SALESFORCE DX

Campfire Workbook

About the Workbook

About the ALM Workbook

This workbook provides an introduction to new Application Lifecycle Management (ALM) capabilities and tooling introduced with Salesforce DX. This workbook assumes that you know about Salesforce ALM best practices and a little programming. If you don't, you'll still manage to follow along, but it may be a little more difficult. It might make sense to team up with a more experienced developer or admin to get the most out of these exercises. We recommend that you've completed the Application Lifecycle and Development Models Trail and the App Development with Salesforce DX trails prior to attempting the labs.

The workbook is divided into three sections:

1. Development:
2. Deploy/Packaging:
3. Testing/CI/CD:

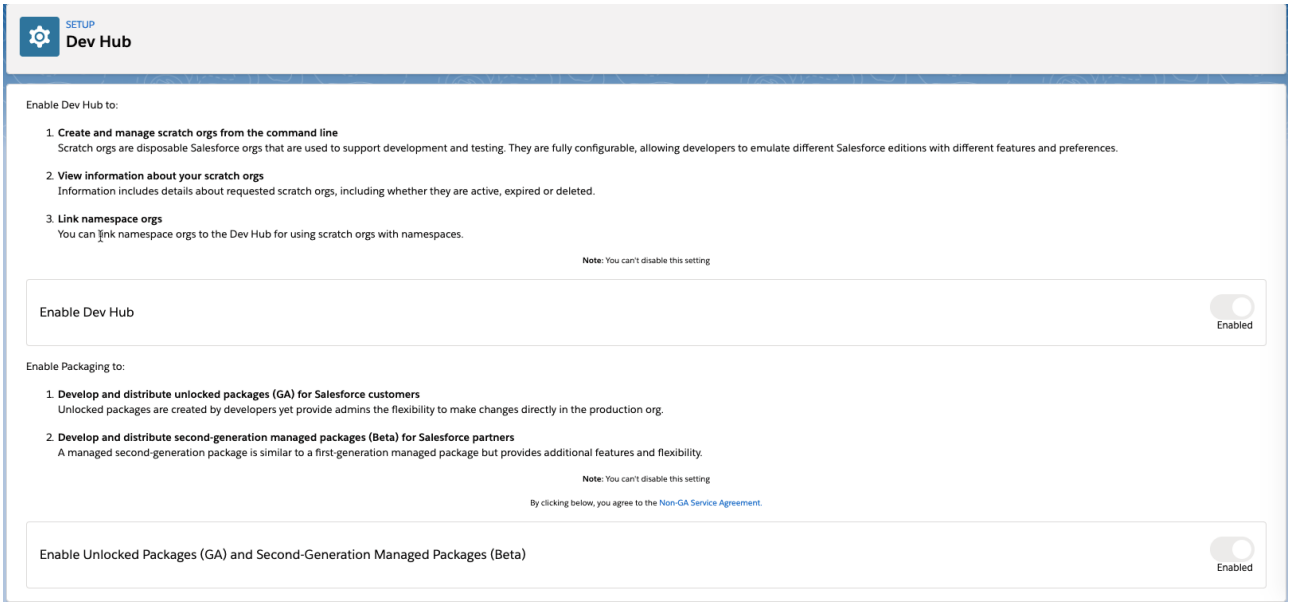
The goal of these exercises are to give you an understanding of how you can adopt new Salesforce ALM capabilities introduced with DX to augment or even reimagine the way you develop, test, and deliver applications on the Salesforce Platform with an integrated, end-to-end lifecycle designed for high-performance. While touring along, feel free to experiment a little and have fun!

Intended Audience

This workbook is intended for IT Managers, Salesforce Developers, Salesforce Administrators and Salesforce Release Managers that are interested in learning how Salesforce DX can help optimize or reimagine their Salesforce ALM processes.

Part 0 : Before you Begin

1. Sign up for a Developer Edition Org.
2. If you haven't used Git and/or GitHub before, complete the Git and GitHub Basics trail on Trailhead.
3. Make sure Git is installed on your machine`
    a. **macOS** - Git is available on Mac OS via the Xcode Command Line Tools. You can trigger an install by simply running ***git --version*** from the **Terminal** app. A macOS git installer is also available at http://git-scm.com/download/mac
    b. **Windows** - You can install git from http://git-scm.com/download/win
4. Setup Salesforce DX following the instructions at the Set Up Salesforce DX Trail . This will involve installing the Salesforce CLI and enabling your DE org as a Dev Hub.
5. Enable Unlocked Packages (GA) in your Dev Hub org by going to **Setup → DevHub**



6. The lab exercises will refer to the Visual Studio Code (VS Code) IDE for the purposes of creating new source files or updating existing ones. Install VS Code by following the instructions at Quick Start: Visual Studio Code
7. OPTIONAL - Setup TravisCI by following the instructions at the Continuous Integration Using Salesforce DX trail if you want to complete the Continuous Integration section of the labs.
NOTE: Per the Travis CI Docs, there is a report of some functions not working on a local Windows machine. Please use the WSL (Windows Subsystem for Linux) or a Linux or macOS machine.

Part 1: Development

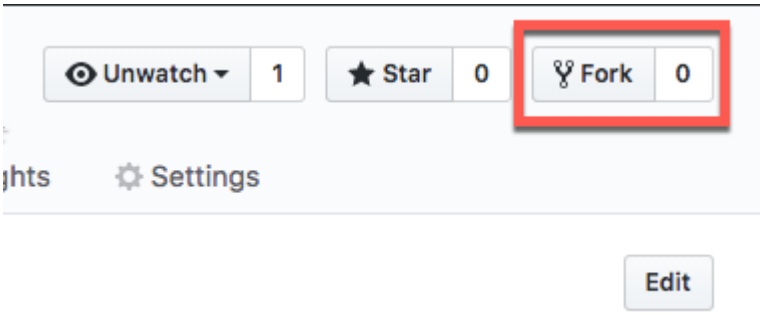
Introduction

In this first section, we'll review and expand on a few of the key features introduced in the Salesforce DX Trailhead modules. You'll go through the process of forking and cloning the Dreamhouse App git repository that will be used throughout all of the labs. Pay special attention to the flow of the exercise. The focus is on how a developer interacts with VCS and DX artifacts/projects in the context of Salesforce Development. For those of you with experience with developing on other platforms, parts of this exercise should seem familiar since Salesforce DX helps developers align with the best of modular development practices used in the industry.

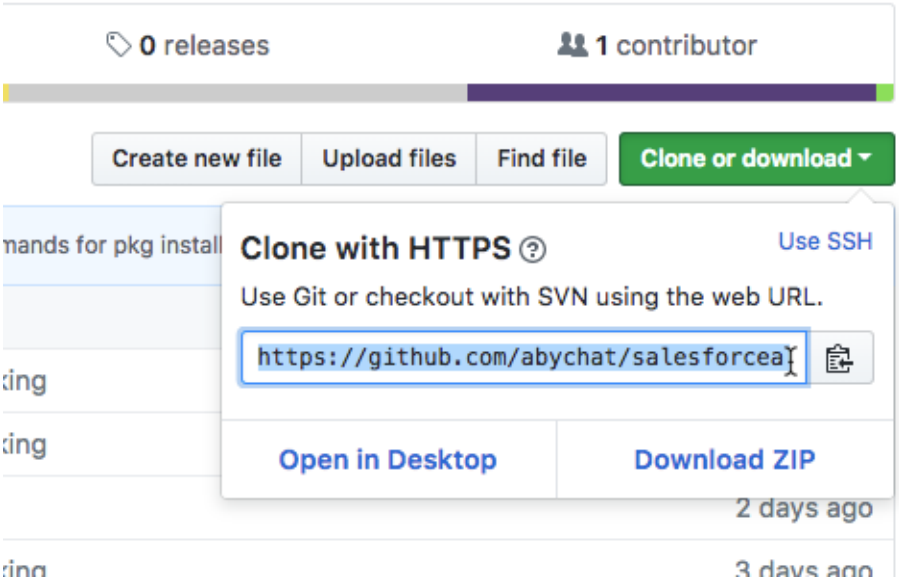
Lab Instructions

Fork project repository

1. Make sure you are signed into your GitHub account.
2. Access the **salesforcealm** repository at https://github.com/abychat/salesforcealm
3. Fork the **salesforcealm** repository by clicking the 'Fork' button located on the top right corner of the screen.



4. A 'Fork' is essentially your copy of a particular repository where you can experiment and make changes to the source without affecting the original repository.
5. Clone your newly forked repository to your local workspace.
    a. Click on the **Clone or Download** button.



b. Copy the URL that appears on the '**Clone with HTTPS**' window.

c. Go to the **Command Prompt/Terminal** and navigate to your working directory

d. Issue the following command

```
git clone <URL copied in step b>
```

e. Verify the project has been cloned to your file system.

6. Change directory to the **salesforcealm** project by running **cd salesforcealm**
7. Checkout the **labs** branch by running **git checkout labs**
8. Open the **sfdx-project.json** file. Do you notice anything different?

Create a Scratch Org and Push Source

1. If you haven't already, authenticate to your DevHub with the following command

   `sfdx force:auth:web:login -d -a DevHub`
2. Create a scratch org and designate it as the default scratch org by issuing the following command

   `sfdx force:org:create -f config/project-scratch-def.json -a WkshpScratch -s`

   Here the -a param designates an alias for the username and -s designates the default scratch org.
3. If you inspect this org, you will notice that it is an empty org except for some core object. Open and inspect the org using the following command

   `sfdx force:org:open`
4. Run the following command

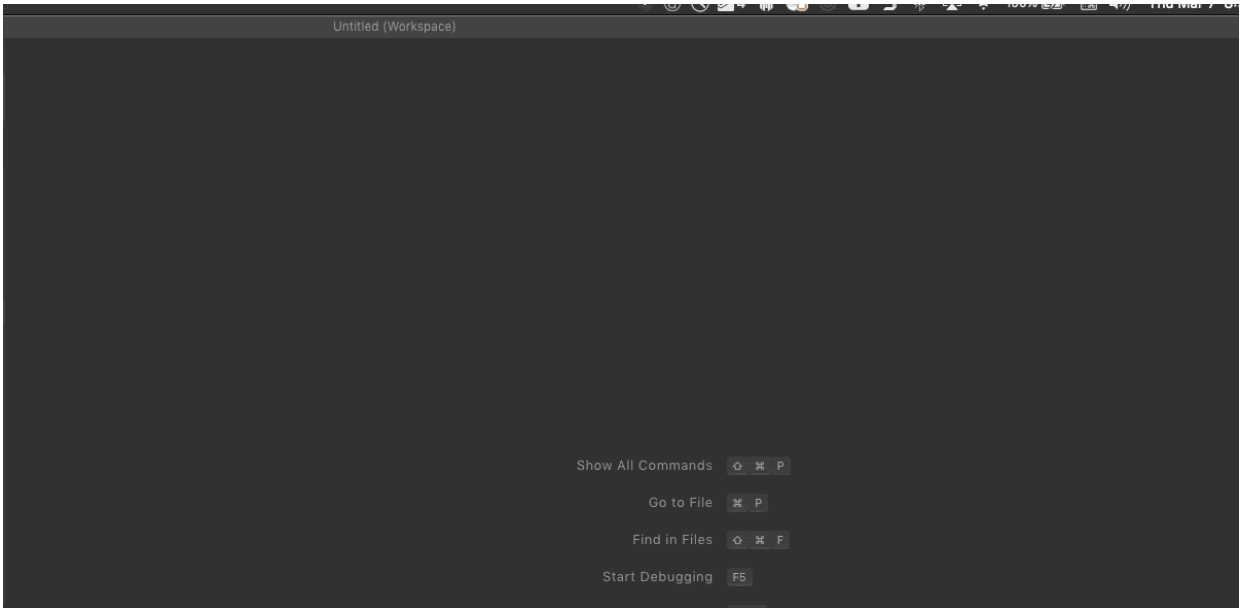
   `sfdx force:source:push`

   This will push source in all project directories/folders to the scratch org you just created.

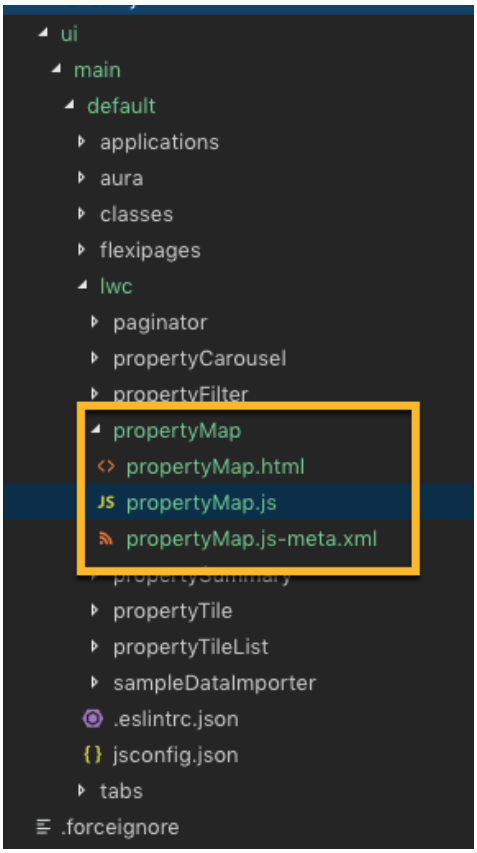
5. Through the course of working on a project you might have files and folders in your project that you might not want to track with git or push to scratch orgs. The following files will help you with this -
    a. .**gitignore** - Use the .gitignore file to specify the files that you do not want to track. If a repository doesn't have a .gitignore file you can create one with your favorite text editor or IDE. Take a look at the .gitignore file in your project.
    b. .**forceignore** - Similar to .gitignore, if you don't want the Salesforce CLI to track local or remote changes to specific files and folders, you can specify them in the .forceignore file.
6. Assign the **dreamhouse** permission set to the scratch org user by executing the following command
   `sfdx force:user:permset:assign -n dreamhouse -u WkshpScratch`
7. Run the **sfdx force:org:open** command to access your scratch org.
8. Access the **Dreamhouse** app by clicking the App Launcher.
9. You will notice there is no data to view when you view the Properties or Brokers tab.
10. Go back to the Command Prompt or Terminal and execute the following command
    `sfdx force:data:tree:import -p assets/data/Broker__c-Property__c-plan.json`
11. The previous command essentially executes a plan file to import data from multiple data files. These files can be easily created by querying Sandbox, Production orgs or other scratch orgs by using the force:data:tree:export command. The query can be written directly on the command line or can be stored in a file. For example -
    `sfdx force:data:tree:export -q assets/data/brokers.soql --plan --outputdir assets/data --targetusername username`

Create a new Lightning Web Component (LWC)

1. Go to VSCode and open your project. Skip to step 8 if you would like to use the CLI to create the LWC.
2. Open the Command Palette by pressing Ctrl+Shift+P (Windows) or Cmd+Shift+P (macOS).
3. Type in **SFDX : Create Lightning Web Component**
4. Choose the following path as the location to save the LWC - **ui/main/default/lwc**
5. When prompted provide the following name for the LWC - **propertyMap**
6. The **propertyMap.js** file will open in a new editor window



7. You should see a new folder called propertyMap created as a subfolder of the lwc folder of the ui package with the following files in it - propertyMap.html, propertyMap.js, propertyMap.js-meta.xml



8. You can also use the CLI to create a Lightning Web Component. Run the following command to create a LWC

   `sfdx force:lightning:component:create -n propertyMap --type lwc -d ui/main/default/lwc`

9. You should see the following output

```
target dir = /Users/achaturvedi/Google Drive/Work/Project Code/salesforcealm/ui/main/default/lwc
   create propertyMap/propertyMap.js
   create propertyMap/propertyMap.js-meta.xml
   create propertyMap/propertyMap.html
```

10. From the Terminal App/Command Prompt run

    `sfdx force:source:status`

11. Note that the CLI tracks changes made locally as well as remotely -

```
=== Source Status
STATE          FULL NAME                  TYPE                    PROJECT PATH
-------------  -------------------------  ----------------------  ------------------------------------------------------------------
Local Add      DreamHouse                 CustomApplication       ui/main/default/applications/DreamHouse.app-meta.xml
Local Changed  propertyMap/propertyMap.html  LightningComponentBundle  ui/main/default/lwc/propertyMap/propertyMap.html
Local Changed  propertyMap/propertyMap.js   LightningComponentBundle  ui/main/default/lwc/propertyMap/propertyMap.js
Local Changed  propertyMap/propertyMap.js   LightningComponentBundle  ui/main/default/lwc/propertyMap/propertyMap.js-meta.xml
```

12. Enable Git tracking of these files by running **git add .**
13. Open the propertyMap.js file and copy paste the code from the following URL - [propertyMap.js]
14. Open the propertyMap.html file and copy paste the code from the following URL - [propertyMap.html]
15. Open the propertyMap.js-meta.xml and copy paste the code from the following URL - [propertyMap.js-meta.xml]
16. Save all the files.
17. Push these changes to the scratch org by running **sfdx force:source:push**

```
=== Pushed Source
STATE  FULL NAME                  TYPE                    PROJECT PATH
-----  -------------------------  ----------------------  ------------------------------------------------------------
Add    propertyMap/propertyMap.html  LightningComponentBundle  ui/main/default/lwc/propertyMap/propertyMap.html
Add    propertyMap/propertyMap.js   LightningComponentBundle  ui/main/default/lwc/propertyMap/propertyMap.js
Add    propertyMap/propertyMap.js   LightningComponentBundle  ui/main/default/lwc/propertyMap/propertyMap.js-meta.xml
```

18. Open the default scratch org and access the Lightning App Builder to edit the Property Explorer Lightning page. This can be done by following one of two options (a or b)

    a. Use the Salesforce CLI

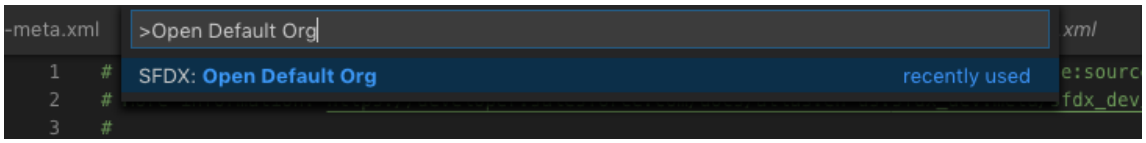
       i. Execute the following command

          `sfdx force:source:open -f ui/main/default/flexipages/Property_Explorer.flexipage-meta.xml`

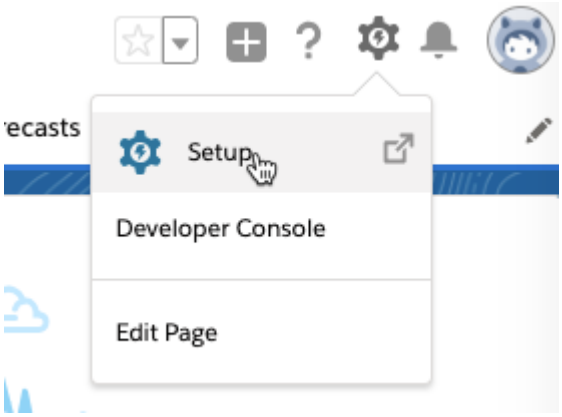
       ii. Done!

b. Open the default scratch org using the command palette in VS Code and access the Lightning App Builder

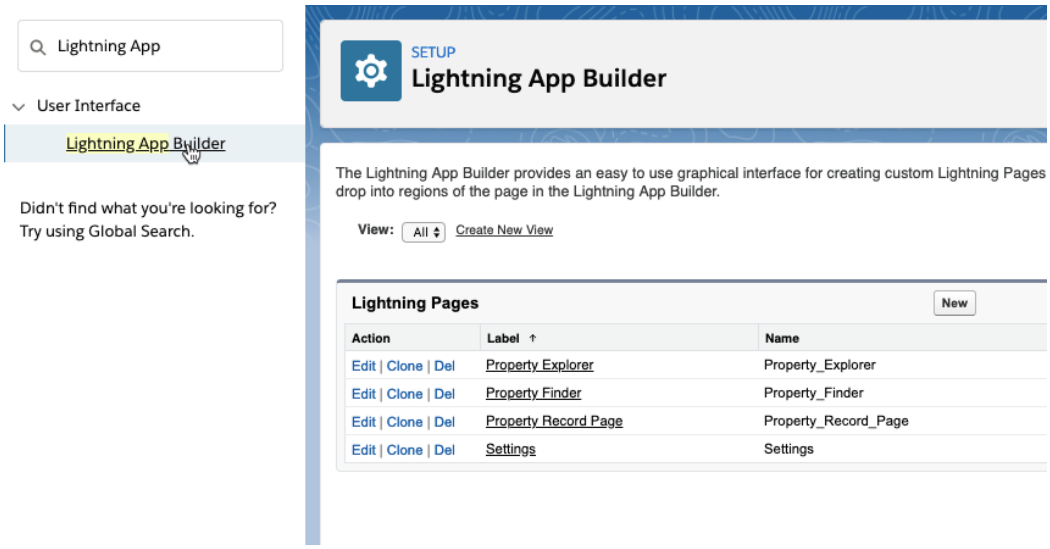
i. Open the command palette and type in 'Open Default Org'. Choose the 'SFDX: Open Default Org' option and press **Enter**.



ii. Click **Setup** on the top right corner of the screen.

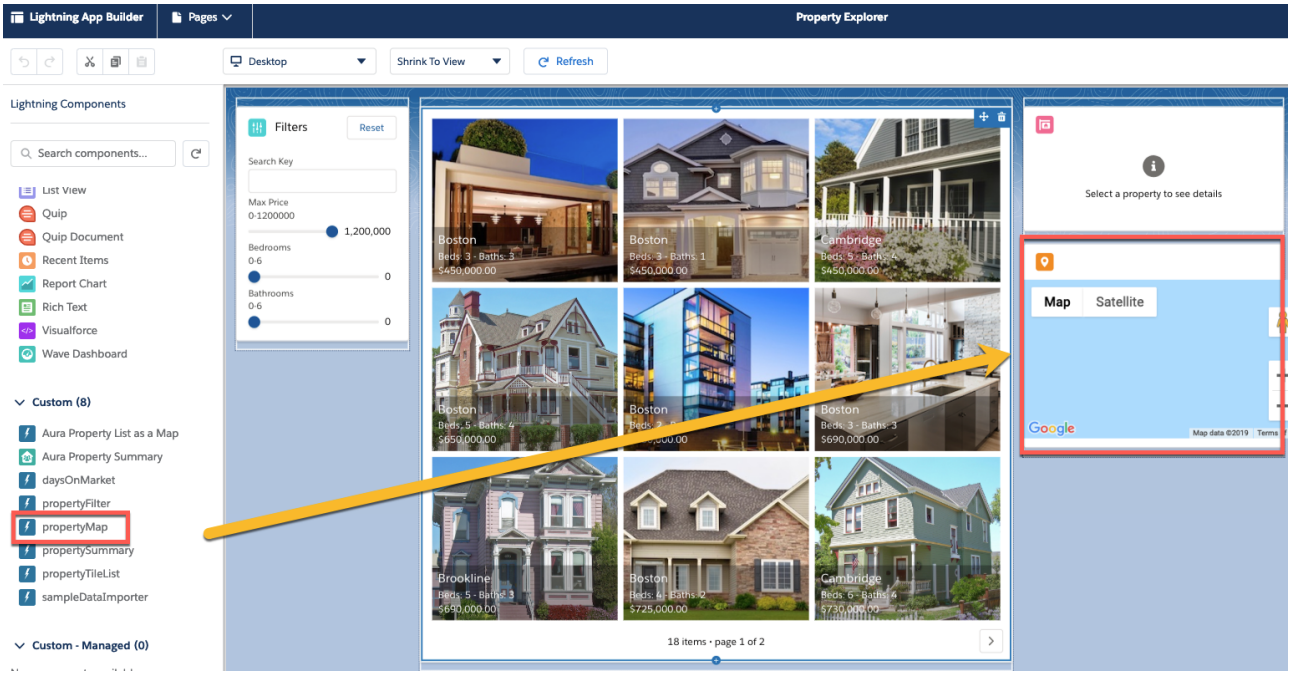


iii. Type Lightning App Builder in the Quick Search page and click on **Lightning App Builder**

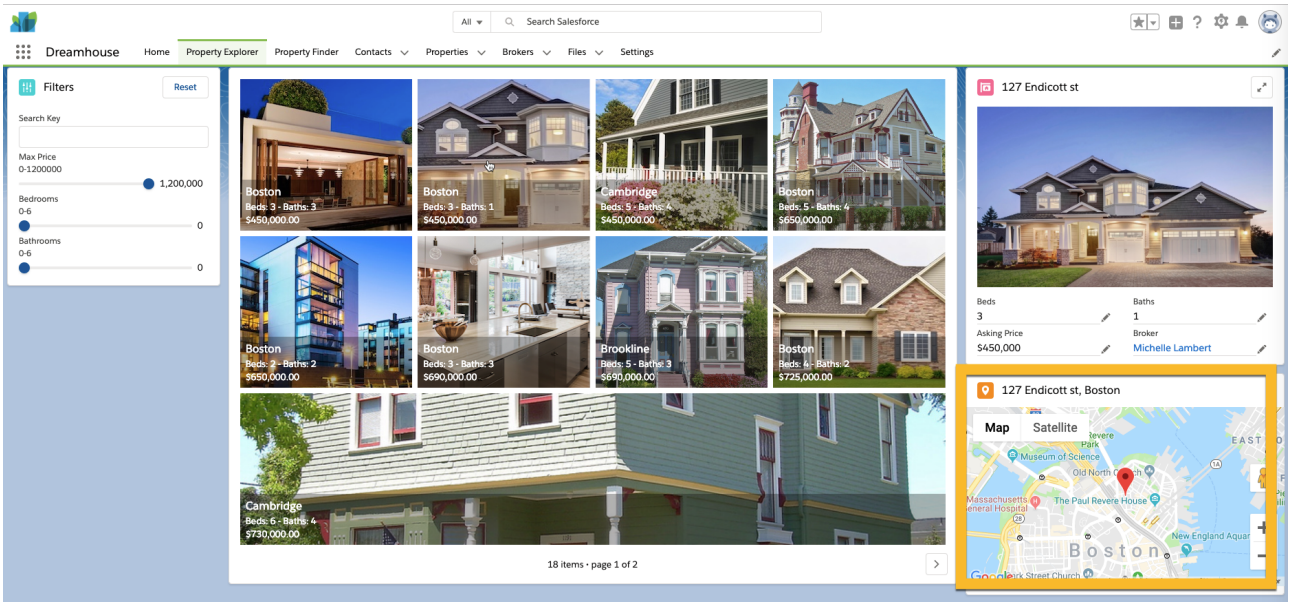


iv. Click **Edit** next to the the 'Property Explorer' page.

19. Add the ‘propertyMap’ component to the ‘Property Explorer’ page and click **Save**.



20. Launch the Dreamhouse App and access the Property Explorer tab.
21. Select a property to see the location of the property on the map.



22. From the Command Prompt/Terminal app, run the **sfdx force:source:status** command and take note of the remotely tracked changes.

```
=== Source Status
STATE           FULL NAME          TYPE       PROJECT PATH
--------------  -----------------  ---------  ------------------------------------------------------------------
Remote Changed  Property_Explorer  FlexiPage  ui/main/default/flexipages/Property_Explorer.flexipage-meta.xml
```

23. Run the following command to pull these changes from your scratch org

    ```
    sfdx force:source:pull
    ```

24. Verify these changes were pulled by running the following command to check the status of your repository

    ```
    git status .
    ```

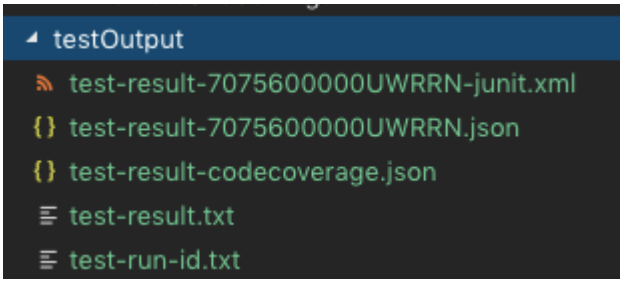
25. You can then commit this code and push it to your remote repository. Hold off on doing that just yet, we will do that in the last lab exercise to kick-off our CI/CD process.
26. This is a typical cycle of how you can create, maintain and update your Salesforce apps using Salesforce DX.

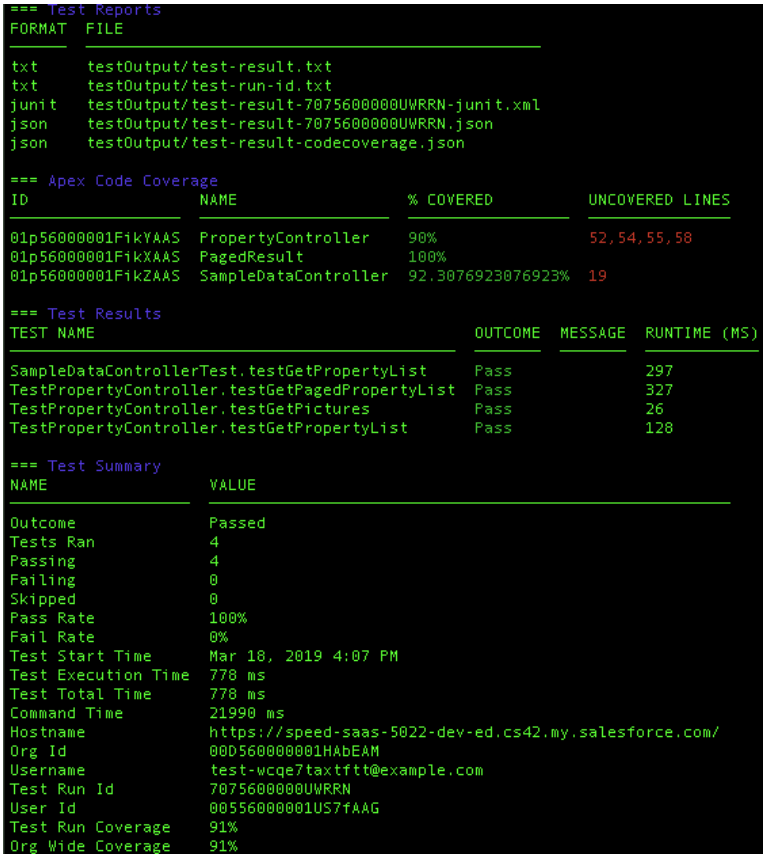
Test Your Apex Code

1. Now that you have successfully pushed your source to a scratch org, it's time to run some tests.
2. Execute the following command

   ```
   sfdx force:apex:test:run -u WkshpScratch -c -r human -w 30 -d testOutput
   ```

3. In the above command the -u flag specifies the user name, -c retrieves code coverage results, -r specifies the result format, -w the time to wait for log responses and -d specifies the directory for outputting the test files.
4. Inspect the output. Note the code coverage, test classes executed and the Test Summary. Also check VS Code to see that a folder name testOutput has been created and has test result files in multiple formats in it





```
=== Test Reports
FORMAT  FILE
------  ----------------------------------------------------
txt     testOutput/test-result.txt
txt     testOutput/test-run-id.txt
junit   testOutput/test-result-7075600000UWRRN-junit.xml
json    testOutput/test-result-7075600000UWRRN.json
json    testOutput/test-result-codecoverage.json

=== Apex Code Coverage
ID                  NAME                 % COVERED           UNCOVERED LINES
------------------  -------------------  ------------------  ---------------
01p56000001FikYAAS  PropertyController   90%                 52,54,55,58
01p56000001FikXAAS  PagedResult          100%
01p56000001FikZAAS  SampleDataController  92.3076923076923%  19

=== Test Results
TEST NAME                                          OUTCOME  MESSAGE  RUNTIME (MS)
-------------------------------------------------  -------  -------  ------------
SampleDataControllerTest.testGetPropertyList       Pass               297
TestPropertyController.testGetPagedPropertyList    Pass               327
TestPropertyController.testGetPictures             Pass               26
TestPropertyController.testGetPropertyList         Pass               128

=== Test Summary
NAME                 VALUE
-------------------  ----------------------------------------------------
Outcome              Passed
Tests Ran            4
Passing              4
Failing              0
Skipped              0
Pass Rate            100%
Fail Rate            0%
Test Start Time      Mar 18, 2019 4:07 PM
Test Execution Time  778 ms
Test Total Time      778 ms
Command Time         21990 ms
Hostname             https://speed-saas-5022-dev-ed.cs42.my.salesforce.com/
Org Id               00D560000001HAbEAM
Username             test-wcqe7taxtftt@example.com
Test Run Id          7075600000UWRRN
User Id              0055600000lUS7fAAG
Test Run Coverage    91%
Org Wide Coverage    91%
```

5. As you can see, the test summary report can easily highlight the extent/lack of code coverage, Passed and Failed tests as well as other details that include the id of the org, user who ran the test etc.

Part 2: Deployment and Packaging

Introduction

In the Convert and Deploy an Existing App section of the Trailhead pre-work, we discussed how unmanaged packages can be used to bundle up and retrieve applications and components in your org and convert them to DX projects. This can be a great approach to use when deciding how to organize your Salesforce "Happy Soup" into smaller, manageable pieces that can be leveraged in your Salesforce ALM processes. While unmanaged packages are a great tool, they do have their limitations like the inability to protect your code from changes once deployed or the ability to upgrade or change packages from a distribution perspective. Unlocked Packages helps with these limitations and more.

Unlocked Packages usher in a new way for partners and enterprise customers to develop, distribute, and manage their apps and metadata. We'll explore how to create and manage these packages using the Salesforce DX CLI. We'll also look at how to specify dependencies across packages and how to configure your DX project to support dependencies. Finally, we'll also explore how to version and release packages for deployment into other environments. Let's get started with Unlocked Packages.

Lab Instructions

Important Concepts Before We Start

1. Salesforce Packaging allows you to create multiple types of packages.
    a. **Second-Generation Managed Packages (Beta)** - A managed package is similar to a first-generation managed package. The managed package type is often used by Salesforce partners (ISVs) to develop, distribute, and sell applications to customers. A managed package is fully upgradeable.
    b. **Unlocked Packages (GA)** - These are packages with no manageability rules. The package development team controls the package and can modify the components as needed. Both the subscriber and the package developer can update components but any changes made by the developer "always win" if a package is upgraded.
2. When you create an Unlocked Package, there are multiple IDs that are generated in the background. Each of these IDs can be assigned aliases for easy reference.
   Here are examples and the meanings of the different ID types. Please note that each type of ID always has the same prefix.

ID Example	Short ID Name	Description
04t6A0000004eytQAA	Subscriber Package Version ID	Use this ID to install a package version. Returned by `force:package:version:create`.
0Hoxx00000000CqCAI	Package ID	Use this ID on the command line to create a package version. Or enter it into the sfdx-project.json file and use the directory name. Generated by `force:package:create`.
05ixx00000000DZAAY	Package Version ID	Returned by `force:package:version:create`. Use this ID to specify ancestry among package versions and for promoting a package version preleased using `force:package:version:promote`.
08cxx00000000BEAAY	Version Creation Request ID	ID for a specific request to create a package version such as `force:package:version:create:get`

3. Note these considerations for package versions
    a. Version numbers are formatted as major.minor.patch.build. For example, 1.2.0.8.
    b. Ancestors IDs are used for Managed Packages. Ancestor IDs are also IDs of Packages that can be used to implement an inheritance structure of versions. For our exercise today we will be using Unlocked Packages.
    c. Use the NEXT keyword to automatically increment the build number to the next available for the package. “versionNumber”: “1.2.0.NEXT”.
    d. Use the LATEST keyword to automatically assign the latest version of the package dependency when you create a new package version.

Creating Unlocked Packages

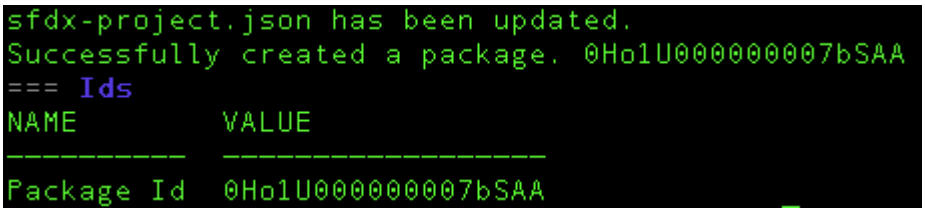
1. Now that we have covered some of the important concepts of Unlocked Packages, let’s create our packages.
2. The aim of creating packages is to divide your source into logically separated, interdependent packages. The source could be separated based on components belonging to a specific app (vertical separation) or based on function vis-a-vis a Model View Controller (MVC) style paradigm.
3. For this exercise we have taken the Dreamhouse App example and separated the metadata components based on function.
4. There are 4 main packages - core, logic, ui and security.
5. When deciding on the components of a package, keep in mind the dependencies between these packages. Make sure there are no circular dependencies.

Generate the core Package

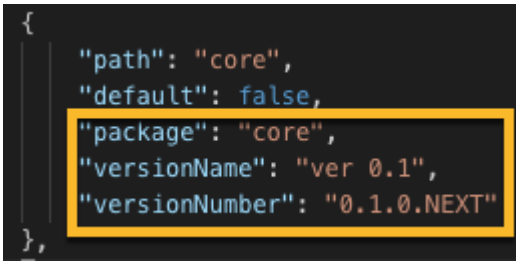
1. core is our base package i.e. it doesn't depend on any other packages.
2. Run the following command-

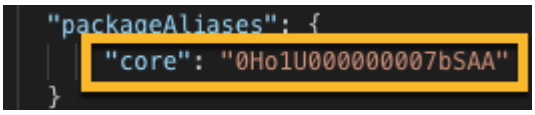
   ```
   sfdx force:package:create -n core -d 'core package' -t Unlocked -r core -e
   ```

   In this command -n is the name of the package, -d is the description, -t is the type of package, -r is the path of the folder and -e specifies that there is no namespace associated with this package.
3. The output of the command will be the Package Id

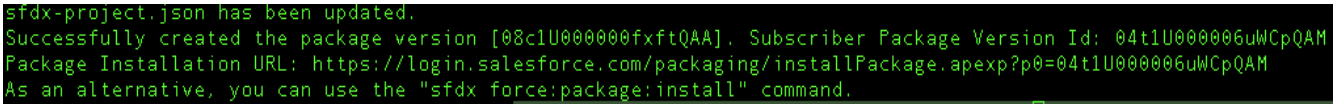
4. The CLI updates the sfdx-project.json in the background. It adds the package name we supplied on the CLI, a version number and a version name for the core package since one wasn't present. This defaults to the *major.minor* version of the package. It also adds a mapping for the core package's alias and Package Id. The alias, version name, version name and version number can all be edited.

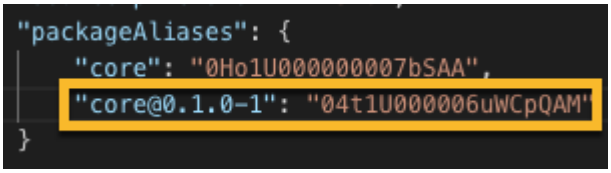
   

5. At this point we have created an empty placeholder for the contents of our package. To push the content within the core directory on our file system we need to create a version of the core package. Run the following command to create a version of the package with the installation key 'packagepwd'. -w specifies the minutes to wait for installation status. The default is 0.

   ```
   sfdx force:package:version:create -p core -k packagepwd -w 60
   ```
6. Hang tight! Version creation can take a little while. At Least one version of the core package has to be fully created before we can create versions of any packages that depend on it. While the version is being created let's start working on the logic package.
7. On successful creation of the package the sfdx-project.json will be updated with alias mappings for the version of the core package. These aliases can be edited manually as well.





Generate the logic Package

1. The logic package contains any independent/utility Apex Classes and Processes.
2. The logic package depends on the 'core' package.
3. First let's create the core package. Run the following command to create the logic package

   `sfdx force:package:create -n logic -d 'logic package' -t Unlocked -r logic -e`
3. The logic package's information will be updated in the sfdx-project.json file similar to the core package.
4. Now before we create a version for the logic package, let's specify the dependencies for the logic package. Add the code found in the logic_package_dependencies file to the declaration of the logic package, right after the version number. Don't forget to add a ',' at the end of the previous line.
5. Your sfdx-project.json should look like the following image

6. Run the following command to create a version of the package after the version of the core package has been successfully created.

   ```
   sfdx force:package:version:create -p logic -k packagepwd -w 60
   ```

4. Once the package is created, notice the addition of the version name and aliases for the logic package to the sfdx-project.json file.

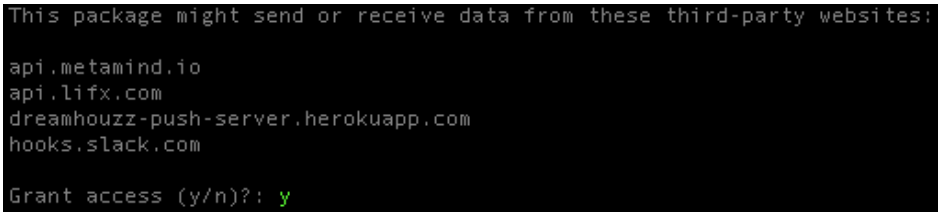
Generate the ui and security packages

1. Use your newly acquired Unlocked Packaging knowledge to generate the ui and security packages.
2. The ui package depends on the core and logic packages.
3. The security package depends on the core, logic and ui packages.
4. Follow the same steps to first generate the package, add the dependency and create the version.
5. Your final sfdx-project.json should look like this file.

Install the Packages in a scratch org

1. Now that you have created at least one version of each package let's install them in a scratch org to check if they install without any errors and the package dependencies are set correctly.
2. Remember that the order of installation of the packages is important and is based on the dependencies defined in the sfdx-project.json file.
3. In our case packages have to be installed in the following order core → logic → ui → security
4. Create a new scratch org and designate it as the default scratch org-
   `sfdx force:org:create -fconfig/project-scratch-def.json -s -a myPkgOrg`
5. You will use the aliases of the package versions you want to install. Execute the following command to list the available versions of all packages -
   `sfdx force:package:version:list`
6. Once the scratch org is created run the following commands one after the other. In these commands -k specifies the installation key, -p is the alias of the package or the package ID, -s specifies which set of users the package is available to (AllUsers or AdminsOnly) and -w specifies wait time for the status messages. (Note - if you have created multiple versions, use the alias of the latest version)
   - `sfdx force:package:install -k packagepwd -p core@0.1.0-1 -s AllUsers -w 60`
   - `sfdx force:package:install -k packagepwd -p logic@0.1.0-1 -s AllUsers -w 60`
   - `sfdx force:package:install -k packagepwd -p ui@0.1.0-1 -s AllUsers -w 60`
   - `sfdx force:package:install -k packagepwd -p security@0.1.0-1 -s AllUsers -w 60`

   `If prompted, type 'y' and press enter`



7. Assign the dreamhouse permset to the scratch org user

   `sfdx force:user:permset:assign -n dreamhouse -u myPkgOrg`

8. Load data -

   `sfdx force:data:tree:import -p assets/data/Broker__c-Property__c-plan.json`

9. Access the scratch org and check if the app is working as expected.

Part 3: Testing and CI/CD (Optional)

Introduction

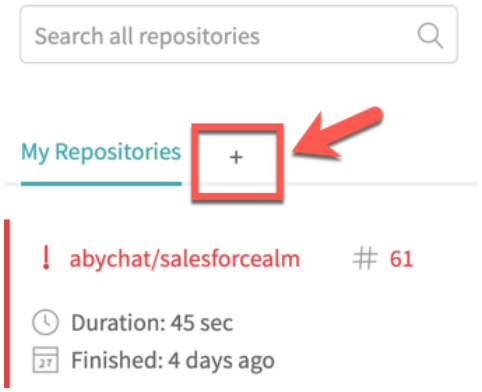
How would you like to make your release process faster and more reliable? Continuous Integration helps developers detect and fix problems, automatically, before releasing an update to their production environments. This section will take you through the process of manually running tests to validate the code and examine the test report to ensure we're in compliance with coverage policies. We'll then setup a CI/CD process to automate the validation and deployment of validated code.

NOTE: *Windows Users there is a known issue with Travis encryption on Windows which may result in encryption/decryption not working. Suggestion is to run Step2 on WSL, Mac or Linux and copy keys to Windows machine.* *https://docs.travis-ci.com/user/encrypting-files*

Lab Instructions

Enabling Continuous Integration (CI) with Travis CI (Bonus)

1. If you are setting up Travis CI now,please follow the instructions at the Continuous Integration Using Salesforce DX trail. Keep in mind the following while following the steps -
    a. Wherever instructed to navigate to a directory/project, all the commands should be executed in the **salesforcealm** directory and not the sfdx-travisci directory.
    b. **IMPORTANT - In the 'Wire it All Together' module, skip the 'Kick Off Continuous Integration' step. Don't worry, we will get to that in the following steps.**
2. If you have already setup TravisCI and completed the Continuous Integration Using Salesforce DX trail, execute the following steps on the Command Prompt/ Terminal from the salesforcealm directory
    a. Enable Travis CI for the salesforcealm repository you forked
        i. On the Travis CI site, click on the + sign next to 'My Repositories'



ii. Locate the salesforcealm repository and toggle the enable switch to turn on CI/CD

b. Run - `travis env set CONSUMERKEY <connected app consumer key>`

c. Run - `travis env set USERNAME <your Dev Hub username>`

d. Copy the server.key project file from the **sfdx-travisci/assets or certificates** directory to the **salesforcealm/assets** folder. Click OK/Yes if prompted to replace any existing files.

e. Open the .travis.yml file in the **salesforcealm** directory.

f. Delete the line starting with `openssl aes-256-cbc…`

g. Issue the following commands (respond yes to any questions)

i. `travis login --org`

ii. `travis encrypt-file assets/server.key assets/server.key.enc --add`

```
bmccall-ltm:salesforcealm bmccall$ travis encrypt-file assets/server.key assets/server.key.enc --add
encrypting assets/server.key for bmc-sf/salesforcealm
storing result as assets/server.key.enc
DANGER ZONE: Override existing assets/server.key.enc? |no| yes
storing secure env variables for decryption

Make sure to add assets/server.key.enc to the git repository.
Make sure not to add assets/server.key to the git repository.
Commit all changes to your .travis.yml.
```

iii. Confirm a new line starting with 'openssl aes-256-cbc…' is added to the travis.yml file

3. Go to the command prompt and execute the following sequence of commands

```
git add .
git commit -m 'Added property map LWC'
git push origin labs
```

4. With the push to the remote, you should see the Travis CI job in action. This script is similar to the one available in the Trailhead module. It simply creates a scratch org, pushes all the code into the scratch org, runs Apex tests and deletes the scratch org. The failure of any of these steps will cause the build to fail.
5. Get Excited! In the next section we will look at creating Packaging 2 packages and including them in the CI process.

Including Packages in the CI process

1. As promised, we are going to include packages in the CI process. Let's get on it.
2. The Salesforce CLI makes integrating with a variety of CI tools really easy and flexible. Depending on the CI you are using you can add different degrees of dynamicity into your CI scripts.
3. In this case we are using Travis CI and we will be making some changes to our scripts to introduce environment variables that hold the IDs for the packages that we want to use in our script. This will enable us to change the IDs of the packages as we create new versions without having to replace the ID in several places.
4. Open the .travis.yml file.
5. Replace the text under the **env** block at the beginning of the script with the text in the file @ https://github.com/abychat/salesforcealm/blob/master/assets/travis_package.txt
6. Don't forget to update the aliases of your packages used in your sfdx-project.json file if they are different from the text you copied in the .travis.yml file in the previous step. Please keep in mind that the indentation is important.
7. We just defined our environment variables. Now let's use them in our script. Copy the text found @ https://github.com/abychat/salesforcealm/blob/master/assets/travis_package_install.txt
8. Paste the text copied in step 7 at the end of the file under the **script** section of the travis.yml file
9. These commands are creating a scratch org, installing the packages in the org to check if anything errors out and subsequently destroying the scratch org.
10. Created a new version of the package? Just update the package version alias in the travis.yml file before you commit your changes and you are good to go.
11. To see your updated Travis CI script in action just save all files, commit your changes to git and push to your remote repository.

Wrap-Up

Review

Feeling Adventurous? - Try On Your Own

1. In the real world, you would test all your code and packages in scratch org and eventually push them to an integrated sandbox environment (SIT, UAT etc).
2. Let's look at what we would have to do install our packages in a sandbox.
3. The awesome thing about SalessforceDX is that you can push your code and your packages from the CLI to any **unrelated** environments (scratch org, sandbox and production orgs). Unlike Change Sets, you are not restricted from moving metadata between sandboxes that have spawned from the same production org.
4. For the rest of this exercise you are going to need a sandbox. You can use your own sandbox - any type would do - or you could check with the Salesforce team to see if they have a Sandbox available.
5. The first step is to authorize your sandbox to work with the CLI. Run the following command - `sfdx force:auth:web:login -r https://test.salesforce.com -a PkgSandbox`
6. Enter your sandbox credentials in the browser window that opens up and return to the Terminal app after successfully logging in.
7. Make sure you have never installed the Dreamhouse app in this sandbox before.
8. Install the packages into the sandbox using the package install commands we used in Part 2. Remember the username should specify the sandbox user alias, PkgSandbox for example as used in step 5.
9. Assign the dreamhouse permission set to the sandbox user once all the packages are installed.
10. Access your sandbox and explore the Dreamhouse app.