaconda3\Scripts;` or `C:\ProgramData\Anaconda3\Scripts;` and `C:\Users\<User>s\AppData\Local\Anaconda3\Library\bin;` or `C:\ProgramData\Anaconda3\bin;` in order for our interpreter to run and in order to have access to conda related commands using the conda keyword in the command line in any location, respectively. And in order to copy these locations, once in the `Anaconda3` folder we navigate to the `Scripts` folder, copy the current location, add to the path in the environment variables and then go back and navigate to the `bin` folder, copy the current location, and finally add to the path in the environment variables. Once we have added these 3 locations in to our path in the system environment variables we can finally use a list of all conda related commands.

Moreover a developer needs to install a python interpreter as well that can be downloaded in the site mentioned previously. For the sake of understanding also for now, compatibility between versions of python and conda, as well as compatibility between versions of python and python packages will not be tackled yet. Now usually when installing python using the installer program it won't ask where python will be installed in the system/device but it is usually installed in the location C:\Users\<User>\AppData\Local\Programs. Inside this folder will be the folder named `Python` and inside it the `Python<version number>-<system type>` folder (the version number of python and system type for this machine/device for example is 3.8 and 32-bit which means the folder name will be `Python38-32`) which will contain the interpreter itself. Essentially we need to add the locations where the interpreter is located and the python scripts that allow us to use the `python` keyword for python related commands, to our path in our environment variables in order to fully setup our python environment. These are `C:\Users\<User>\AppData\Local\Programs\Python\Python38-32\Scripts\;` and `C:\Users\<User>\AppData\Local\Programs\Python\Python38-32\;`.

Note that sometimes there will be a missing dll file error when attempting to test if conda is indeed installed using conda -V. This can be fixed by copying and pasting the missing dll files into the bin folder of conda which can be found in

Creating virtual environment using conda

Because conda and python's necessary folder locations have already been added to the path in our environment variables, to create a virtual environment which a developer can enter at any time to be able to run a python file or script that uses a module/package

Because if this python file that uses these packages/modules is ran outside the environment where obviously and possibly no python modules/packages are installed, the python file might not run since it will keep looking for the package/module that does not exist in its environment. This is why when we run a python file that uses a package/module installed in a specific environment we must always be in that environment where it is installed in order to run the python file/script that depends and uses these installed packages/modules

```
C:\Users\Cueva>activate


(base) C:\Users\Cueva>conda activate sjka

Could not find conda environment: sjka

You can list all discoverable environments with `conda info --envs`.
```

```
(base) C:\Users\Cueva>conda info -e

# conda environments:

#

base                  *  C:\ProgramData\Anaconda3

mlwdenv                  C:\ProgramData\Anaconda3\envs\mlwdenv


(base) C:\Users\Cueva>
```

Basically commands in conda are divided into numerous parts depending on the command, which can be added to it if need be. Every command has a root command name, such as conda `create`, conda `install`, conda `remove`, and conda `list` which are some of the more important ones.

The command conda `create` has 10 sets of parts that are part of an overall command that we can use to form and run this overall command. Note that the angular braces written below indicate its value, and if the value has a white space enclose the whole value in double quotations

1. first set
   - `–h`
   - `--clone <env name>` – the `<env name>` is the path to or the name of the existing local environment
   - `-n <env name>` or `-p <full path name of env location>`
- `-c <channel name or url name>` – sometimes when installing a conda package it will not use the simple command of conda `install <packagename>` sometimes it will have to include this `–c` command then the `<channel name>` or the `<url name>` what website conda will get the package from. Suppose the package is from anaconda.org, when we do visit the website for our package, we see the specific command that we have to use in order to install this package and it tells still the same thing that we need to include the channel or website url where this package is from in order to install it, which in this case is if the package is from anaconda.org then the command will be conda `create –n anaconda <packagename>`. This is used because using just an ordinary installation of a package without the channel name might not install the package since this ordinary installation normally gets its packages from the official anaconda website which may not have some packages that we want. To solve this there are other channels like `conda-forge` which have channels used in an installation command such as this: conda `install -c conda-forge/label/cf202003 djangorestframework`, `conda-forge` being the website the package is downloaded from and installed
2. Second set
   - `--use-local`
   - `--override-channels`
3. Third set
   - `--repodata-fn <repodata_fns>`
   - `--strict-channel-priority`
4. Fourth set
   - `--no-channel-priority`
   - `--no-deps or –only-deps`
5. Fifth set
   - `--no-pin`
   - `--copy`
   - `-C`
   - `-k`
   - `--offline`

- `-d`
- `--json`

6. Sixth set
   - `-q`
   - `-v`
   - `-y`
   - `--download-only`
   - `--show-channel-urls`
7. Seventh set
   - `--file=<file name>`
   - `--no-default-packages`
8. Eighth set
   - `--experimental-solver {classic, libmamba, libmamba-draft}`
9. ninth set
   - `--dev`
10. Tenth set
    - `<package name==version number>` or `<package I name>` `<package II name>` `<package III name>` – these are the packages to install or update in the conda environment

In a simple manner we can use the most common ways of using this root command through the use of a simple conda `create –n <env name> <package name>`

```
conda create -n <yourenvname> python=x.x anaconda
```

```
conda create -n test_env python=3.6.3 anaconda
```

The theory for this command is that whenever we use a indicate the python version as part of our command and then preceded by this the packages we want to install, conda installs these packages that will most surely be compatible with the python version that we indicated. So packages installed will have versions compatible with this version of python

Once the environment is created along with its packages inside that were installed during creation, we can now enter newly created virtual environment by activating it. To activate our environment we will need to type two consecutive commands in order to do so which is first `activate`, which will show in our command line that we are in the base environment by placing a `(base)` beside our current location

```
(base) C:\Users\Cueva>
```

For the second and final phase of activation of our environment we type in conda `activate <env name>` which will change the `(base)` to `(<env name>)` beside our current location. Because we have done this we can install and uninstall different packages that do not yet exist or already exist in our environment, because every time we leave our environment or enter another one we essentially change the location as well of where our packages are installed or uninstalled should we do so, meaning that if we were to enter the base environment and install a package there it would not only install it only in this base environment but should we run a python script/file that uses such a package in our other environment we will be unable to run this python file since this package we used essentially does not exist under current environment. But we can yes install and uninstall packages in our desired environment even if we are not in that environment through a simple conda `install` or `uninstall` but we need to specify its different parts to form the overall command that will install or uninstall a package in our desired environment even if we are outside that environment. But for the sake of ease and understanding we first must always enter our environment first before installing or uninstalling a package.

`conda install` has about 12 sets of parts that are part of an overall command that we can use to form and run this overall command.

1. First set
   - `-h`
   - `--revision <revision name>`
   - `-n <env name>` or `-p <full path to env location>`
2. Second set
   - `–c <channel name>`
   - `--use-local`
   - `--override-channels`
3. Third set
   - `--repodata-fn <repodata_fns>`
   - `--strict-channel-priority`
4. Fourth set
   - `--no-channel-priority`
   - `--no-deps or –only-deps`
5. Fifth set
   - `--no-pin`
   - `--copy`
   - `-C`
   - `-k`
   - `--offline`
   - `-d`
   - `--json`
6. Sixth set
   - `-q`
   - `-v`
   - `-y`
   - `--download-only`
   - `--show-channel-urls`
7. Seventh set
   - `--file=<file name>`
8. Eighth set
   - `--experimental-solver {classic, libmamba, libmamba-draft}`
9. Ninth set
   - `--force-reinstall`
10. Tenth set
    - `--freeze-installed or --update-deps` or `–S or --update-all or --update-specs`
11. Eleventh set
    - `-m`
    - `--clobber`
    - `--dev`
12. Twelfth set
    - `<package name==version number>` or `<package I name> <package II name> <package III name>` – these are the packages to install or update in the conda environment

When installing packages as well that does not exist on the official conda site, search in https://anaconda.org/conda-forge/ or https://anaconda.org/anaconda/psycopg2 if there is such a package. Usually the package will exist here in this channel and once we finally pick a package we can click the page which will contain and display the following commands for us to install it in the command line. Take note that the first command is usually the safer and widely used by developers for lack of a better word. Usually conda `install -c conda-forge <packagename>` will be the command used.

- `conda install -c conda-forge djangorestframework`
- `conda install -c conda-forge/label/gcc7 djangorestframework`
- `conda install -c conda-forge/label/cf201901 djangorestframework`
- `conda install -c conda-forge/label/cf202003 djangorestframework`
- `conda install -c conda-forge django`
- `conda install –c conda-forge django-cors-headers`
- `conda install –c anaconda psycopg2`

`conda remove` has about 6 sets of parts that are part of an overall command that we can use to form and run this overall command.

1. First set
   - `-h`
   - `-n <env name>` or `–p <full path to env location>`
   - `-c <channel name>`
   - `--use-local`
2. Second set
   - `--override-channels`
   - `--repodata-fn <repodata_fns>`
   - `--all` – using this will remove all packages in the environment
3. Third set
   - `--features`
   - `--force-remove`
   - `--no-pin`
4. Fourth set
   - `--experimental-solver {classic, libmamba, libmamba-draft}`
5. Fifth set
   - `-C`
   - `-k`
   - `--offline`
   - `-d`
   - `--json`
   - `-q`
   - `-v`
   - `-y`
   - `--dev`
6. Sixth set
   - `<package name==version number>` or `<package I name> <package II name> <package III name>` – these are the packages to install or update in the conda environment

This command will also remove any package that depends on any of the specified packages as well, unless a replacement can be found without that dependency. If you wish to skip this dependency checking and remove just the requested packages, add the `--force` option. conda `remove –n <env name> --all`

These are just some of the most important conda commands used in a development setup that uses conda. And if we were to finally leave our environment we would just type the command `conda deactivate` to leave our environment (note this applies as well if we are in the base environment), this does not delete our environment but just allows us to exit our environment safely and still retain the packages we have installed in our environment.

Some useful but not that important commands as well is `conda info –e` – which displays all the existing environments

```
(base) C:\Users\Cueva>conda info -e

# conda environments:

#

base                  *  C:\Users\Cueva\AppData\Local\Anaconda3_

MLWDenv                  C:\Users\Cueva\AppData\Local\Anaconda3_\envs\MLWDenv
```

Another is `conda list` and like other commands, to ease our use of this commands other parts we can just enter in our environment and use this command once we have entered in our environment. When we do use this command it provides a tabled list of all the packages/modules installed in the environment we are currently in. And in each column of the table it specifies the name of the package, the version of the package, the build information of the package, and the channel or website where the package was available for download and installation.

```
# packages in environment at C:\ProgramData\Anaconda3\envs\mlwdenv:

#

# Name                    Version                   Build  Channel

asgiref                   3.4.1              pyhd8ed1ab_0    conda-forge

ca-certificates           2020.10.14                    0    anaconda

certifi                   2020.6.20                 py37_0    anaconda

django                    4.0                pyhd8ed1ab_0    conda-forge

django-cors-headers       3.10.1             pyhd8ed1ab_0    conda-forge

djangorestframework       3.13.1             pyhd8ed1ab_0    conda-forge

krb5                      1.17.1               hc04afaa_0    anaconda

libpq                     12.2                 h3235a2c_0    anaconda

openssl                   1.1.1l               h8ffe710_0    conda-forge

pip                       21.2.4           py37haa95532_0

psycopg2                  2.8.5            py37h7a1dbc1_0    anaconda

python                    3.7.11               h6244533_0

python_abi                3.7                     2_cp37m    conda-forge

pytz                      2021.3             pyhd8ed1ab_0    conda-forge

setuptools                58.0.4           py37haa95532_0

sqlite                    3.36.0               h2bbff1b_0

sqlparse                  0.4.2              pyhd8ed1ab_0    conda-forge

tk                        8.6.10               he774522_0    anaconda

typing_extensions         4.0.1              pyha770c72_0    conda-forge

vc                        14.2                 h21ff451_1

vs2015_runtime            14.27.29016          h5e58377_2
```

```
wheel                     0.37.0            pyhd3eb1b0_1

wincertstore              0.2             py37haa95532_2

zlib                      1.2.11          vc14h1cdd9ab_1  [vc14]  anaconda
```

And because most likely we will be working in our environment where our packages are also installed; in such a time when we decide to deploy our python project files/scripts we can use the command `conda list --export > <textfilename>.txt` which will essentially make a file similar to `package.json` in node containing all the packages we have installed in our environment, so that other people can use and run our python project files/scripts by first installing these packages and modules in their desired environments in their own device/system. This command moreover is an alternative to the `pip freeze > <textfilename>.txt` which also makes a file containing all the packages and its version information for other people to use alongside the deployed project of a developer.

Creating environments and installing packages under different python versions

Because we know the basics in creating a virtual environment and installing packages in the environments that we create it is important to know that in every environment that we create a certain version of python will always be used to operate in that environment. When we do create an environment using `conda create –n <envname>` it has already been established that this creates an environment that we can always enter in and install different python packages, but more than that is that conda creates this environment that will use the current version of python that we have installed in our machine system as the default if the python version is not provided in the command used to create an environment. Should we decide to provide a python version that our environment will use we add the part `python=<x>.<x>` `x` being the two `x's` being the version number that will form the overall command `conda create –n <envname> python=<x>.<x> <packagename I>…<packagename n>`.

Whenever we enter an environment we virtually and automatically use the version of python used in this environment when the environment was first created. Coding in python can be done either in IDE's such as VS Code and in the built in python IDLE in the command line; and this is where knowing what environment to use becomes useful, because we know that different environments may use different python versions, and it is imperative to know that every time we use platforms like a code editor or the command line we are given the option to choose the python version we want to write code with, depending on what environment we choose to enter in. In VS Code to we normally select our workspace first and then we decide what environment that has a specific version of python do we enter in. Finally should we use the command line, to write python code, the concept of selecting what environment that has a specific version to use is relatively the same, that is because when we enter an environment aside from the `base` environment using the two step activation command which are `conda activate/activate` and then `conda activate <envname>` we may potentially change our python version if and only if the environment we enter in has used a different python version upon which when the environment was first created.

```
C:\Users\Cueva>python --version

Python 3.8.2


C:\Users\Cueva>activate
```

```
C:\Users\Cueva>conda.bat activate


(base) C:\Users\Cueva>python --version

Python 3.8.3


(base) C:\Users\Cueva>conda activate mlwdenv


(mlwdenv) C:\Users\Cueva>python --version

Python 3.7.11
```

In the above information we see that in the command line all the environments we have. And when we do enter in the environments and use the command `python --version` to see what python version is used in the environment we automatically see that each environment differs in terms of their versions of python.

Because we use different python versions in each environment it is imperative to know that packages installed in these environments are also installed in order to be compatible to the python version the environment is using. For instance if an environment uses python 3.8.3 it will install a package only compatible to this version, moreover when we do decide to leave an environment and we use a package that was supposedly installed in that environment we previously left and used it in this new environment, it may potentially give an error since no package may exist in this new environment we are in.

```
(mlwdenv) C:\Users\Cueva>python

Python 3.7.11 (default, Jul 27 2021, 09:42:29) [MSC v.1916 64 bit (AMD64)] ::
Anaconda, Inc. on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> import django

>>> import numpy

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ModuleNotFoundError: No module named 'numpy'

>>>
```

This example shows how our command line gives an error simply because of the fact that the package numpy is not installed in the environment named `mlwdenv`. If we check the packages in this environment itself by using the command `conda list` which will list all the packages installed in an environment

```
(mlwdenv) C:\Users\Cueva>conda list

# packages in environment at C:\ProgramData\Anaconda3\envs\mlwdenv:

#

# Name                    Version                   Build  Channel

asgiref                   3.4.1              pyhd8ed1ab_0    conda-forge
```

```
beautifulsoup4            4.9.3                  pyhb0f4dca_0    anaconda
…
cython                    0.29.28              py37hd77b12b_0
django                    3.1.2                          py_0    anaconda
django-cors-headers       3.10.1                 pyhd8ed1ab_0    conda-forge
djangorestframework       3.13.1                 pyhd8ed1ab_0    conda-forge
djangorestframework-jwt   1.11.0                         py_0    conda-forge
…
pillow                    8.0.0                py37hca74424_0    anaconda
pip                       21.2.4               py37haa95532_0
psycopg2                  2.8.5                py37h7a1dbc1_0    anaconda
pycparser                 2.21                   pyhd8ed1ab_0    conda-forge
pyjwt                     1.7.1                          py_0    conda-forge
…
zstd                      1.4.4                    ha9fde0e_3    anaconda
```

Now if we enter another environment wherein these packages that would have otherwise worked in our other environment were installed and used in the built in python command line interface we would see that those packages when imported and used in our command line interface would work just fine.

```
 (base) C:\Users\Cueva>python
Python 3.8.3 (default, Jul  2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> import scipy
>>> import matplotlib
>>> import sklearn
>>> import pandas
>>>
```

This works because we see in this `base` environment that the packages numpy, `scipy`, `matplotlib`, `scikit-learn/sklearn`, and `pandas` are installed in this environment with this python version

```
(base) C:\Users\Cueva>conda list


astropy                   4.0.1.post1      py38he774522_1
atomicwrites              1.4.0                    py_0
attrs                     19.3.0                   py_0
```

```
autopep8                  1.5.3                      py_0
babel                     2.8.0                      py_0
backcall                  0.2.0                      py_0
backports                 1.0                        py_2
backports.functools_lru_cache 1.6.1                      py_0
backports.shutil_get_terminal_size 1.0.0                      py38_2
backports.tempfile        1.0                        py_1
backports.weakref         1.0.post1                  py_1
bcrypt                    3.1.7            py38he774522_1
beautifulsoup4            4.9.1                    py38_0
…
cython                    0.29.21          py38ha925a31_0
…
flask                     1.1.2                      py_0
…
jsonschema                3.2.0                    py38_0
jupyter                   1.0.0                    py38_7
jupyter_client            6.1.6                      py_0
jupyter_console           6.1.0                      py_0
jupyter_core              4.6.3                    py38_0
jupyterlab                2.1.5                      py_0
jupyterlab_server         1.2.0                      py_0
…
matplotlib                3.2.2                         0
matplotlib-base           3.2.2            py38h64f37c6_0
…
nltk                      3.5                        py_0
…
numba                     0.50.1           py38h47e9c7a_0
numexpr                   2.7.1            py38h25d0782_0
numpy                     1.18.5           py38h6530119_0
numpy-base                1.18.5           py38hc3f5095_0
numpydoc                  1.1.0                      py_0
…
pandas                    1.0.5            py38h47e9c7a
```

```
…
pillow                    7.2.0            py38hcc1f983_0
pip                       20.1.1                   py38_1
…
scikit-image              0.16.2           py38h47e9c7a_0
scikit-learn              0.23.1           py38h25d0782_0
scipy                     1.5.0            py38h9439919_0
seaborn                   0.10.1                     py_0
…
watchdog                  0.10.3                   py38_0
…
zstd                      1.4.5                ha9fde0e_0
```

Setting up a database for projects and finished applications

It is imperative as a developer of both the front end and back end of websites that a database be set up, in order for the project or the application to retrieve and store information whenever a user wants to. Essentially there are two kinds of databases: SQL that uses a structured way of storing data, by rows and columns and no-SQL which is otherwise. Generally for medium to smaller applications SQL is ideal to use since it is structured in a manner that data is easy to get, for bigger and more complex applications however no-SQL is preferred since it is much faster than a SQL database, but generally is not favorable for use in smaller applications. But for the sake of the understanding of setting up a database the concepts aforementioned will be all in this topic. This will cover only making an account for a database in order to store our data by our project or application; moreover this is because that in each database we create for our project and application to use it will always be under a certain account, which has credentials, the username and password to the account that holds our databases as such, that we must always include in the settings of our backend.

Unlike Postgres which is usually paired with Django, Django has an admin panel which uses a GUI in the form of a website that can directly interact with the database and add our models, moreover Django has a built in command line interface which we can use to import our models and interact with these models which therefore may change the state of our data in our database. But other databases such as mongoDB, or any other structured query language for that matter will have certain syntax which we can use to interact with our database like the actions creating, reading, updating, and deleting data.

To download the mongoDB installer is the first thing

Moving on to create an account in order to manage the databases we will need in our application we first need to download the software entitled pgAdmin v4 which administrates our postgresql databases through a browser. Once the installer for pgAdmin v4 is downloaded it will ask us to enter a username and a password – this is will be vital for this will be the account itself that will manage all the databases under it which will moreover be used potentially frequently by our project and application itself.

Installing packages with compatible versions

sass

visual studio code

All Paths Needed To Be Added In System Environment Variables

- For using gcc or g++ compiler and 'gcc'/'g++' commands
  - C:\Program Files\mingw-w64\x86_64-8.1.0-posix-seh-rt_v6-rev0\mingw64\bin;
  - E:\Compilers and Interpreters\mingw64\bin

- For using sass compiler to convert sass files to css files and 'sass'/'scss' commands
  - E:\Projects\sass\dart-sass

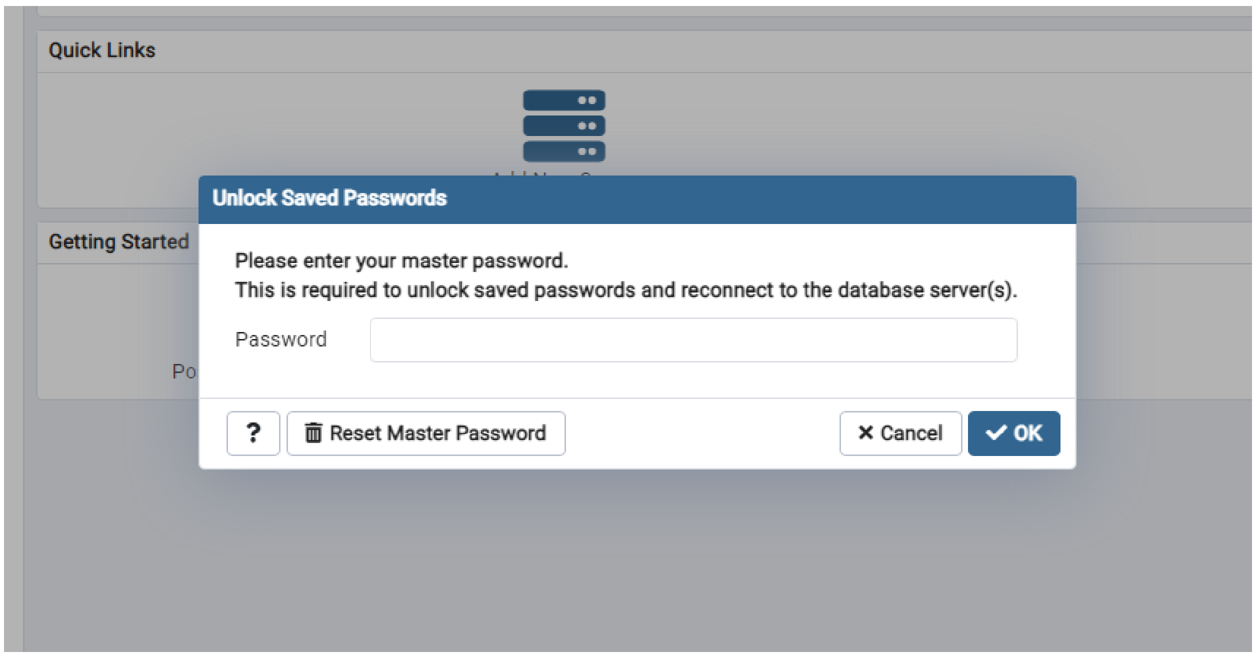
Postgresql

Got to installation location of PostgreSQL

Copy the path of the bin folder: C:\Program Files\PostgreSQL\14\bin

Paste it in environment variables for user or the whole system

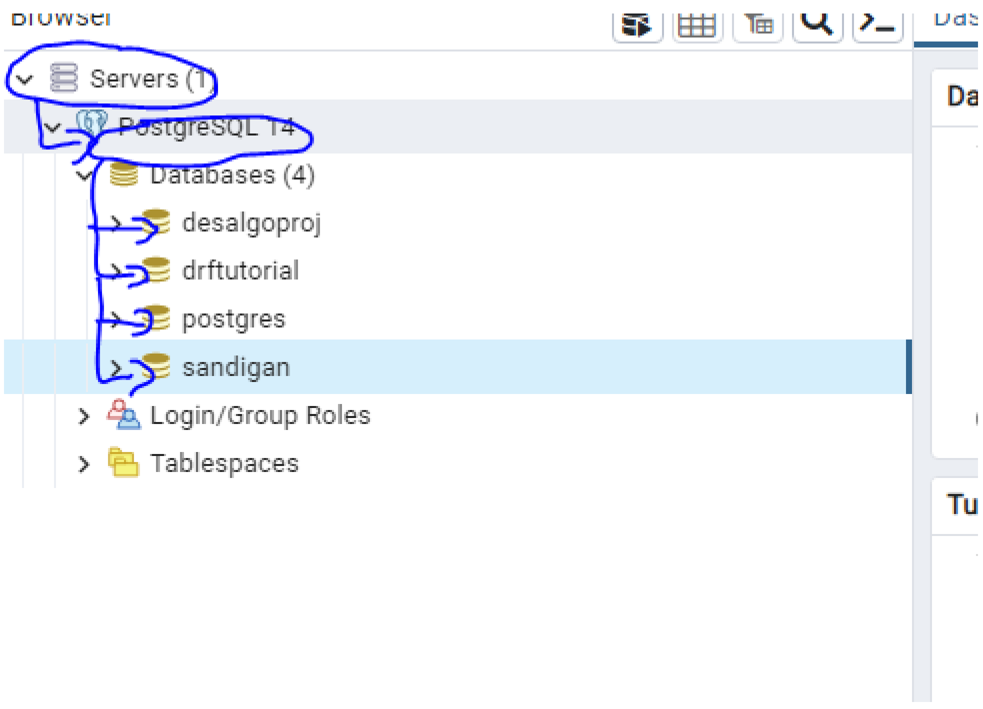
```
C:\Users\Mig>psql -U postgres
Password for user postgres:
psql (14.4)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=#
```


Unlock Saved Passwords

Please enter your master password.
This is required to unlock saved passwords and reconnect to the database server(s).

Password

Take note that the master password of the pgAdmin application itself is different from the password of the servers of the application. To understand this further, hierarchically it can be thought of the App nesting the server and the server nesting a, or the database/s itself.



```
postgres=# \l

                                List of databases

    Name       |  Owner    | Encoding |          Collate              |           Ctype
|    Access privileges

-------------+---------+---------+------------------------+-------------
------------+----------------------

 desalgoproj   |  postgres  |  UTF8          |  English_Philippines.1252   |
English_Philippines.1252 |

 drftutorial   |  postgres  |  UTF8          |  English_Philippines.1252   |
English_Philippines.1252 |

 postgres          |  postgres  |  UTF8          |  English_Philippines.1252   |
English_Philippines.1252 |
 sandigan        |  postgres  |  UTF8          |  English_Philippines.1252  |
English_Philippines.1252 |

 template0      |  postgres  |  UTF8          |  English_Philippines.1252  |
English_Philippines.1252 | =c/postgres          +

             |              |             |                                  |
| postgres=CTc/postgres

 template1      |  postgres  |  UTF8          |  English_Philippines.1252  |
English_Philippines.1252 | =c/postgres          +

             |              |             |                                  |
| postgres=CTc/postgres

(6 rows)
```

Connecting to different databases

```
postgres=#  \c desalgoproj

You are now connected to database "desalgoproj" as user "postgres".

desalgoproj=#
```

In clearing the output of the PostgreSQL command line tool the command is as simple as `\! cls` if in Windows and `\! Clear` if in Linux and which still keeps our session running

Exporting sql queries to a .csv file

```
test=# SELECT * FROM person

test-# LEFT JOIN car ON car.id= person.car_id;

id first_name | last_name I gender I email I date_of_birth I country_of_birth
| car_id | id | make I price

2 | Omar | Male 1 | Beardon | Female | fernandab@is.gd | 1921-04-03 I Finland
| 1 | 1 | Land Rover

| Sterling | 87665.38

|

| Colmore

| 1953-10-28

| Comoros

21 21 GMC

I model

Acadia

| 17662.69

1 | Fernanda

3 | Adriana

| Matuschek | Female | amatuschek2@feedburner.com

| 1965-02-28
| Cameroon
(3 rows)
```

A command to to use when we are having trouble with the PSQL commands, we can use the \? Command to see all the possible commands we can use to execute a task in the PSQL shell or command line.

test=# \?

When command is used the other following commands are shown in order to be further understood by a developer

1. General
   a. \copyright show PostgreSQL usage and distribution terms
   b. \crosstabview [COLUMNS] execute query and display results in crosstab
   c. \errverbose show most recent error message at maximum verbosity
   d. \g [(OPTIONS)] [FILE] execute query (and send results to file or |pipe);
   e. \g with no arguments is equivalent to a semicolon
   f. \gdesc describe result of query, without executing it
   g. \gexec execute query, then execute each value in its result
   h. \gset [PREFIX] execute query and store results in psql variables
   i. \gx [(OPTIONS)] [FILE] as \g, but forces expanded output mode
   j. \q quit psql
   k. \watch [SEC] execute query every SEC seconds
2. Help
   a. \? [commands] show help on backslash commands
   b. \? options show help on psql command-line options
   c. \? variables show help on special variables
   d. \h [NAME] help on syntax of SQL commands, * for all commands

Query Buffer

\e [FILE] [LINE]   edit the query buffer (or file) with external editor

\ef [FUNCNAME [LINE]] edit function definition with external editor

\ev [VIEWNAME [LINE]] edit view definition with external editor

\p   show the contents of the query buffer

\r   reset (clear) the query buffer

\w FILE   write query buffer to file

Input/Output

\copy ...   perform SQL COPY with data stream to the client host

\echo [-n] [STRING]   write string to standard output (-n for no newline)

\i FILE   execute commands from file

\ir FILE   as \i, but relative to location of current script

\o [FILE]   send all query results to file or |pipe

\qecho [-n] [STRING]   write string to \o output stream (-n for no newline)

```
  \warn [-n] [STRING]    write string to standard error (-n for no newline)

Conditional
  \if EXPR               begin conditional block
  \elif EXPR             alternative within current conditional block
  \else                  final alternative within current conditional block
  \endif                 end conditional block

Informational
  (options: S = show system objects, + = additional detail)
  \d[S+]                 list tables, views, and sequences
  \d[S+]  NAME           describe table, view, sequence, or index
  \da[S]  [PATTERN]      list aggregates
  \dA[+]  [PATTERN]      list access methods
  \dAc[+] [AMPTRN [TYPEPTRN]]  list operator classes
  \dAf[+] [AMPTRN [TYPEPTRN]]  list operator families
  \dAo[+] [AMPTRN [OPFPTRN]]   list operators of operator families
  \dAp[+] [AMPTRN [OPFPTRN]]   list support functions of operator families
  \db[+]  [PATTERN]      list tablespaces
  \dc[S+] [PATTERN]      list conversions
  \dC[+]  [PATTERN]      list casts
  \dd[S]  [PATTERN]      show object descriptions not displayed elsewhere
  \dD[S+] [PATTERN]      list domains
  \ddp    [PATTERN]      list default privileges
  \dE[S+] [PATTERN]      list foreign tables
  \des[+] [PATTERN]      list foreign servers
  \det[+] [PATTERN]      list foreign tables
  \deu[+] [PATTERN]      list user mappings
  \dew[+] [PATTERN]      list foreign-data wrappers
  \df[anptw][S+] [FUNCPTRN [TYPEPTRN ...]]
                         list [only agg/normal/procedure/trigger/window] functions
  \dF[+]  [PATTERN]      list text search configurations
  \dFd[+] [PATTERN]      list text search dictionaries
  \dFp[+] [PATTERN]      list text search parsers
  \dFt[+] [PATTERN]      list text search templates
  \dg[S+] [PATTERN]      list roles
  \di[S+] [PATTERN]      list indexes
  \dl                    list large objects, same as \lo_list
  \dL[S+] [PATTERN]      list procedural languages
  \dm[S+] [PATTERN]      list materialized views
  \dn[S+] [PATTERN]      list schemas
  \do[S+] [OPPTRN [TYPEPTRN [TYPEPTRN]]]
                         list operators
  \dO[S+] [PATTERN]      list collations
  \dp     [PATTERN]      list table, view, and sequence access privileges
  \dP[itn+] [PATTERN]    list [only index/table] partitioned relations [n=nested]
  \drds [ROLEPTRN [DBPTRN]] list per-database role settings
  \dRp[+] [PATTERN]      list replication publications
  \dRs[+] [PATTERN]      list replication subscriptions
  \ds[S+] [PATTERN]      list sequences
  \dt[S+] [PATTERN]      list tables
  \dT[S+] [PATTERN]      list data types
  \du[S+] [PATTERN]      list roles
  \dv[S+] [PATTERN]      list views
  \dx[+]  [PATTERN]      list extensions
  \dX     [PATTERN]      list extended statistics
  \dy[+]  [PATTERN]      list event triggers
  \l[+]   [PATTERN]      list databases
  \sf[+]  FUNCNAME       show a function's definition
  \sv[+]  VIEWNAME       show a view's definition
  \z      [PATTERN]      same as \dp

Formatting
  \a                     toggle between unaligned and aligned output mode
  \C [STRING]            set table title, or unset if none
  \f [STRING]            show or set field separator for unaligned query output
  \H                     toggle HTML output mode (currently off)
  \pset [NAME [VALUE]]   set table output option
                         (border|columns|csv_fieldsep|expanded|fieldsep|
                         fieldsep_zero|footer|format|linestyle|null|
                       numericlocale|pager|pager_min_lines|recordsep|
                       recordsep_zero|tableattr|title|tuples_only|
                       unicode_border_linestyle|unicode_column_linestyle|
                       unicode_header_linestyle)
  \t [on|off]            show only rows (currently off)
  \T [STRING]            set HTML <table> tag attributes, or unset if none
  \x [on|off|auto]       toggle expanded output (currently off)

Connection
  \c[onnect] {[DBNAME|- USER|- HOST|- PORT|-] | conninfo}
                       connect to new database (currently "postgres")
  \conninfo              display information about current connection
  \encoding [ENCODING]   show or set client encoding
  \password [USERNAME]   securely change the password for a user

Operating System
  \cd [DIR]              change the current working directory
  \setenv NAME [VALUE]   set or unset environment variable
  \timing [on|off]       toggle timing of commands (currently off)
  \! [COMMAND]           execute command in shell or start interactive shell

Variables
  \prompt [TEXT] NAME    prompt user to set internal variable
  \set [NAME [VALUE]]    set internal variable, or list all if no parameters
  \unset NAME            unset (delete) internal variable

Large Objects
  \lo_export LOBOID FILE
  \lo_import FILE [COMMENT]
  \lo_list
  \lo_unlink LOBOID      large object operations
```

```
test=# \copy (SELECT * FROM person LEFT JOIN car ON car.id = person.car_id) TO '/Users/amigoscode/Desktop' DELIMITER ',' CSV HEADER;
/Users/amigoscode/Desktop: Is a directory
```

```
test=#
```

Running a .sql file from psql command line

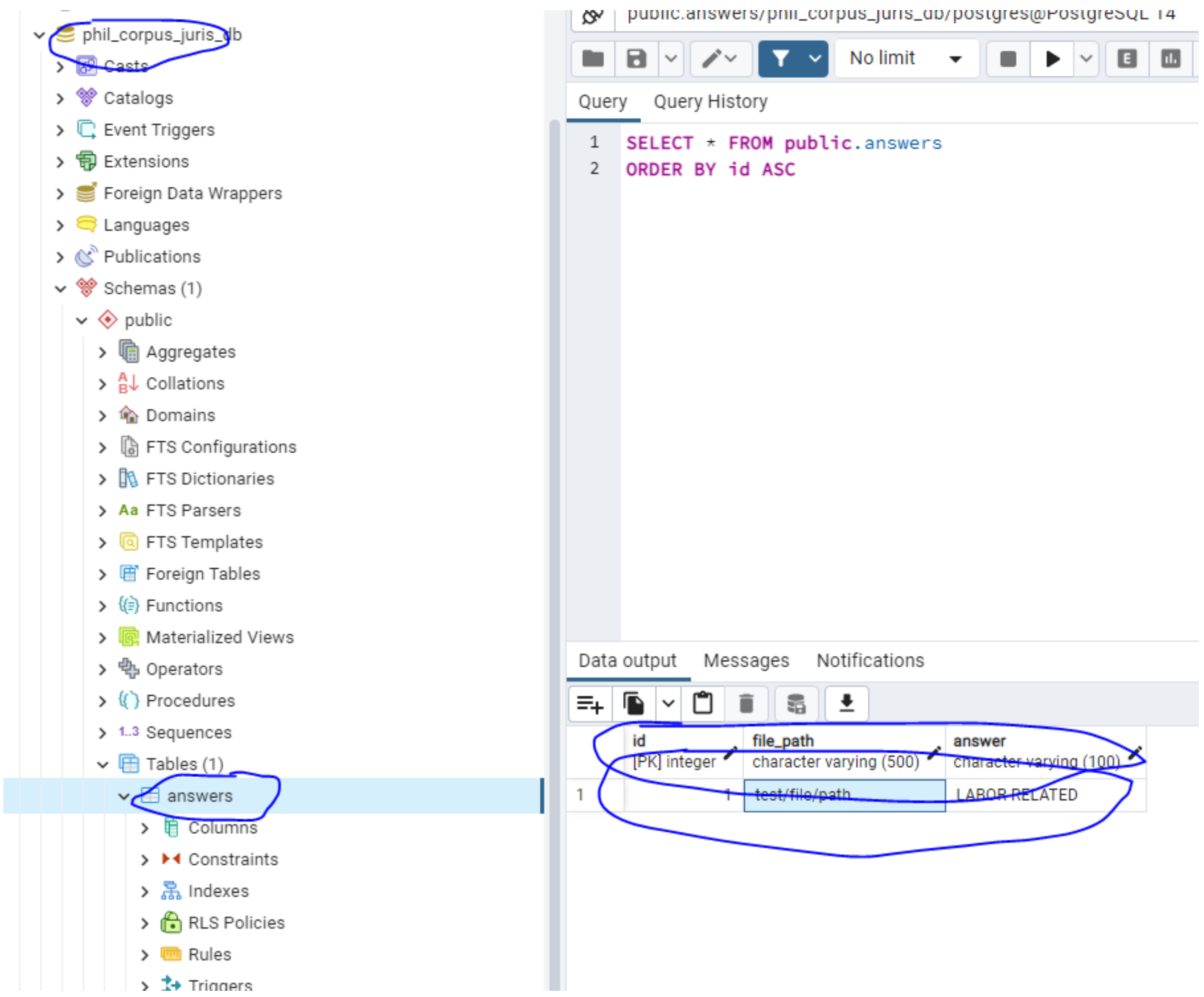
Suppose we had a file named load_labor_jurisprudence.sql

```
CREATE DATABASE phil_corpus_juris_db;


\c phil_corpus_juris_db


CREATE TABLE answers (
    id SERIAL PRIMARY KEY,
    file_path VARCHAR(500),
    answer VARCHAR(100)
);


INSERT INTO answers (id, file_path, answer)
VALUES (1, 'test/file/path', 'LABOR RELATED');
```

Note that when running the file once we've entered into our PostgresSQL command line environment, one has to know prior whether the directory currently in before entering the environment contains the .sql file we want to run in our environment. Assuming this is true we can use instead the command \ir "./<name of file>.sql" if there are spaces in the relative path or \ir ./<name of file>.sql if there are no spaces e.g. \ir ./create_db.sql to access and run the file relative to our directory we are currently in. But alternative to this should the situation arise is just using the \i "<absoloute path of file>/<name of file>.sql" to access and run a file regardless of what directory we are currently in using the absolute path of the file.

```
C:\Users\Mig\Desktop\projects\To Github\etl-phil-corpus-juris>psql -U postgres
Password for user postgres:
psql (14.4)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.


postgres=# \ir ./load_labor_jurisprudence.sql
CREATE DATABASE
You are now connected to database "phil_corpus_juris_db" as user "postgres".
CREATE TABLE
INSERT 0 1
phil_corpus_juris_db=#
```

Once we have indeed run our file the result when opened in the PostgreSQL GUI will now be as follows based on the .sql script we had run



```
SELECT * FROM public.answers
ORDER BY id ASC
```

	id [PK] integer	file_path character varying (500)	answer character varying (100)
1	1	test/file/path	LABOR RELATED

And as we can see we had created a table answers as expected and inserted a record/row in it.

mongodb

Compiling and running program through a specific programming language:

- compiling c files – gcc -o <nameofprogram> <nameoffiletocompile>.c
- compiling c++ files – g++ -o <nameofprogram> <nameoffiletocompile>.cpp
- compiling java files – javac <nameoffile>.java
- running c/c++ files – <nameofprogram>
- running java files – java <nameoffile>
- running python files – python <nameoffile>.py
- running javascript files – node <nameoffile>.js

It is ideal to put and install all these interpreters and compilers in one storage

Deleting compilers, interpreters, and all additional packages installed

Remove first, paths added to the environment variable PATH

For c/c++ remove Mingw-w64 in PATH

For python remove scripts folder and interpreter location in PATH

For conda remove scripts folder, interpreter location, and bin folder that contain conda commands in PATH

For java remove javapath in PATH

For remove npm in PATH

remember all installation locations

conda: ProgramData

nodejs: Program Files/nodejs

mingw: Program Files/mingw-w64

python: Users/AppData/Local/Programs/Python

now when you remember all these installation locations delete all these folders

then uninstall python, node, conda/anaconda, java, and mingw

open conda shell and type the command where conda to show you the list of paths that you can add to the environment variables to access the 'conda' commands

```
(base) C:\Users\Cueva>where conda

C:\ProgramData\Anaconda3\Library\bin\conda.bat

C:\ProgramData\Anaconda3\Scripts\conda.exe

C:\ProgramData\Anaconda3\condabin\conda.bat
```

Some installers directly add the path to the environment variables like python and nodejs but for others like mingw and conda we need to find the location of the bin folder, copy this location and add it to our environment variables which will result in the following

```
C:\Users\Cueva\AppData\Local\Programs\Python\Python38-32\Scripts\;

C:\Users\Cueva\AppData\Local\Programs\Python\Python38-32\;

C:\Program Files (x86)\Common Files\Oracle\Java\javapath;

C:\Windows\system32;

C:\Windows;

C:\Windows\System32\Wbem;
```

C:\Windows\System32\WindowsPowerShell\v1.0\;

C:\Program Files\Common Files\Autodesk Shared\;

C:\Program Files\dotnet\;

C:\Program Files\Microsoft SQL Server\130\Tools\Binn\;

C:\Program Files (x86)\Common Files\Oracle\Java\javapath;

C:\Windows\system32;

C:\Windows;

C:\Windows\System32\Wbem;

C:\Windows\System32\WindowsPowerShell\v1.0\;

C:\Program Files\Common Files\Autodesk Shared\;

C:\Program Files\dotnet\;

C:\Program Files\Microsoft SQL Server\130\Tools\Binn\;

C:\Users\Cueva\AppData\Local\Programs\Microsoft VS Code\bin;

E:\Projects\sass\dart-sass;

C:\Users\Cueva\AppData\Roaming\npm

Installing necessary packages

pip install -r requirements.txt or conda install -r requirements.txt

to create these dependency file without typing we must be in our environment and simply run the pip freeze > requirements .txt

asgiref @ file:///home/conda/feedstock_root/build_artifacts/asgiref_1625395912402/work

certifi==2021.10.8

cffi @ file:///D:/bld/cffi_1636046306023/work

cryptography @ file:///D:/bld/cryptography_1649035372035/work

Django @ file:///tmp/build/80754af9/django_1601563330930/work

django-cors-headers @ file:///home/conda/feedstock_root/build_artifacts/django-cors-headers_1638740536582/work

djangorestframework @ file:///home/conda/feedstock_root/build_artifacts/djangorestframework_1639677796037/work

djangorestframework-jwt==1.11.0

olefile==0.46

Pillow @ file:///C:/ci/pillow_1602782968170/work

psycopg2 @ file:///C:/ci/psycopg2_1594305209841/work

pycparser @ file:///home/conda/feedstock_root/build_artifacts/pycparser_1636257122734/work

PyJWT==1.7.1

pytz @ file:///home/conda/feedstock_root/build_artifacts/pytz_1633452062248/work

sqlparse @ file:///home/conda/feedstock_root/build_artifacts/sqlparse_1631317292236/work

typing_extensions @ file:///home/conda/feedstock_root/build_artifacts/typing_extensions_1638334978229/work

wincertstore==0.2

in the above example, these are the packages that were outputted when the command pip freeze > requirements.txt was ran, and to be able to run this in another computer the user must also have his environment set up such that when he installs these packages via this requirements.txt file he is in his designated environment where he will use the application/web application that came from let's say another developer

pending:

django

djangorestframework

django-cors-headers

djangorestframework-jwt

psycopg2

sometimes these packages do not work if interpreter is not updated

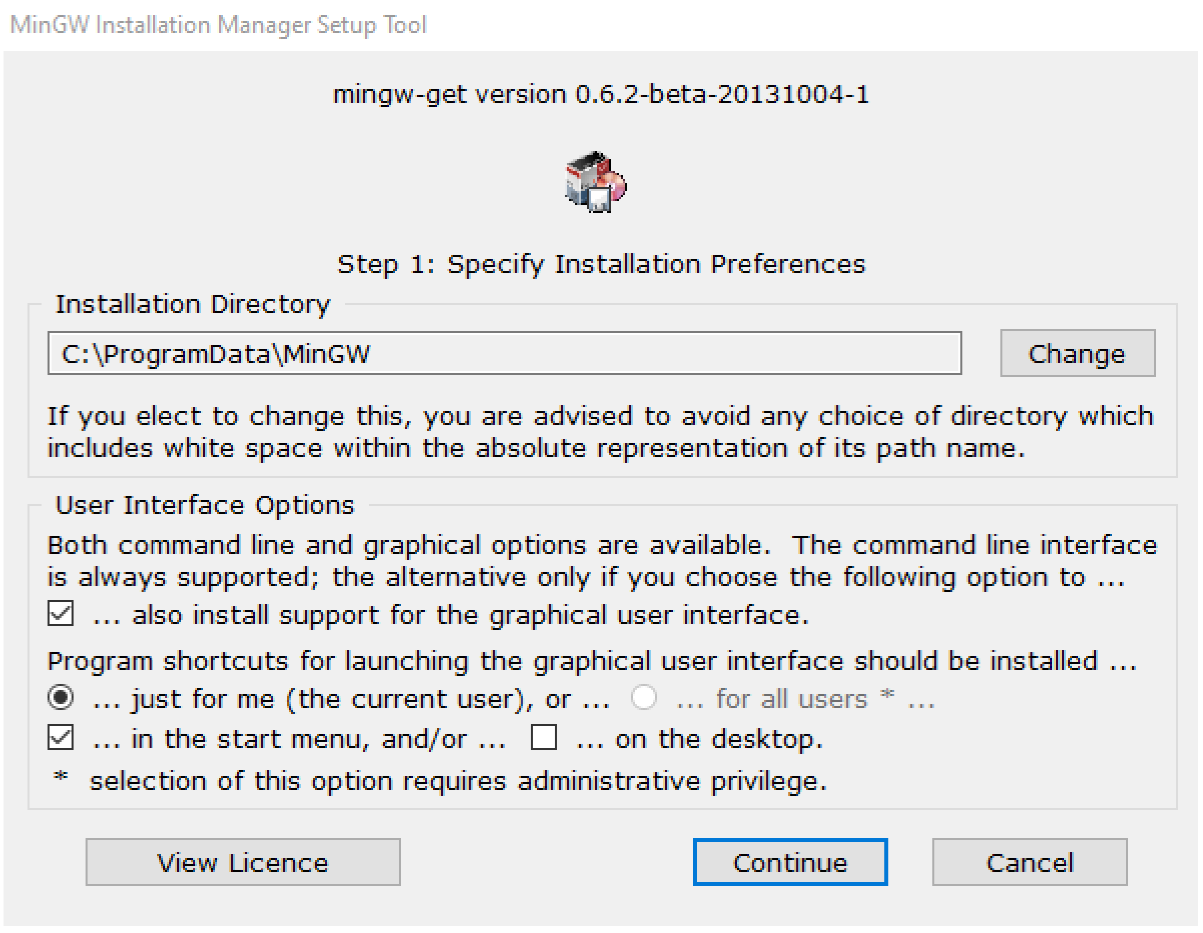
that is why you should pick an interpreter version that can accommodate versions of packages within it's range

Setting up C/C++ environment

For using the C/C++ compiler and its associated commands like g++/gcc we need to add the path of the inside directory of the bin folder which appears or is installed when we download and install the MinGW compiler. Note that not all MinGW downloadables contain the bin folder that we need to get inside to in which to copy its path and paste in our Path environment variables, so make sure that the MinGW compiler we download contains this bin folder so we can easily access it and add the its path to said Path environment variables

A link to a MinGW downloadable that contains a setup.exe file to run and install a MinGW package manager that will install in turn a set of compilers for c and C++ therefore containing the bin folder for our Path environment variable is available here https://sourceforge.net/projects/mingw/.

Once we've downloaded the MinGW installation manager setup tool from that aforementioned link we can just run the mingw-get-setup.exe file which will show the dialog box below


MinGW Installation Manager Setup Tool

mingw-get version 0.6.2-beta-20131004-1

Step 1: Specify Installation Preferences

Installation Directory

C:\ProgramData\MinGW

If you elect to change this, you are advised to avoid any choice of directory which includes white space within the absolute representation of its path name.

User Interface Options

Both command line and graphical options are available. The command line interface is always supported; the alternative only if you choose the following option to ...

[✓] ... also install support for the graphical user interface.

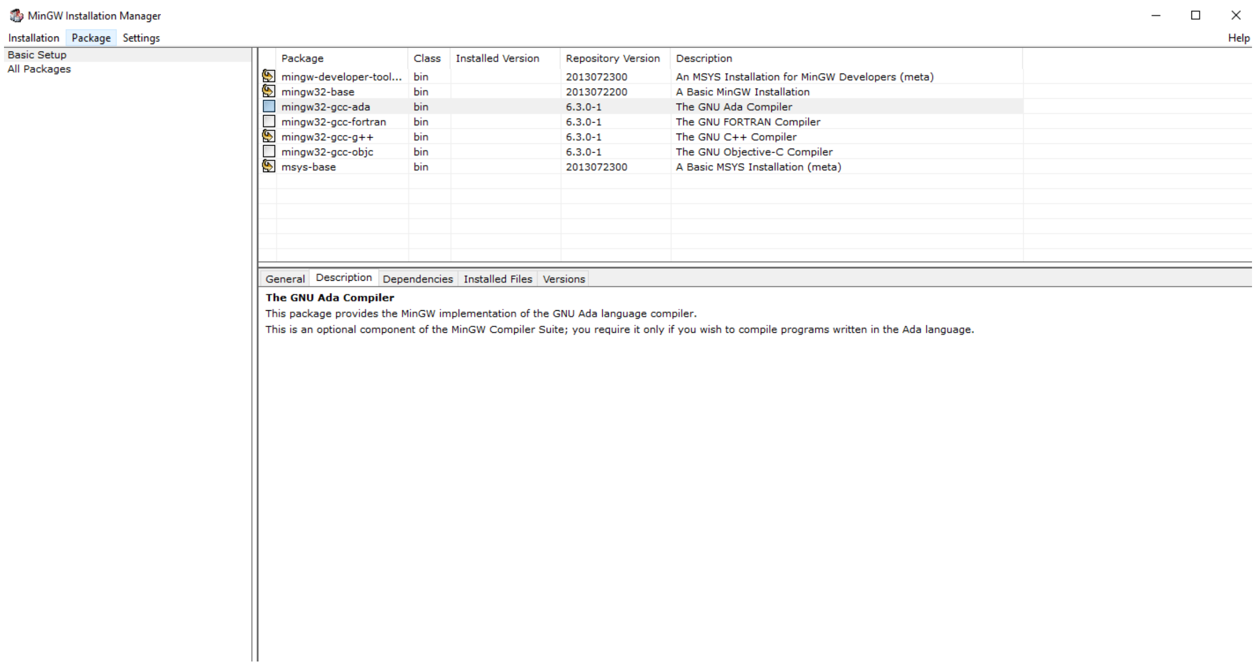
Program shortcuts for launching the graphical user interface should be installed ...

[✓ ... just for me (the current user)], or ... ... for all users * ...

[✓] ... in the start menu, and/or ... ☐ ... on the desktop.

* selection of this option requires administrative privilege.

Here we just basically install the MinGW Installation Manager (or for more understanding a package manager for external libraries that we can use akin to Conda for python) in our desired directory. We can also just uncheck the on `the desktop` option so we make sure a distracting shortcut isn't there. Once done this will install the MinGW Installation Manager in our desired directory and add a shortcut to our start menu (because of the checked `in the start menu` option). Once we click continue this will install the application and take us to the window below.



Package	Class	Installed Version	Repository Version	Description
mingw-developer-tool...	bin		2013072300	An MSYS Installation for MinGW Developers (meta)
mingw32-base	bin		2013072200	A Basic MinGW Installation
mingw32-gcc-ada	bin		6.3.0-1	The GNU Ada Compiler
mingw32-gcc-fortran	bin		6.3.0-1	The GNU FORTRAN Compiler
mingw32-gcc-g++	bin		6.3.0-1	The GNU C++ Compiler
mingw32-gcc-objc	bin		6.3.0-1	The GNU Objective-C Compiler
msys-base	bin		2013072300	A Basic MSYS Installation (meta)

The GNU Ada Compiler
This package provides the MinGW implementation of the GNU Ada language compiler.
This is an optional component of the MinGW Compiler Suite; you require it only if you wish to compile programs written in the Ada language.

Here we now basically have a package manager akin to conda only this time we aren't confined to the command line interface or the command prompt but the MinGW Installation Manager actually has an interface that we can interact with to install the necessary libraries and compilers that we want. In this case because we are setting up our environment for C/C++ which entails compiling our code we will need to install the compilers needed to compile our C/C++ source code. In this window we follow the > basic setup click path and click all the necessary packages we would need to compile our source code. Because MinGW offers packages that compile not only C/C++ but also compilers for programming languages like Ada and Fortran which we basically don't need we can just unmark/uncheck these packages in this basic setup tab and check all the other packages we need to install like obviously the mingw-developer-tools, mingw32-base, msys-base, and finally the mingw32-gcc-g++ package which in the description prvides the MinGW implementation of the GNU C++ (and C) language compiler, which we will obviously need to compile our code into a runnable application.

And that's it now we've selected the necessary packages needed for our C/C++ development environment the next step is now just installing them, and to do that we just need to follow the click path Installation > Apply Changes > Apply, and this will install all our selected packages.

Now to access all C/C++ related commands we go to the installation directory again of the MinGW Installation Manager and there we will find a bin folder we need to access. Once in the folder we copy the path and finally add it in our Path environment variable in our system properties applications environment variables. And that's it we just type the command g++/gcc –v in our command line interface or terminal to know if our compiler is working, and what you will see is it will display the current version of our compiler meaning oru compiler was successfully installed in our system.

Likewise we can also use Cygwin installation manager instead of MinGW installation manager. Why we use it over MinGW can be the following

MinGW is designed to let you use GNU development tools, such as GCC, to build native Windows applications. It has a minimal set of POSIX tools, just enough to support running the GNU development toolchain. But it is designed to compile applications that use Windows APIs, not POSIX APIs. Since programs compiled using MinGW tools only use Windows APIs, they can run on standard Windows, with no special DLLs or other run-time tools.

Cygwin is designed to let you take source code written to use POSIX APIs, and build them to run on Windows. The Cygwin project distributes multiple POSIX applications that have been recompiled to run on Windows. Since these applications are built to use the POSIX API, they require the user to install the Cygwin DLL. The Cygwin DLL handles translating the application's POSIX API calls into native Windows API calls. You can also use the Cygwin versions of the GNU development tools to compile other POSIX applications' source code.

In summary

GNU development tools (GCC, etc.): MinGW yes, Cygwin yes

Full set of POSIX tools (similar to a Linux distribution): MinGW no, Cygwin yes

Allows you to compile native Windows API applications: MinGW yes, Cygwin yes

Allows you to compile POSIX API applications: MinGW no, Cygwin yes

Should we use Cygwin we should only again install gcc-core, gcc-g++, make, and gdb

One thing to note also is that should we use code editors and ide's like VS Code or Code Blocks for that matter we should specify always the directory url or path in which the header files or standard header files our C/C++ source code will eventually use. In Cygwin this will be located in two places, namely in the `C:\ProgramData\Cygwin\lib\gcc\x86_64-pc-`

`cygwin\11\include`, and the `C:\ProgramData\Cygwin\lib\gcc\x86_64-pc-cygwin\11\include\c++` for C++ and `C:\ProgramData\Cygwin\usr\include` for C.

Setting up Java

For using java compiler and '`java`'/'`javac`' commands we need to add the path to where the `javapath` directory/folder is. However, we can also use the path of the `bin` folder when we install the Java Development Kit, which are demonstrated in the following steps below:

1. download Java Development Kit first
2. run the Java Development Kit installer
3. take note of the path it will be installed
4. locate the bin folder in the path where the JDK was installed
5. click the bin folder
6. copy the path
7. paste the path in the environment variable in order to use the Java compiler and commands in the command prompt or terminal

Using Coding Pack for Java installer can be another option if one is looking to develop in Java without having to install the individual requirements like an editor like Visual Studio code, and the Java Development Kit. Essentially this installer installs the latter two together with Java extensions needed for Visual Studio code should Java be written using VS code. But if the Java Development Kit is already installed and VS code is also installed essentially this installer will now only install the needed extensions in order to have the full experience of writing Java code using VS code.

```
import javax.swing.JButton;


public class MemberFunction{

    public static void main(String[] args) {

        System.out.print(String.format("arg 1: %s, arg 2: %s", args[0],
args[1]));

    }

}
```

In the example code above save it as a `.java` file, with the name more importantly being always the same as the class name which in this case is `MemberFunction` which must translate to the file name `MemberFunction.java`. In order to run this, go to the directory/folder where this file is located and run the command `javac <name of file>.java` (this is assuming the Java Development Kit bin folder is already added to our environment variable Path), and once done then you can run `java <name of file>.java <arg 1 if any> <arg 2 if any> … <arg n if any>` to run the `.java` file.

Besides .java files, files with .jar as its extension is a Java Archive file format which is typically used to aggregate many Java class files and associated metadata and resources e.g. text, images, etc. into one file to distribute application software or libraries on the Java platform. A .jar file is simple a file that contains a compressed version of .class Java files, audio files, image files, or directories which software like WinZip can be used to extract its content of a .jar file. However, instead of using these software extracting files from .jar files can be as simple as using the command `jar xf <jar file name>.jar` which creates a directory `META-INF` which we can navigate to containing potentially the `.class` java files. Moreover, running a `.jar` file is done by using the command `java -jar <jar file name>.jar` inside the directory containing the `.jar` file to be run.

WAR file formats on the other hand (Web Application Resource or Web Application ARchive) is a container for JAR files, JavaServer Pages, Java Servlets, Java classes, XML files, tag libraries, static sites (HTML and associated files), and other resources that make up an online application. A file entitled web.xml is located in the /WEB-INF directory of the WAR file when extracted, and it describes the structure of the online application. The web.xml file isn't technically essential if the online application is only providing JSP files. If the online apps utilize servlets, the servlet container looks at web.xml to figure out which servlet a URL request should go to. To extract the files from `.war` files the command `jar -xvf <war file name>.war` is used.

to learn:

arranging package and environment folders in python

arranging and installing packages using node

learning every command used by node, react, next js, react native, electron js,

configuring the json files when installing node modules

arranging the file structure of javascript projects/web projects

to use Jupyter notebook files via vs code install ipykernel

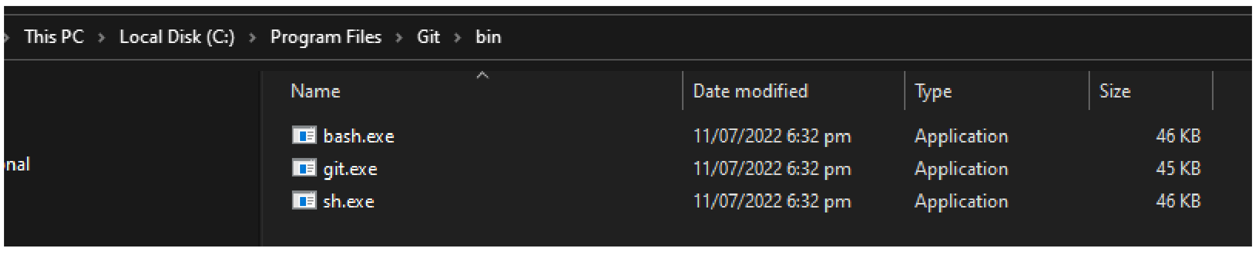
enter your environment and then enter command jupyter notebook to run the notebook in a local server, this will then output a uniform resource identifier that we can enter in our browser

Installing git

Some things to note:

The Git for Windows installer actually has a setting to set the PATH accordingly, so you can run Git from within the Windows command lines (cmd.exe, and also PowerShell, but anywhere else too). You just need to select the option when installing. So the question whether you need to do something to use Git in the windows command prompt is already done when you've selected in the installer to add Git to path

Another is to use Git in you're the windows command prompt you need to add the path of where bin folder of Git is located to the path environment variable, so you can be able to execute all other commands available to git like being able to run .sh files in the windows command prompt which has always been exclusive to merely the Linux OS.


This PC > Local Disk (C:) > Program Files > Git > bin

Name	Date modified	Type	Size
bash.exe	11/07/2022 6:32 pm	Application	46 KB
git.exe	11/07/2022 6:32 pm	Application	45 KB
sh.exe	11/07/2022 6:32 pm	Application	46 KB

WE can also operate Git Bash from the command line by typing either bash or sh since we know we've already added the bin directory's path to our path environment variable

```
D:\Projects>sh


Mig@Mig MINGW64 /d/Projects
$ exit
exit
```

Or

```
D:\Projects>bash


Mig@Mig MINGW64 /d/Projects
$
```