Package Based Development

Overview

What Is PBD?

Package based development, or PBD is a practice of working within condensed nuggets of metadata contained within related groups. You group things that go together and decide what groups depend on other groups. You can think of packages in a similar way that you think of classes in Object Oriented Programming: All the related data and functionality goes within the class itself, and if the class needs to interact with other classes, those classes have to exist first. So packages are like that, just on a larger scale: instead of being a single class, a package is typically many classes, components, permission sets, objects, fields, etc. The key here is that they are all related to each other in some way.

Why PBD?

There's a couple benefits to doing things using packages instead of a manifest or using change sets. Perhaps the biggest benefit is that once you've got the package set up and working, installation is very easy to do. Another benefit is that by using packages, you're able to use version control techniques, as creating packages requires you to create versions of those packages (which we'll talk about later). Some downsides, however, is that you need to be very conscious of what data goes into which package, and making sure that data doesn't overlap. It also requires a more defined structure, as you must know your package dependencies and responsibilities of your package.

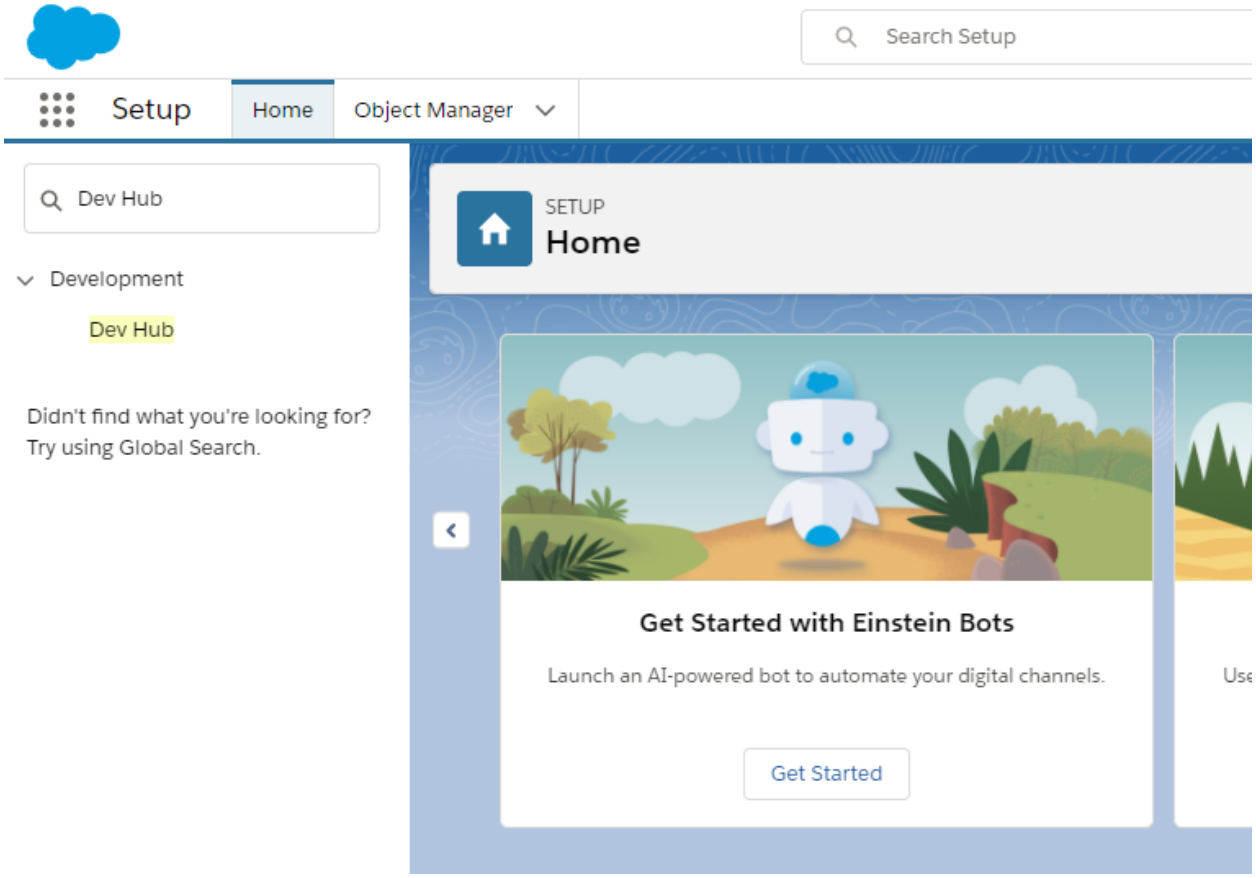
Starting Up

Before You Start

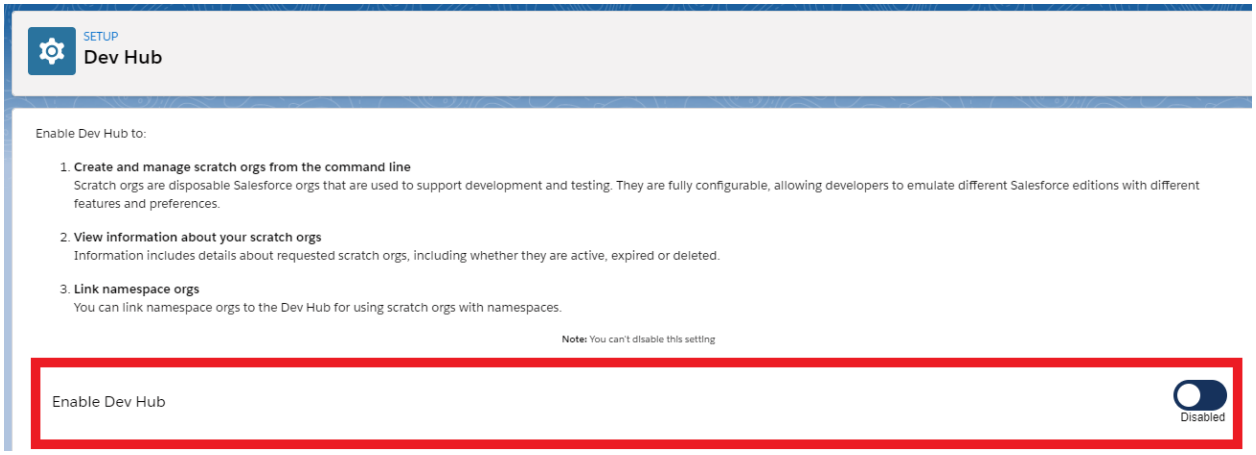
In order to begin creating a package, first you need to set up your org as a Dev Hub. You can do this even with your Trailhead orgs. The Dev Hub that your package is created off of is important, because that then becomes the only place the package can be versioned from, so if you're not all on the same org, make sure that the person versioning the package at the start is willing to accept the responsibility of always being the person to create new versions of the package.

Let’s take a look at how to do this:

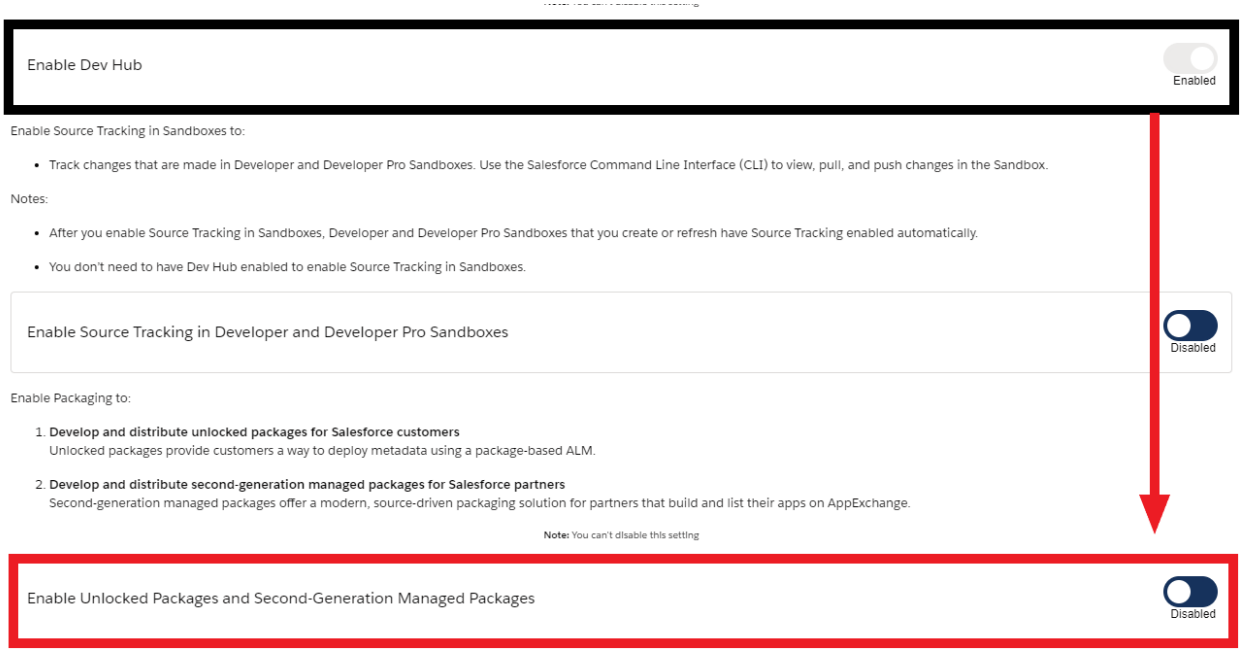
- From Setup, in the Quick Find box type in “Dev Hub” and select it.



- Once selected, click on “Enable Dev Hub”



- Once the Dev Hub is enabled, scroll down and select “Enable Unlocked and Second-Generation Managed Packages”

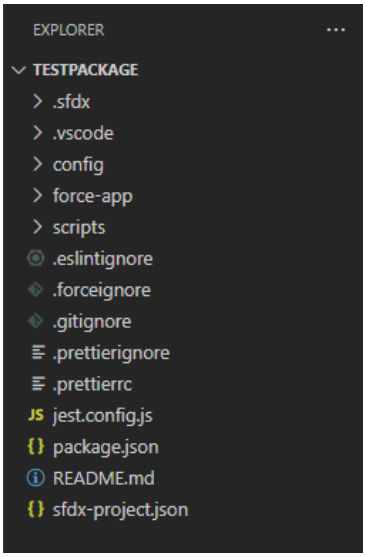


And hey would you look at that, we’re all set up here! Now let’s go into Visual Studio Code and set up our SFDX project.

Visual Studio Code Setup

First things first, we need to set up an SFDX project. So let’s do that:

- Go to the Command Palette by selecting View > Command Palette or by pressing `CTRL + SHIFT + p`
- Type in “`SFDX: Create Project`”
- Select the standard project template
- Give the project a name
- Your Explorer should look like this:



In my case, I’ve named this project as TestPackage, so you can see that was the folder that’s been created for me.

Now that we have the project set up, we need to authorize that org we set up as a Dev Hub in this project.

In order to do that, let's take these steps:

- Go to the Command Palette by selecting View > Command Palette or by pressing `CTRL + SHIFT + p`
- Type in "`SFDX: Authorize a Dev Hub`"
- If your sfdx-project.json doesn't have `sfdcLoginUrl` customized, input the login page for your Dev Hub org, just like authorizing an org. If it is set, it should automatically go to the login page for you to authorize.
- Once you're on the login page, simply login to finish authorizing.

Super cool, now our project is linked to our Dev Hub! In order to work on our package though, we need to create something called a Scratch Org first.

Scratch Orgs

Scratch Orgs are simply temporary, blank orgs for development use. When I say blank, I mean blank: they're brand spanking new and have nothing but the defaults. No metadata is copied over from your org, no nothing. By default, they last for 7 days but you can adjust this to be anywhere between 1 and 30 days. These are very useful for developing your package because you're able to test out if just your package contents work without having to worry about interacting with anything currently on the org, since nothing is there. A key distinction should be made at this point, because it can probably be a little confusing:

- Your Org: This is the base org you're working off of. This is where everything will end up, you just need to develop a new feature for this. In order to develop this feature, you're using Package Based Development.
- Dev Hub: This is actually just another word for Your Org, it's the same thing. Your Org is a Dev Hub, and because it's a Dev Hub, you can create and version packages off it.
- Scratch Org: This is a temporary, development org that starts with nothing on it. This is where you're going to be writing, testing, and developing your new feature.

You can view the Scratch Orgs that are associated with your Dev Hub Org by going into the App Menu and searching for Active Scratch Orgs. You can see when they expire and can delete them early if you so choose. There is a limit to how many Scratch Orgs you can have at once, so don't go too crazy here.

Now that we know about Scratch Orgs, let's create one back in Visual Studio Code:

- Go to the Command Palette by selecting View > Command Palette or by pressing `CTRL + SHIFT + p`
- Type in "`SFDX: Create a Default Scratch Org`"
- Use `project-scratch-def.json` to define your Scratch Org (this is the default)
- Give it an alias to identify which Scratch Org it is

- Give it a lifetime, by default it lasts 7 days but can last up to 30

Once you set all those up, SFDX takes care of the rest and now you have a Scratch Org set up and ready to go!

Working With Packages

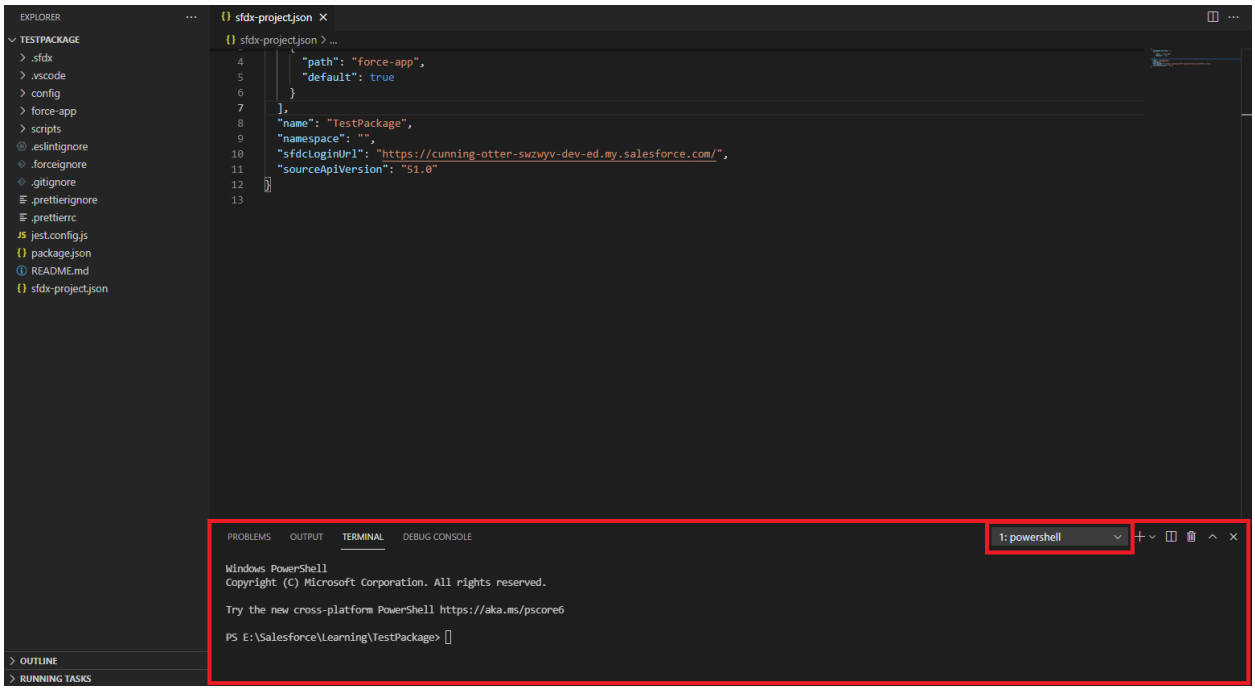
Creating a Package

In order to create a package, it's actually very very easy. If you've gotten everything set up as outlined above, then the creation of a package is just a command line away. However, before we actually do that, we should discuss that `sfdx-project.json` thing that gets put in your project, as it's very important for PBD.

Right now, your sfdx-project.json file should look something like this. The `"name"` and `"sfdcLoginUrl"` fields will likely be different, but the important thing to look at is the `"packageDirectories"` section. Right now the only package in our project is the `"force-app"` package, which is what is created by default and should be familiar if you've worked with a manifest before. This is where, by default, all of your files get put whenever you pull via manifest and if we were to load up a Scratch Org, make some changes, and then pull those changes, they'd go into the `"force-app"` directory as well.

Since we're working in packages, we'll be defining our own, custom package to be working with. We don't need to change anything in our `sfdx-project.json` file yet, just keep in mind what it looks like so you can notice the changes that happen when we DO create a package. Speaking of creating a package, let's go ahead and do that:

- At the bottom of VS Code, you should have a tab labeled TERMINAL, go ahead and open that up and make sure that the powershell terminal is selected in the dropdown menu on the right.



  This should all be set up by default, as well as you being in the correct directory inside the terminal. As you can see above, I'm in the TestPackage directory I specified earlier when I created my project.
- Next, we need to create the folder that will become our package. Just like force-app has a folder, we need to create our own folder for our package. Name it anything you'd like, I'm going to name mine CustomPackage.
- Back in our terminal, we need to use the `SFDX force:package:create` command in order to create our package. The full command looks like this:

`SFDX force:package:create -n PackageName -t PackageType -r Path -d "Description"`

- Let's break down that command:
  - `PackageName` is the name you want to give your package. Typically this would be the same name as the folder you put it in just for clarity's sake, but it doesn't have to be the same. This is the name that gets referenced by everyone else who's trying to use your package, so make sure this is clear and descriptive.
  - `PackageType` is the kind of package you want to create. Since we're working with unlocked packages, you'll be specifying "Unlocked" here.
  - `Path` is the directory in which your package currently lies. For us, that'll be the name of the folder we just created.

    - `Description` is a brief blurb about your package in case your PackageName needs more clarification.
- For me, since my package is in the CustomPackage folder, my command is going to look like this:

`SFDX force:package:create -n CustomPackage -t Unlocked -r CustomPackage -d "A package created to practice creating packages."`

- After running that command, I get this output inside of my terminal:

```
PS E:\Salesforce\Learning\TestPackage> SFDX force:package:create -n CustomPackage -t Unlocked -r CustomPackage -d "A package created to practice creating packages."
sfdx-project.json has been updated.
Successfully created a package. 0Ho5e000000XZUzCAO
=== Ids
NAME        VALUE
----------  ------------------
Package Id  0Ho5e000000XZUzCAO
```

Excellent, everything worked and if you've been following along, we've just created our first package! Now let's take a look at what that actually did, back in our `sfdx-project.json` file:

As you can see, there's now another package set in our "`packageDirectories`", which should be the package with the name and path you specified in the command line. Also changed at the bottom, there is now a "`packageAliases`" with our package name and a

crazy looking string after it. If you look closely, this is actually the same string that our console spat out at us when it finished creating the package. The string is the ID of the package, a unique identifier that distinguishes it from all the other packages. The name of the package that we created earlier is an alias for that package, meaning that if another package needed to reference this package, you could use either the crazy string or the simple alias of the package. If that was confusing, for now don't worry about it. We'll talk more about all this when we come back and define any dependencies that our package has.

Now, those of you who are observant will say "But wait, there's nothing inside this package!" and you would be absolutely correct. While we don't need anything inside our package to actually create it, we will need something inside of it to create a version of the package. Which we'll do right now, actually.

Versioning a Package

As just mentioned, we need to have something inside of our package in order to create a version of it. So let's go ahead and do that right now:

- Go to the Command Palette by selecting View > Command Palette or by pressing `CTRL + SHIFT + p`
- Type in `"SFDX: Create Apex Class"`
    - NOTE: We can create anything here, I've simply chosen to create an Apex Class for this example. My recommendation is that any time you're creating anything within VS Code, you use the Command Palette in order to do so, as it will create what you want as well as the metadata for it.
- Give your class a name, I've decided to name mine `"MyClass"`
- Specify a path for the files to be created in. Since we've created a package, it will give you the option to create a path within your folder containing your package.
    - NOTE: In order for data to be pushed and pulled correctly, data needs to be within the correct folder within your package. For example, all classes need to be contained within a `"classes"` folder, no exceptions. While putting them within a `main\default\...` path is not necessary, it is advisable to do so as these folders will be automatically created when you pull data from your Scratch Org. Using the suggested path from the Command Palette is advisable since it will follow this structure.

Once you've done that, it will create an empty class with the name you've specified, and now we have content that we can version! You can see in the `sfdx-project.json` that there is a new alias and ID for our package to represent the new version. So what exactly is a package version?

A package version is simply a snapshot of the package at the time of versioning. When you're specifying dependencies, you'll specify a specific version of a package to list as a dependency.

Whatever is in the package when you version it will be the files that SFDX looks at when someone else specifies that package version as a dependency. For example, if my `1.0.0` version of my package doesn't contain the `MyClass` Apex Class, and some other package uses `MyClass` and lists my `1.0.0` version as a dependency, SFDX will throw errors saying that the class doesn't exist, even if MyClass exists in the `1.1.0` version of the package. Therefore, it's important to keep up with versioning your package, so that other packages can use content within your package without getting errors.

Now that we know what a package version is, let's create one:

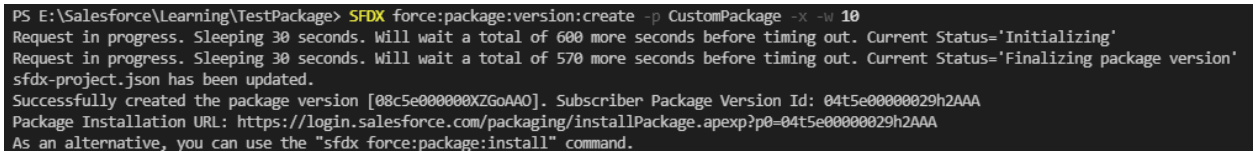
- Go back to your terminal, making sure you're still in the correct directory and using the powershell terminal.
- We'll need to use the `SFDX force:package:version:create` in order to create a version, the full command looking like this:

`SFDX force:package:version:create -p PackageName -x Password -w WaitTime`

- Again, let's break down that command:
  - `PackageName` is the name of the package we specified when we created it. Remember how I said it's important to name it something sensical? Hopefully you did, because you'll need to type that name here.
  - `Password` is a lock you set on your package if you want to restrict who can use it. If you set this, make sure you don't forget it. For most purposes, it's fine to just leave this blank.
  - `WaitTime` is a time in minutes to wait for the package to be versioned. Versioning packages can take a good amount of time, especially as the project grows so be sure to set this to be long enough so you don't time out. 10 minutes is usually a good time, just be aware that you may need to adjust it if you have a particularly complicated project.
- With all that being said, let's take a look at how my command looks like:

`SFDX force:package:version:create -p CustomPackage -x -w 10`

- Once that command is entered, my terminal looks output looks like this, and yours should be somewhat similar if you've done everything correct so far:



And with that we've successfully versioned our package! As the terminal so helpfully let us know, if we wanted to we could install this package into an org, although it wouldn't be very useful since we've not done any development on it. Another thing to note is that SFDX will verify

all dependencies before verifying the metadata of the package being versioned, so be aware that it will take longer to version a package with many dependencies. Before we get into dependencies, however, let's talk about development workflow with Scratch Orgs and packages.

Push/Pull Development Changes

To start off, let's talk about some good practices for developing using packages and Scratch Orgs.

- Scratch Orgs are, by design, temporary. This means that generally speaking, you're going to want to do as much development as possible within VS Code so that if you've set the lifetime of your Scratch Org to be too short, you don't wake up the next day with all of your development gone.
- When pulling from a Scratch Org, while you can specify what you do and don't want to push/pull, it's generally just easier to push/pull everything you've been working on. If you do your development on a Scratch Org, when you pull you might get some default metadata (like layouts and profiles) that you might not want. Another reason to develop in VS Code and push to your Scratch Org.
- Since Scratch Orgs are empty, they're not going to have any data for you when you want to see how your feature actually looks with data in it. Therefore, it's a good idea to create utility classes to create data for your Scratch Org.
- Content in one package cannot be in another package if one is dependent on the other. Therefore, whenever you are creating objects for your package and need to give permissions for those objects, you should use Permission Sets and Permission Set Groups instead of changing an existing profile.
- Since you should be doing most of your development within VS Code, now is a good time to mention you can hotkey Command Palette actions by clicking the gear icon on a command whenever you search for it within the Command Palette.

Pushing Changes

So let's say we've spent some time and written some code within our Apex Class and want to run it within our Scratch Org; let's say it's a Utility Class that creates Accounts and we're ready to try it out. All we need to do is push those changes like so:

- Go to the Command Palette by selecting View > Command Palette or by pressing `CTRL + SHIFT + p`
- Type in "`SFDX: Push Source to Default Scratch Org`"
    - NOTE: Here you'll notice that there's an option to override conflicts. Sometimes, there will be metadata in your org that conflicts with the metadata you're trying to deploy to it. Overriding the conflicts means that if there's conflicting metadata, you want to use the data within your VS Code instead of the data within your org. This commonly happens if you develop code in both the Scratch Org and in VS Code, yet another reason to only develop in VS Code.

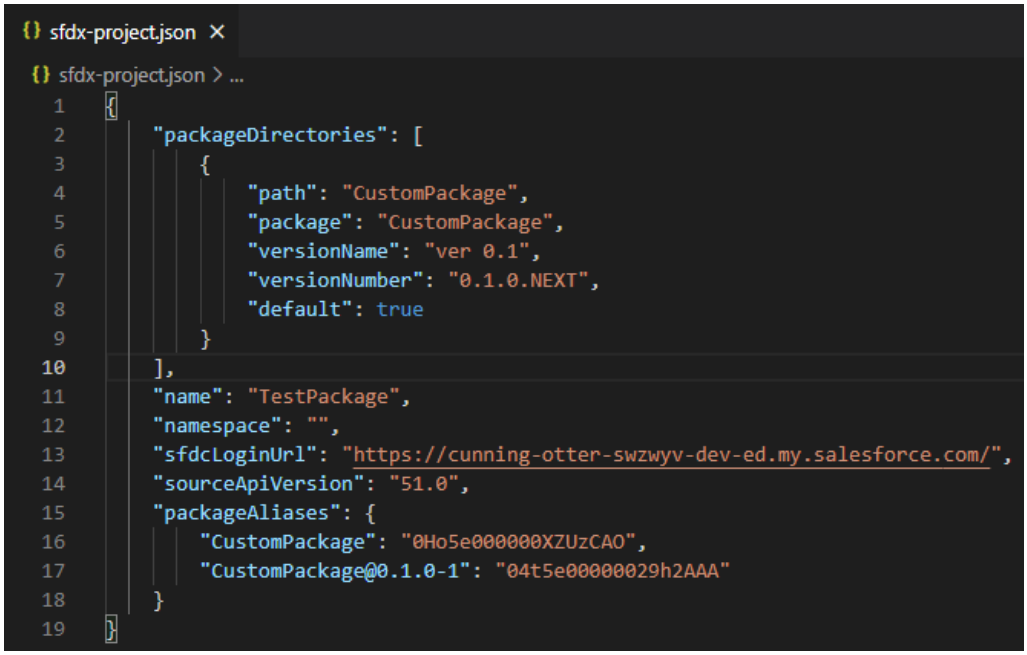
- Run the command and you should be successful as long as there are no compilation errors or metadata conflicts. If you've created a fresh Scratch Org, your first push should always be successful as long as there are no compilation errors.
    - This command is highly recommended to be hotkey'd, you'll use this a lot.
- In order to see our org, let's open it with the `"SFDX: Open Default Org"` command.

Awesome, now we're in our Scratch Org and can manipulate it just like any other org. Since we're here, let's go ahead and create a custom object. Doesn't matter what it is, I'm going to create one called `MyObject` with a custom Number field called `MyNumber`. And of course since we've created this new object, let's also create a permission set for our Admin so that we can easily include custom field access for the System Administrator profile. This is important because each time you load this data within a new Scratch Org, no profiles will have access to any Custom Fields you include in your package.

Pulling Changes

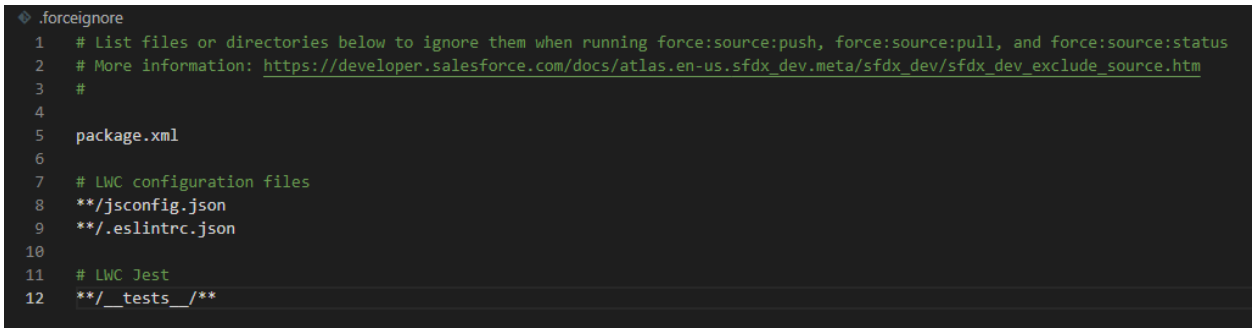
Good deal, now we've got some changes within our org and we'd like to pull those changes into our package. As mentioned before, pulling changes will actually pull the profiles as well, so we'll need to do something to make sure we're not pulling those default profiles. In addition, there's one more change we need to make to our `sfdx-project.json` for us to pull things properly.

Let's set up our sfdx-project.json first. We don't actually need the `"force-app"` package at all anymore, so we can get rid of all that. We also need to set our created package to the default package so that our pulled changes go into that package. Let's take a look at the end result:



As you can see, I've gotten rid of all that `force-app` nonsense. If you wish, at this point you can also delete the entire `force-app` folder. I've gone and done that for my project, but this is optional. I find it's better to delete what you don't need anymore. The second change you should notice is that the `"default"` field is now set to true, that way our changes will get pulled into this directory.

Once we've done that, now let's open up a file we haven't talked about before: The `.forceignore` file. The `.forceignore` file is a file that tells SFDX which files to push and pull, or more specifically, which files to IGNORE when pushing and pulling (hence forceIGNORE). Let's take a look at what is there by default:



As the comment at the top specified, you list files or entire directories in this file in order to ignore them when pushing and pulling to your Scratch Org (`force:source:push` and `force:source:pull` are the actual command line commands being run when we use the Command Palette actions). In order to specify what you want to ignore here, simply add this line:

`**path`

Where path is the folder or file reference you wish to ignore. So, in our case we want to ignore all the profiles. So we simply add this line to our `.forceignore`:

`**profiles`

Let's add it underneath line 5 just for organization purposes. This line will ignore ALL profiles when we're pushing and pulling. If we want to push/pull all profiles except one, we'll need to specify it like so:

`!**profiles\fileName`

Super easy, right? Just use the ! to include it instead! For our purposes though, we want to ignore all profiles, so just the previous line will suffice. Let's go ahead and save our changes to these files, then we can pull our changes like so:

- Go to the Command Palette by selecting View > Command Palette or by pressing `CTRL + SHIFT + p`
- Type in "`SFDX: Pull Source from Default Scratch Org`" and run the command.

And that's it, your changes should be pulled! Let's take a look at what files got pulled:

```
=== Pulled Source
STATE  FULL NAME                  TYPE           PROJECT PATH
─────  ─────────────────────────  ─────────────  ──────────────────────────────────────────────────────────────────────────────────────────────
Add    MyObject__c                CustomObject   CustomPackage\main\default\objects\MyObject__c\MyObject__c.object-meta.xml
Add    MyObject__c.MyNumber__c    CustomField    CustomPackage\main\default\objects\MyObject__c\fields\MyNumber__c.field-meta.xml
Add    MyObject__c-MyObject Layout  Layout       CustomPackage\main\default\layouts\MyObject__c-MyObject Layout.layout-meta.xml
Add    Admin_CustomPackage        PermissionSet  CustomPackage\main\default\permissionsets\Admin_CustomPackage.permissionset-meta.xml
15:57:08.405 sfdx force:source:pull
 ended with exit code 0
```

As you can see, it pulled the custom object I made, the custom field on that object, the default layout for that object and the permission set I created. You can also see that those files were added into the appropriate directories within my CustomPackage folder, since CustomPackage is labeled as the default package.

If your results include the Profiles or get placed inside the `force-app` directory, make sure you've followed the previous steps and saved your files correctly.

With all that taken care of, we now know how to create a package, version it, and work within our package! The last thing to take a look at are dependencies, so I'll be back after I version this package and create a new package with a dependency on this one.

Dependencies

That's all done, now let's take a look at the new `sfdx-project.json`:

Some stuff to point out:

- CustomPackage has a new version: `0.1.0-2`. This is the version that actually contains the `MyObject` custom object.
- There's a new package in town, this time called DependentPackage. It's got a new field called `"dependencies"`, and within it there's an array of objects that contain the dependency information. Each of these objects contains the `"package"` field, and can optionally contain the `"versionNumber"` field. I'll talk later about how that field is optional, but for right now just know that I've decided to split up the package and version declaration like so for demonstration purposes.
- Those of you with a keen eye will notice that the `"versionNumber"` is the first version of CustomPackage, which is the one that doesn't have the `MyObject` custom object.

DependentPackage contains a single Apex Class called `DependentClass` that constructs a `MyObject__c` variable called `dependentObject`. I've pushed my source to the Scratch Org, and everything pushes with no compilation errors! So I'm all set, right? Well, no, this is a common trap to fall into. Let's delve a little deeper.

When you push content to your Scratch Org, it's pushing everything across all your packages with no regard for dependencies. This means that if you reference, for example, an object within an Apex Class, as long as that object is in one of your packages within your project or already exists in your Scratch Org, there will be no compilation errors. However, as mentioned earlier, a package version is simply a screenshot of the current state of the package. So when you are versioning the package, if the package you are versioning is dependent on another package, it will take the contents of the screenshot supplied (the package version) and check to see if the data you referenced is in that screenshot. If it isn't, it'll fail to create the version. Let's see that in action by trying to version our DependentPackage package:

```
PS E:\Salesforce\Learning\TestPackage> SFDX force:package:version:create -p DependentPackage -x -w 10
Request in progress. Sleeping 30 seconds. Will wait a total of 600 more seconds before timing out. Current Status='Initializing'
ERROR running force:package:version:create:  DependentClass: Invalid type: MyObject__c
```

Uh oh, when we try to version that package, SFDX doesn't know what this `MyObject__c` thing is! That's because we specified the CustomPackage version that does not contain `MyObject` in it. In order to fix this, we need to specify the correct version, `0.1.0.2`. Let's go ahead and update that and see the results:

```
PS E:\Salesforce\Learning\TestPackage> SFDX force:package:version:create -p DependentPackage -x -w 10
Request in progress. Sleeping 30 seconds. Will wait a total of 600 more seconds before timing out. Current Status='Initializing'
Request in progress. Sleeping 30 seconds. Will wait a total of 300 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 270 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 240 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 210 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 180 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 150 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 120 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 90 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 60 more seconds before timing out. Current Status='Finalizing package version'
Request in progress. Sleeping 30 seconds. Will wait a total of 30 more seconds before timing out. Current Status='Finalizing package version'
sfdx-project.json has been updated.
Successfully created the package version [08c5e000000XZH3AAO]. Subscriber Package Version Id: 04t5e00000029yCAAQ
Package Installation URL: https://login.salesforce.com/packaging/installPackage.apexp?p0=04t5e00000029yCAAQ
As an alternative, you can use the "sfdx force:package:install" command.
```

It worked! Something to note here, this time it took almost the entire 10 minutes. Sometimes that's just how it goes, even though there's only a single class within this package.

Now let's talk about the `"versionNumber"` field.

If you have a package that is dependent on another package within THE SAME Dev Hub, you can fix issues like this one with the use of a version keyword. We've already seen one keyword, the `NEXT` keyword in the `"versionNumber"` field outside of the dependencies, within the package declaration itself that increments the number every time you create a new version. There's another keyword you can use in the dependency declaration called `LATEST`. If you specify the version number then `LATEST`, like `0.1.0.LATEST`, then SFDX will automatically

pull the latest version as a dependency. Similarly, you can specify `RELEASED`, like `0.1.0.RELEASED` to specify the latest released dependency.

If you have a package that is dependent on another package within A DIFFERENT Dev Hub, instead of specifying both `"package"` and `"versionNumber"`, you can supply the alias or ID associated with the version you want. You can do this with packages in the same Dev Hub, but typically you'll want to separate them out to use the `LATEST` keyword. For example, you could specify `"package" : "CustomPackage@0.1.0-2"` to handle the dependency information in a single line instead of two.

To wrap things up, the syntax for creating a package with two dependencies just adds in a new object to the list like so:

You can see here I'm using single line notation for this OtherPackage and I'm using the `LATEST` keyword for the package found within the same DevHub. You can do this for all your dependencies, just adding in new ones as necessary.

As a final note on dependencies, it should be mentioned that any transitive dependencies will not be calculated. That is, if `Package B` depends on `Package A`, and `Package C` depends on `Package B` and `A`, then `Package C` needs to list `Package A` and `Package B` as dependencies even though `Package A` is already listed as a dependency of `Package B`.

PBD Final Advice

With that, you know just about everything you need in order to be successful with Package Based Development. So let's wrap things up by giving some final bits of advice:

- You can find a list of SLI Commands here, these should be helpful to look at in case you want to do something a bit more specific not detailed here.
- Make sure you have your dependencies sorted out before diving into PBD, it'll help in the long run.
- Make sure you create new Scratch Orgs often. It'll help catch when you're being lazy with your package contents and relying on data that lives within your Scratch Org and not within your actual package.
- Make sure to strongly define what objects/fields go within what package, and be very careful to ONLY give permissions to objects/fields that are included in your package.
- It's totally valid to have fields for an object that doesn't exist in your package. Just make sure that the object meta is only in one package. For example, if you have a Custom Object in a Core package, it's ok to add on a new Custom Field in a later package, just make sure that the Object meta file is only in the Core package.
- If you need to give permissions for custom objects and fields you've created in your package, it's better practice to use a permission set instead of a profile.