Build Your First AppExchange App Workbook

Welcome to Build Your First AppExchange App! This workbook is designed to take you and other ISV partners through the process of creating a simple app on the Salesforce platform, packaging the solution, and making it available for distribution.

Part 1 | Create a Developer Edition Org and Define a Namespace

In this project, we will work together to use different Salesforce technologies to build a fully functioning Property Management app. Our goal is to give you, our Independent Software Vendor partner, a fundamental introduction to AppExchange app development.

By the time you have finished completing all parts of this workbook, you will have created a custom table to store records pertinent to property management, a custom user interface to show the properties on a map, a simple wizard to book property maintenance and a report visualizing the number of properties owned by each landlord. What's more, all of this will be packageable and installable into customer instances of Salesforce.

This workbook will use 2nd Generation Managed Packaging, which will help you better organize your source code, integrate with your version control system, and improve utilization of your custom Apex code. To learn more about 2nd Generation Packaging, take a look at the Salesforce DX Developer Guide here.

Create a Developer Edition Org

In this first section, we'll use the Environment Hub within your Partner Business Org (PBO) to create a Partner Developer Edition org. The Environment Hub lets you connect, create, view, and log in to Salesforce orgs from one location. If your company has multiple environments for development, testing, and trials, the Environment Hub lets you streamline your approach to org management.

1. To create a Partner Developer Edition org and assign a unique namespace, log into your Partner Business Organization (PBO).
2. Open the App Launcher.
3. Find and select "Environment Hub." You can do this by typing "Environment Hub" into the search box and clicking on it when it appears (see Figure 1.1). Or you can scroll down and find the Environment Hub link at the bottom of the App Launcher page.

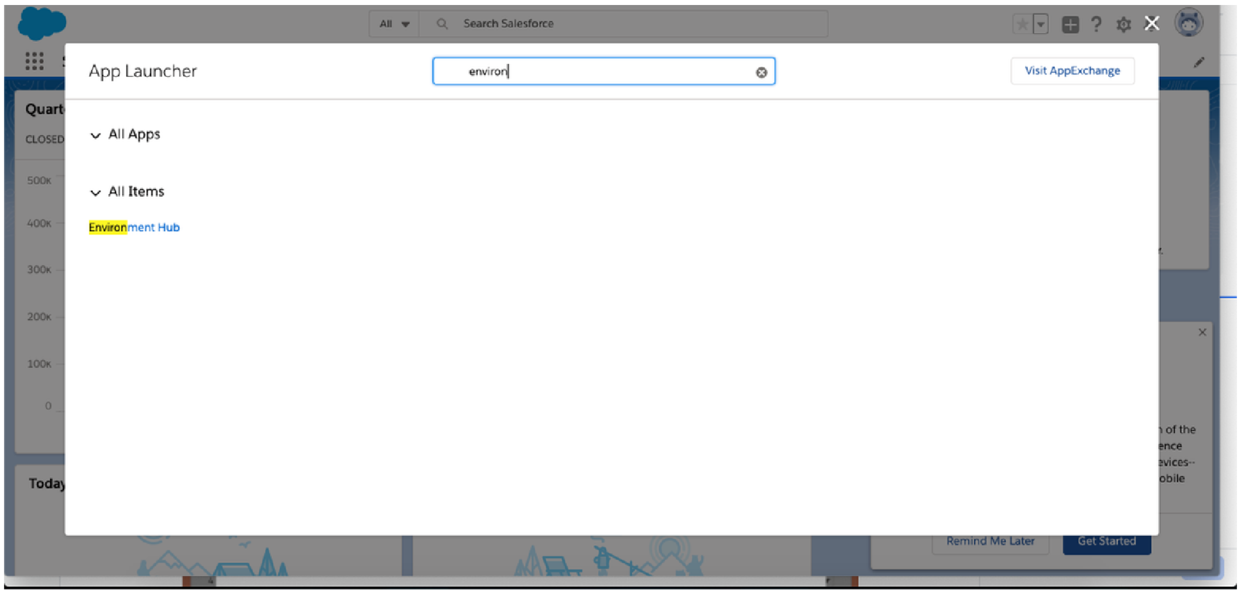
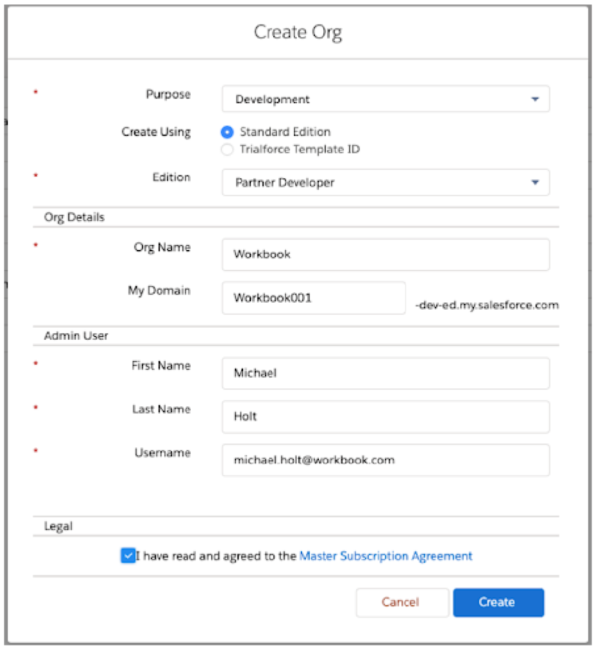


Figure 1.1: After you open the App Launcher, type “Environment Hub” into the search box and then click the Environment Hub hyperlink.

4. In the upper right corner of the Environment Hub tab, select Create Org.
5. Fill out the Create Org form as shown in Figure 1.2. The My Domain name must be unique among all Salesforce domains. Because this is an example project, choose a My Domain name you won’t need again; don’t use a name that you might use for a real app that you plan to develop. I’ve chosen Workbook001.

Note: Your user name must be in the form of a unique email address that you have not used to sign-up for other Salesforce accounts. It does not need to be a real email address.



Create Org

* Purpose	Development
Create Using	[✓ Standard Edition] Trialforce Template ID
* Edition	Partner Developer

Org Details

* Org Name	Workbook
My Domain	Workbook001 -dev-ed.my.salesforce.com

Admin User

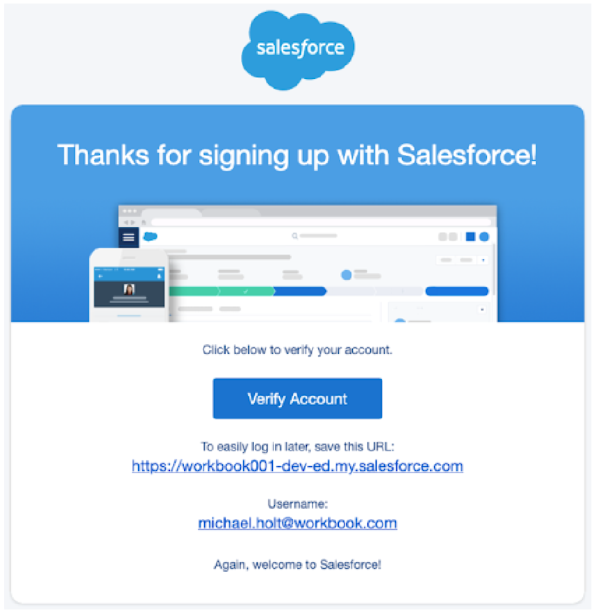
* First Name	Michael
* Last Name	Holt
* Username	michael.holt@workbook.com

Legal

[✓] I have read and agreed to the Master Subscription Agreement

Figure 1.2: Enter all the fields in the Create Org form. The My Domain name must be unique, so choose one that you do not plan to use again.

6. After you have completed the Create Org form, click the Create button. Wait patiently while our infrastructure provisions your very own Partner Developer Edition org. When your org is ready, you will receive an email message that will look similar to the one in Figure 1.3.
7. Click the Verify Account button.


Thanks for signing up with Salesforce!

Click below to verify your account.

To easily log in later, save this URL:
https://workbook001-dev-ed.my.salesforce.com

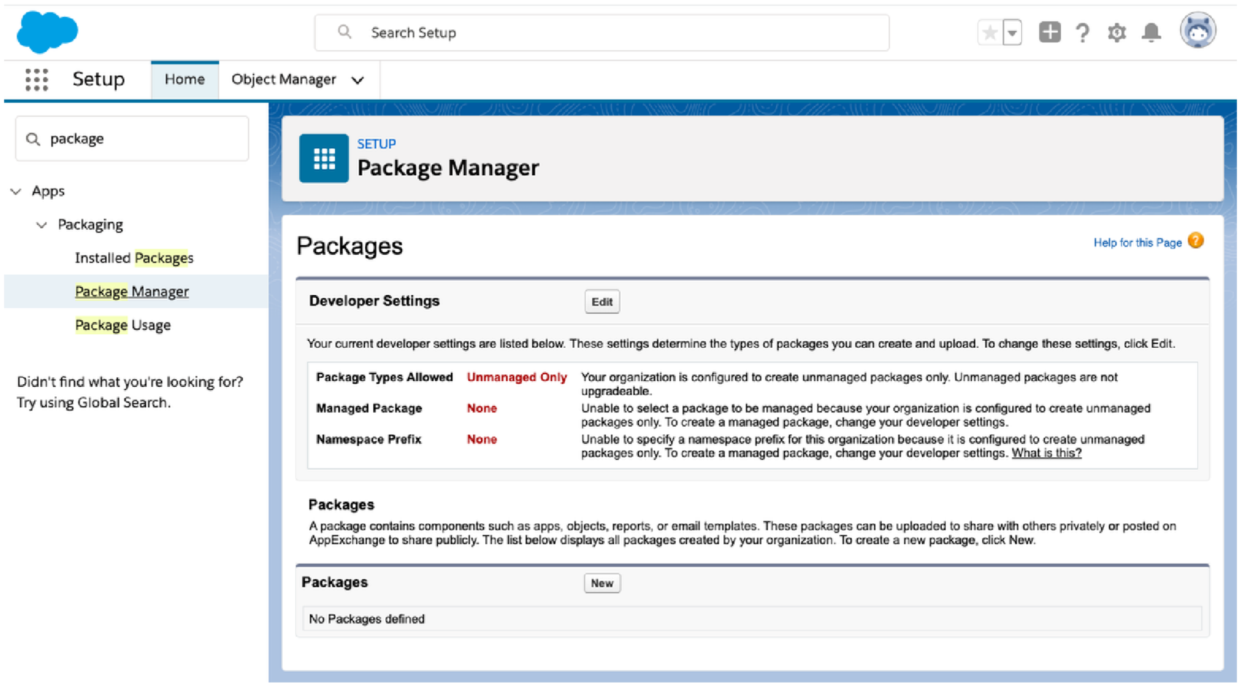
Username:
michael.holt@workbook.com

Again, welcome to Salesforce!

Figure 1.3: When your Partner Developer Edition org is ready, you will receive an email message asking you to verify your account.

8. A screen will appear asking you to change your password. Enter and confirm a new password and set up a security question. When you have logged in, the Setup Home screen will appear.

9. In the Quick Find box on the left corner of the screen, enter "Package Manager." When you see Package Manager appear in the list below the Quick Find box , select it. The Setup Package Manager box will appear (Figure 1.4).



Developer Settings

Your current developer settings are listed below. These settings determine the types of packages you can create and upload. To change these settings, click Edit.

Package Types Allowed	Unmanaged Only	Your organization is configured to create unmanaged packages only. Unmanaged packages are not upgradeable.
Managed Package	None	Unable to select a package to be managed because your organization is configured to create unmanaged packages only. To create a managed package, change your developer settings.
Namespace Prefix	None	Unable to specify a namespace prefix for this organization because it is configured to create unmanaged packages only. To create a managed package, change your developer settings. What is this?

Packages

A package contains components such as apps, objects, reports, or email templates. These packages can be uploaded to share with others privately or posted on AppExchange to share publicly. The list below displays all packages created by your organization. To create a new package, click New.

Packages

No Packages defined

Figure 1.4: The Setup Package Manager screen.

10. Click the Edit button in the Developer Settings section.

11. In the Change Developer Settings screen, click the Continue button.

Enter a unique name to register a namespace prefix. **The namespace prefix needs to be unique and can never be changed, modified or removed**. Click the Check Availability button to see if the name you entered is not already in use.

12. From the Package to be managed drop down menu, select the package you created in the previous steps. Click the Review My Selections button.

13. Review the **Namespace Prefix** and the **Package to be managed**. If both are correct, click the Save button to return to the Setup Package Manager screen in your Partner Business org.

Set-up My Domain

My Domain enables you to create your own subdomain for your Salesforce org. With a subdomain, you can include your company name in your URL, for example: `https://yourcompanyname.my.salesforce.com.`

***Note**: You set up My Domain in your Partner Business Org, not in the packaging org.*

My Domain enables you to:

- Redirect to your single sign-on provider
- Use SAML Identity Provider to single sign-on to all your apps
- Work in multiple Salesforce orgs at the same time
- Customize your login screen

There are four simple steps to set up My Domain.

Step 1: Register a Subdomain

The first step is to pick and register your subdomain name:

1. Click the Setup cog in the upper right corner of your Partner Business Org.
2. Enter “My Domain” in the Quick Find box. When My Domain link appears, click on it.
3. In Step 1 of the Setup My Domain screen, enter the name of your subdomain. The sub-domain name must be unique across all Salesforce orgs, so you will need to be creative when choosing a name. Click the Check Availability button to find out if your sub-domain name is already taken.
4. If the name is available, click Register Domain.

Step 2: Receive Email Domain Registration Confirmation

After a few minutes, you’ll receive an email letting you know that your subdomain is ready for testing. Click the link in the email to login into My Domain. If you still have your org open, click Setup, find My Domain, and click on it.

Step 3: Verify that the Domain is Ready

When your domain is ready, you’ll see the URL in your browser’s address bar and at the bottom of the My Domain Setup page. If the URL looks good, click the Log In button.

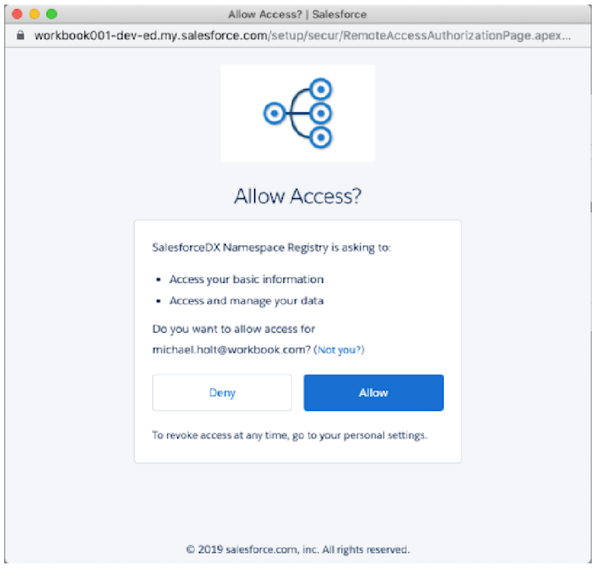
Step 4: Deploy to Users

Click the Deploy to Users button. Now, all of your users can see the subdomain.
Note: Don’t forget to do this step. Your users can’t access the org with the subdomain URLs until you deploy it.

Register the Namespace

Now that you have setup a My Domain subdomain, you can register the namespace.

1. Return to your Partner Business Org. Open the App Launcher and in the search box at the top of the screen enter “Namespace Registries.”
2. In the upper right corner of the Namespace Registries page, select the “Link Namespace” button.
3. Enter the credentials for the Partner Developer Edition which you just created. Choose “Allow” in the prompt which appears (Figure 1.5).


Allow Access? | Salesforce

workbook001-dev-ed.my.salesforce.com/setup/secur/RemoteAccessAuthorizationPage.apex...

Allow Access?

SalesforceDX Namespace Registry is asking to:

- Access your basic information
- Access and manage your data

Do you want to allow access for michael.holt@workbook.com? (Not you?)

Deny Allow

To revoke access at any time, go to your personal settings.

© 2019 salesforce.com, inc. All rights reserved.

Figure 1.5: Click the Allow button to enable the namespace to appear in the namespace registries.

Once done, your newly defined namespace will appear in your list of namespace registries. This allows us to create namespaced scratch orgs, something which will be extremely beneficial for our package development!

Enable Unlocked and Second-Generation Managed Packaging

Enable packaging in your org so you can develop second-generation managed packages. You can work with the packages in scratch orgs, sandbox orgs, and target subscriber orgs.

Enable Dev Hub in your org.

1. Log in to the org where you've enabled Dev Hub.
2. From Setup, enter `Dev Hub` in the Quick Find box and select **Dev Hub**.
3. In the section for Second-Generation Packaging, click **Non-GA Service Agreement** to read the service agreement.
4. Click **Enable Unlocked Packages and Second-Generation Managed Packages**. After you enable Second-Generation Packaging, you can't disable it.

Part 2 | Create a Salesforce DX Project and a Scratch Org

Salesforce DX enables you to set up a project structure for your org's metadata, your org templates, your sample data, and all Apex Unit tests and Jest tests. You can manage these items in your preferred version control system (VCS) to maintain project integrity, particularly when you're ready to add new features to your app.

In this tutorial, we'll use the Command Line Interface (CLI), Salesforce Extension Pack, and Visual Studio Code to create a project. Then we will authenticate the project against our Partner Business Org, which is where you've enabled the Developer Hub.

In the Dev Hub, we will create a Scratch org and start building our first components of the app on the Salesforce platform! Let's dive in!

Create a Project

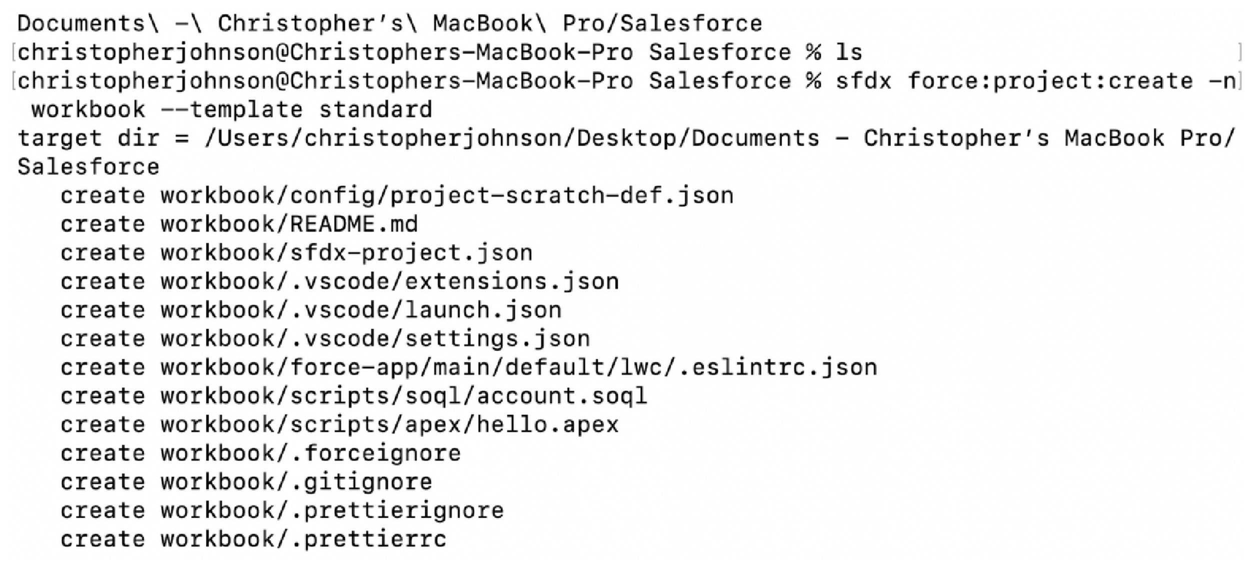
To create a project:

1. Open up your terminal window and enter the following command:

```
sfdx force:project:create -n workbook --template standard
```

Salesforce DX will create the files you'll need to start your project. The project is named "workbook," and the project will use the standard template. The template gives us additional files that will help us manage version control and useVSCode extensions.

When DX is finished generating the project files, your terminal should look similar to Figure 2.1.



```
Documents\ -\ Christopher's\ MacBook\ Pro/Salesforce
christopherjohnson@Christophers-MacBook-Pro Salesforce % ls
christopherjohnson@Christophers-MacBook-Pro Salesforce % sfdx force:project:create -n
 workbook --template standard
target dir = /Users/christopherjohnson/Desktop/Documents - Christopher's MacBook Pro/
Salesforce
   create workbook/config/project-scratch-def.json
   create workbook/README.md
   create workbook/sfdx-project.json
   create workbook/.vscode/extensions.json
   create workbook/.vscode/launch.json
   create workbook/.vscode/settings.json
   create workbook/force-app/main/default/lwc/.eslintrc.json
   create workbook/scripts/soql/account.soql
   create workbook/scripts/apex/hello.apex
   create workbook/.forceignore
   create workbook/.gitignore
   create workbook/.prettierignore
   create workbook/.prettierrc
```

Figure 2.1: After DX has generated your project files, your terminal should look like this.

2. Once DX finished generating the project files, move into the project folder using the following command: `cd workbook`
3. Open our project in VSCode using this command: `code .`

Note: Because we'll be using the Salesforce CLI and VS Code, you may want to open the terminal window in VSCode. Select Terminal > New Terminal from the navigation menu or type ^ ~.

Authenticate Against the Dev Hub

Next, we'll need to authenticate against our Dev Hub, which lives within our Partner Business Org. In the terminal window, type this command:

```
sfdx force:auth:web:login -d -a DevHub.
```

The -d tells SFDX that this is our dev hub org and -a gives the org an alias. After the command executes, a Salesforce web page will launch and request that you enter your Partner Business Org credentials.

After allowing access to your Partner Business Org, a message in the terminal will confirm that this action has been completed and we no longer need the browser. Close the browser window and return to VSCode.

Develop with a Scratch Org

Our next task is to create the all-important scratch org. A scratch org is an environment that can be configured to emulate different features and preferences. For example, you can use scratch orgs to test your code in different Salesforce editions. Scratch orgs are source-driven and disposable, so you can create as many as you need. You can share a scratch org configuration file with other team members, so you all have the same basic org in which to do your development.

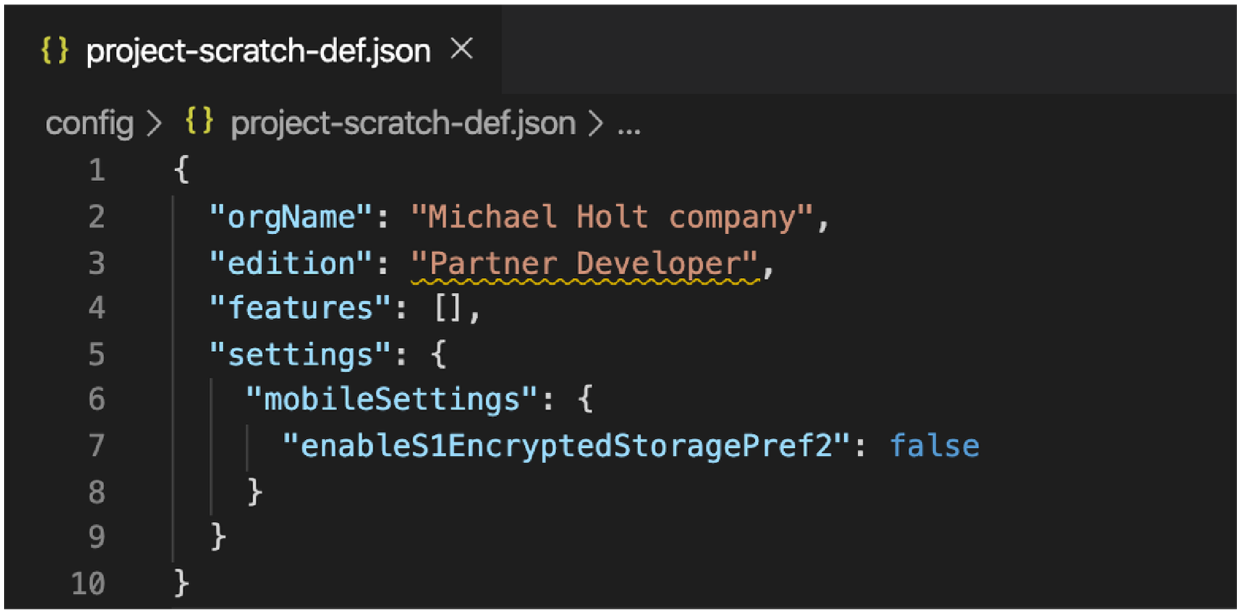
Configure the Scratch Org

Let's take a look at some of the project files we created within our project. In VSCode, navigate to the project-scratch-def.json file which you will find in the project's config folder. The file holds the configuration for the scratch org. Within it, we can enable all sorts of features that might be applicable to our use cases.

1. Add the following, within the enclosing brackets of the definition file:

```
"settings": {
 "mobileSettings": {
 "enableS1EncryptedStoragePref2": false
 }
 }
```

2. Change the edition to "Partner Developer." This will allow us to create the org with enhanced limits. The resulting file will look like Figure 2.2.

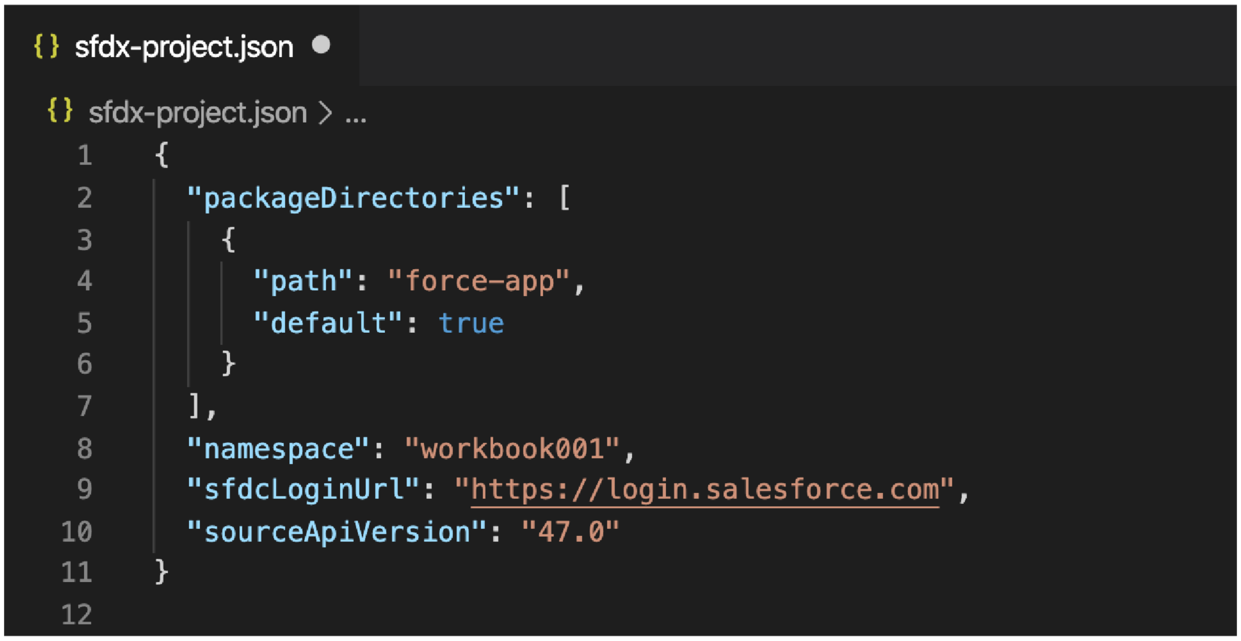

{} project-scratch-def.json
config > {} project-scratch-def.json > ...

```
{
  "orgName": "Michael Holt company",
  "edition": "Partner Developer",
  "features": [],
  "settings": {
    "mobileSettings": {
      "enableS1EncryptedStoragePref2": false
    }
  }
}
```

Figure 2.2: The project-scratch-def.json configuration.

Let's take a look at the sfdx-project.json file, which sits in the root of the project folder. Remember when we registered the namespace with our PBO? In order to make use of the namespace, we need to add the namespace the sfdx-project.json file.

After you add your namespace (remember, your namespace will be different to my own), your sfdx-project.json file should look similar to the one shown in Figure 2.3.



```json
{
  "packageDirectories": [
    {
      "path": "force-app",
      "default": true
    }
  ],
  "namespace": "workbook001",
  "sfdcLoginUrl": "https://login.salesforce.com",
  "sourceApiVersion": "47.0"
}
```

Figure 2.3: Add your namespace to the sfdx-project.json file. Instead of workbook001, use the namespace that you created earlier.

Create the Scratch Org

Now that we've made the modifications to our scratch org's instructions, it's time to create it. We can do that using the Salesforce CLI In your terminal, enter the following command:

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

SFDX will create the scratch org using the configuration file we've modified. In addition, specifying an alias using -a allows us to refer to the correct scratch org in future commands. Lastly, -s sets the org as our *default scratch org*, meaning we don't necessarily need to provide the alias in future commands. For clarity in this workbook, we will specify the alias regardless.

Note*: Your directory could be different, depending on the name of your project and where you are within your terminal*.

When the scratch org is created, open it using the following command, specifying the alias of the org:

```
sfdx force:org:open -u workbookScratch
```

The scratch org will open in your default browser and present the setup menu on the home screen. If the org opens with an error, fear not. This happens when the org is opened before its domain has been properly registered. If you sit tight for a minute or two and then refresh the page, the error should resolve itself.

Congratulations! You've taken your first steps on your journey to AppExchange. Now it's time to start building!

Part 3 | Create and Modify Objects

Now that our Salesforce DX project is setup, it's time to begin work on the Property Management app. We'll start by working with **objects**.

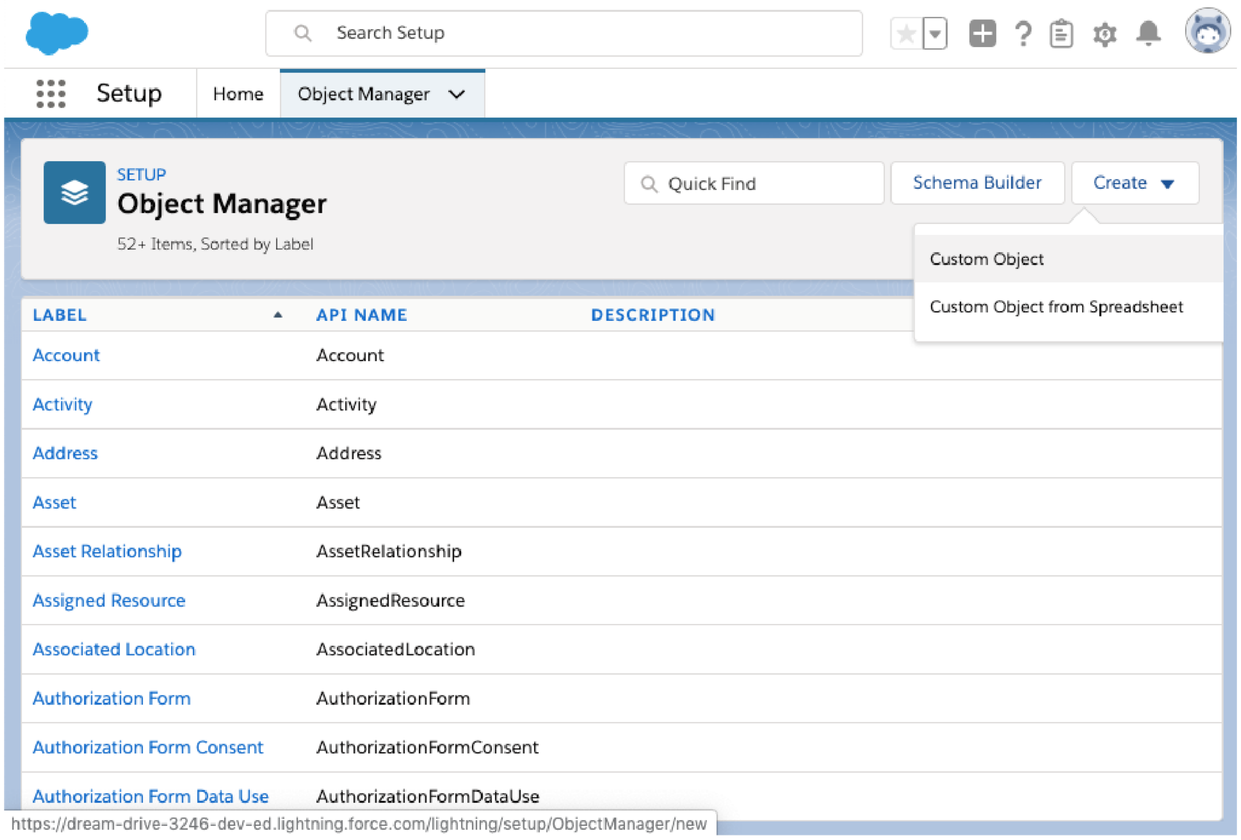
Generally speaking, objects represent database tables that contain your organization's information. For example, the central object in the Salesforce data model represents accounts–companies and organizations involved with your business, such as customers, partners, and competitors. The term "record" describes a particular occurrence of an object (such as a specific account like "IBM" or "United Airlines" that is represented by an Account object). A record is analogous to a row in a database table.

Objects already created for you by Salesforce are called standard objects. Objects you create in your projects are called custom objects. Objects you create that map to data stored outside of Salesforce are called external objects.
In this tutorial, we will create a custom object, a table that contains data that defines the attributes of different properties that the property management app will help manage.

Define a Custom Object

The first thing we'll need to do is define our object model. The core of our object model for a property management solution, will be the property itself, so let's create that as an object!

1. Opening our Scratch org in our browser, navigate to the Setup Object Manager screen and then select Create > Custom Object (Figure 3.1).



LABEL	API NAME	DESCRIPTION
Account	Account	
Activity	Activity	
Address	Address	
Asset	Asset	
Asset Relationship	AssetRelationship	
Assigned Resource	AssignedResource	
Associated Location	AssociatedLocation	
Authorization Form	AuthorizationForm	
Authorization Form Consent	AuthorizationFormConsent	
Authorization Form Data Use	AuthorizationFormDataUse	

https://dream-drive-3246-dev-ed.lightning.force.com/lightning/setup/ObjectManager/new

Figure 3.1: To create a custom object, open the Setup Object Manager screen and select Custom Object from the Create pull-down menu.

2. On the New Custom Objects screen, enter the following options:

- Label = Property
- Plural Label = Properties
- Object Name = Property
- Record Name = Property Name
- Data Type = Text
- Allow Reports = True
- Allow Activities = True
- Object Classification & Deployment Status we can leave as the default settings
- Allow Search = True
- Launch New Custom Tab Wizard after saving this custom object = True

3. Click Save.

4. On the New Custom Object Tab, select any icon from the "Tab Style" lookup. I've chosen "Bank." Click the Next button.

5. On the Add to Profiles page, choose "Apply one tab visibility to all profiles" and change this to Tab Hidden. Click the Next button.

6. On the Add Custom Apps page, uncheck "Include Tab" for every profile and then click the Save button.

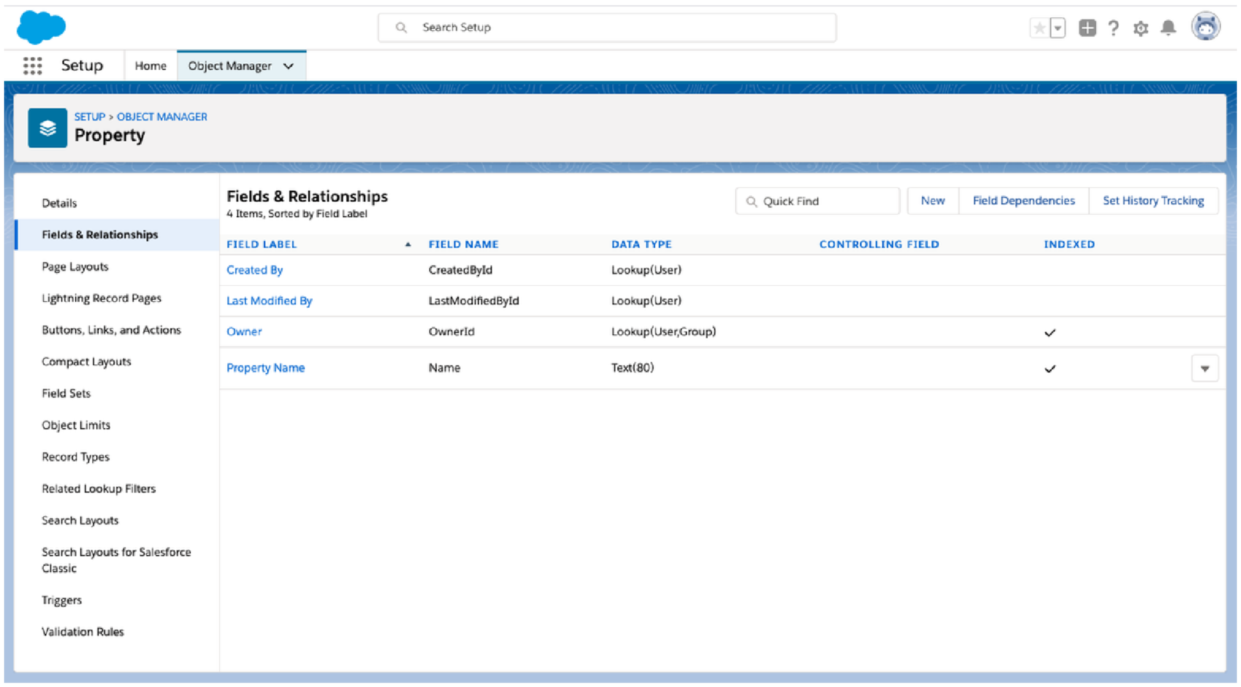
Congratulations! You've built a database table in Salesforce. Look at the object you created. Notice that the object API name is prefixed with your namespace.

The namespace ensures that all objects–fields, components, classes, and so on–are unique, across the ***entire*** Salesforce customer base! In practical terms, this means your app can be installed in customer instances without clashing with objects and components that our customers may have created in their own orgs.

Add Fields to the Object

Now let's allow our users to provide a bit more detail to the property by adding some additional fields to the object.

1. Select "Fields and Relationships" in the left hand pane of the Property object and add a couple of fields (Figure 3.2).

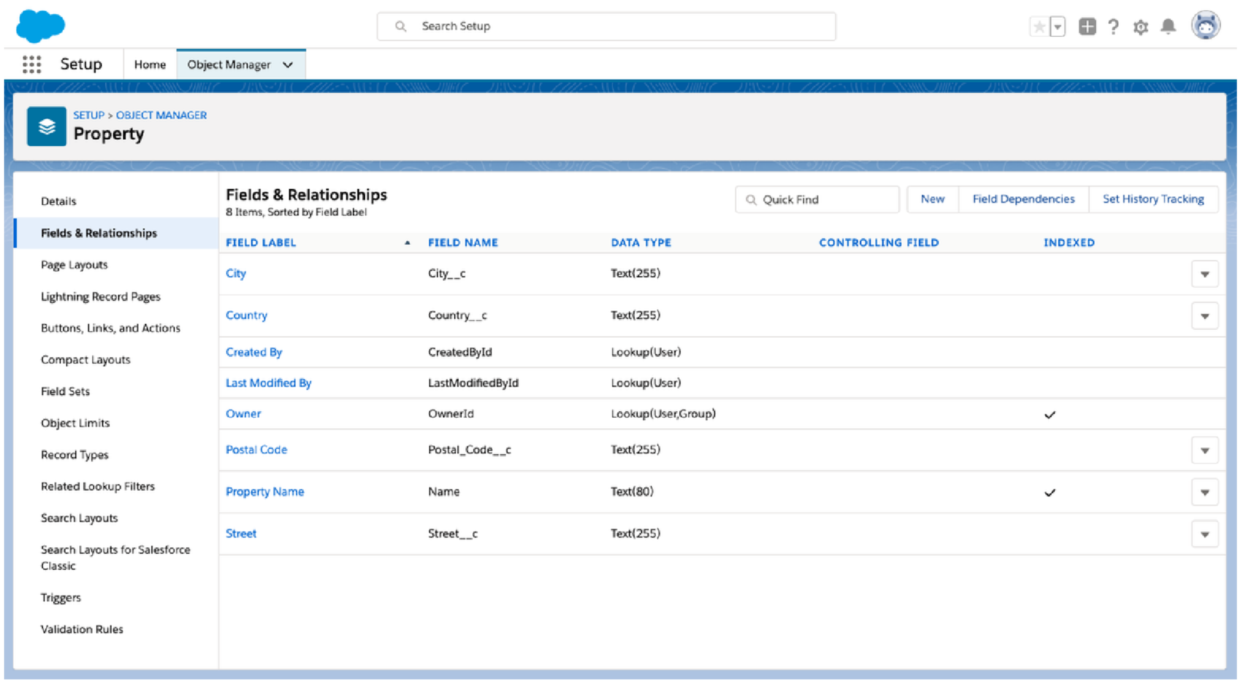


Fields & Relationships
4 Items, Sorted by Field Label

FIELD LABEL	FIELD NAME	DATA TYPE	CONTROLLING FIELD	INDEXED
Created By	CreatedById	Lookup(User)		
Last Modified By	LastModifiedById	Lookup(User)		
Owner	OwnerId	Lookup(User,Group)		✓
Property Name	Name	Text(80)		✓

Figure 3.2: The Fields & Relationships page.

2. Select the New button. In the Choose the field type screen, choose “Text” and then click Next.
3. In the next screen, enter “Street” as the Field Label & Field Name and Length as 255. Click Next.
4. On the Establish field-level security screen, uncheck Visibility for all profiles. Click Next and then Save.
5. Follow the above process, creating fields for City, State, Post Code and Country, until your object looks like Figure 3.3.



Fields & Relationships
8 Items, Sorted by Field Label

FIELD LABEL	FIELD NAME	DATA TYPE	CONTROLLING FIELD	INDEXED
City	City__c	Text(255)		
Country	Country__c	Text(255)		
Created By	CreatedById	Lookup(User)		
Last Modified By	LastModifiedById	Lookup(User)		
Owner	OwnerId	Lookup(User,Group)		✓
Postal Code	Postal_Code__c	Text(255)		
Property Name	Name	Text(80)		✓
Street	Street__c	Text(255)		

Figure 3.3: The object models with the Street, City, State, Postal Code, and Country fields added.

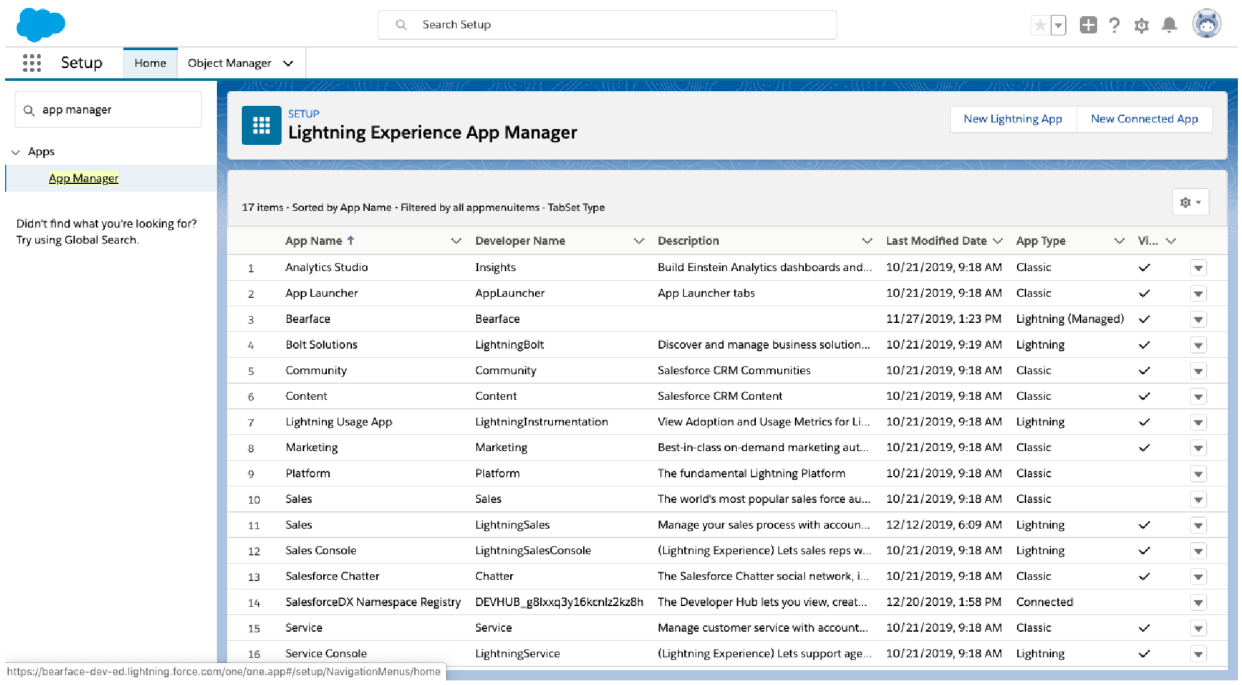
Let's create one more field, but this time, choose "Lookup Relationship."

1. On the Field & Relationships page, click New.
2. On the Choose field type page, choose "Lookup Relationship" and click Next.
3. On the Choose the related object page, select "Contact" from the drop down and then press Next.
4. On the next page, change "Contact" to say "Owner" in both Field Label and Field Name, leave everything else as is and press Next.
5. Once again, we will clear out the Visibility field for all profiles and select Next.
6. Leave Steps 5 & 6 as they are and save.

Great! We have a new object in Salesforce to store a property name, address and owner. Since this is an example app, let's leave it there.

As an ISV though, what we'd really like to do is to keep all of the functionality related to our application contained within its own App in the App Launcher of Salesforce. So let's do that by coming out of the Object Manager and returning to Setup Home.

1. Search for and click on "App Manager" in the Setup menu. Then click the New Lightning App button, which is located in the upper right corner of the page (Figure 3.4).



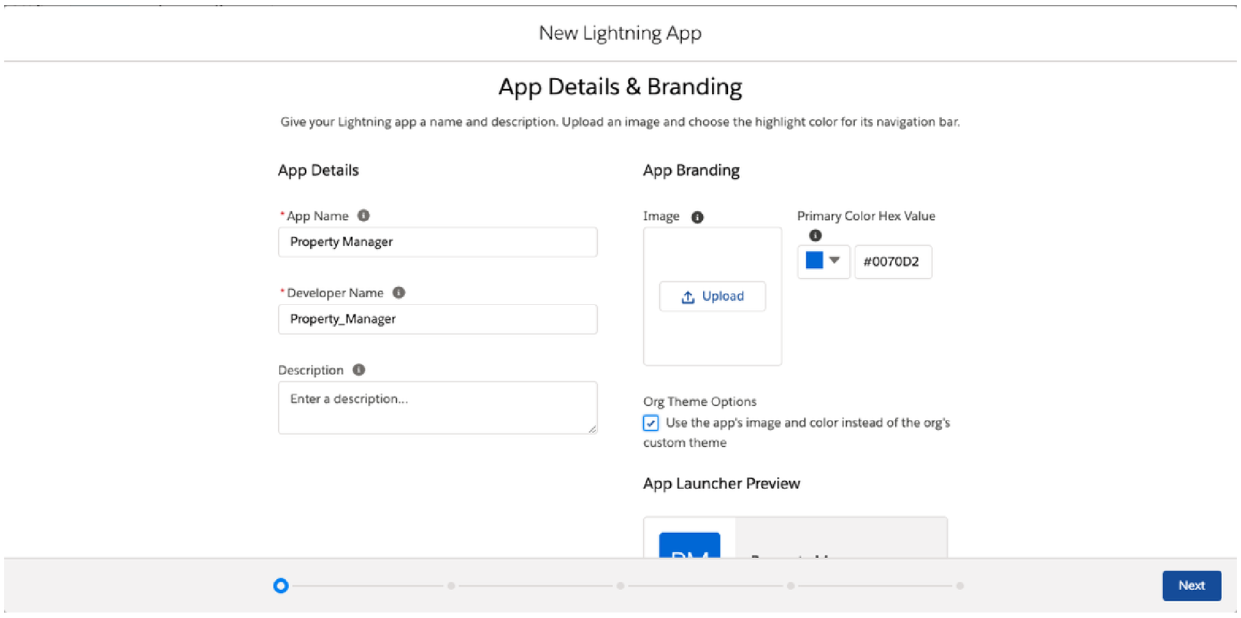
Search Setup
Setup
Home
Object Manager
app manager
Apps
App Manager
Didn't find what you're looking for? Try using Global Search.
SETUP
Lightning Experience App Manager
New Lightning App
New Connected App
https://bearface-dev-ed.lightning.force.com/one/one.app#/setup/NavigationMenus/home

17 items • Sorted by App Name • Filtered by all appmenuitems - TabSet Type

	App Name	Developer Name	Description	Last Modified Date	App Type	Vi...
1	Analytics Studio	Insights	Build Einstein Analytics dashboards and...	10/21/2019, 9:18 AM	Classic	✓
2	App Launcher	AppLauncher	App Launcher tabs	10/21/2019, 9:18 AM	Classic	✓
3	Bearface	Bearface		11/27/2019, 1:23 PM	Lightning (Managed)	✓
4	Bolt Solutions	LightningBolt	Discover and manage business solution...	10/21/2019, 9:19 AM	Lightning	✓
5	Community	Community	Salesforce CRM Communities	10/21/2019, 9:18 AM	Classic	✓
6	Content	Content	Salesforce CRM Content	10/21/2019, 9:18 AM	Classic	✓
7	Lightning Usage App	LightningInstrumentation	View Adoption and Usage Metrics for Li...	10/21/2019, 9:18 AM	Lightning	✓
8	Marketing	Marketing	Best-in-class on-demand marketing aut...	10/21/2019, 9:18 AM	Classic	✓
9	Platform	Platform	The fundamental Lightning Platform	10/21/2019, 9:18 AM	Classic	
10	Sales	Sales	The world's most popular sales force au...	10/21/2019, 9:18 AM	Classic	
11	Sales	LightningSales	Manage your sales process with accoun...	12/12/2019, 6:09 AM	Lightning	✓
12	Sales Console	LightningSalesConsole	(Lightning Experience) Lets sales reps w...	10/21/2019, 9:18 AM	Lightning	✓
13	Salesforce Chatter	Chatter	The Salesforce Chatter social network, i...	10/21/2019, 9:18 AM	Classic	✓
14	SalesforceDX Namespace Registry	DEVHUB_g8lxxq3y16kcnlz2kz8h	The Developer Hub lets you view, creat...	12/20/2019, 1:58 PM	Connected	
15	Service	Service	Manage customer service with account...	10/21/2019, 9:18 AM	Classic	✓
16	Service Console	LightningService	(Lightning Experience) Lets support age...	10/21/2019, 9:18 AM	Lightning	✓

Figure 3.4: The Lightning Experience App Manager.

2. In the App Details and Branding page (Figure 3.5), fill out Property Manager and Property_Manager as the App Name and Developer Name, respectively. Feel free to upload a nice image to accompany your app and style the colour to your choosing. To retain our own branding within our app when installed to a customer org, we'll check the box "Org Theme Options" box.


New Lightning App

App Details & Branding

Give your Lightning app a name and description. Upload an image and choose the highlight color for its navigation bar.

App Details

* App Name: Property Manager

* Developer Name: Property_Manager

Description: Enter a description...

App Branding

Image

Primary Color Hex Value: #0070D2

Org Theme Options

✓ Use the app's image and color instead of the org's custom theme

App Launcher Preview

Figure 3.5: In the App Details and Branding box, name the new Lightning App and check Org Theme Options.

3. Leave App Options and Utility Items as they are and step through the wizard to “Navigation Items.” Here, we can choose from all of the standard and custom tabs within the org. Let’s just choose “Properties” for now and click “Next.”
4. Finally, leave the last page blank and hit “Save & Finish.”

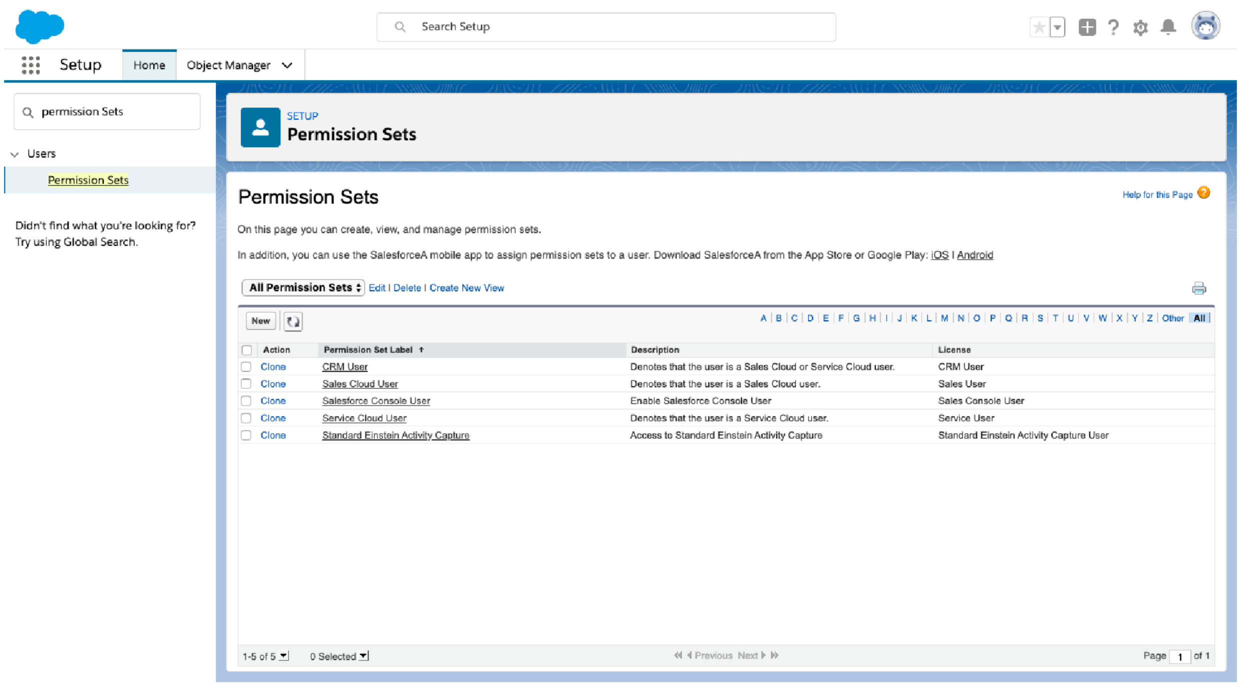
Ideally, at this point you’d be able to click the App Launcher and find that the “Property Manager” App is visible there. Alas, you’ll find that it’s not. This is because we’ve hidden the tab and the app from all profiles. This is done for a very good reason.

Create a Permission Set

As an ISV, we will control access to our application’s functionality using Permission Sets. Permission Sets can be added to specific Users or Profiles by your end customers, in order to provide the relevant access that their users need. You may choose to include multiple permission sets within your solution, for example an “Admin” and “Standard User“ permission which your customers can provision accordingly.

For this project , we’ll create a single permission set which gives access to everything we create. So let’s do that now!

1. In Setup Home, enter “Permission Sets” in the Quick Find box. When Permission Sets appear below the box, click on it (See Figure 3.6).


Search Setup
Setup
Home
Object Manager
permission Sets
Users
Permission Sets
Didn't find what you're looking for? Try using Global Search.
SETUP
Permission Sets
Help for this Page
All Permission Sets
Edit | Delete | Create New View
New
A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Other | All
1-5 of 5
0 Selected
Previous Next
Page 1 of 1

Permission Sets

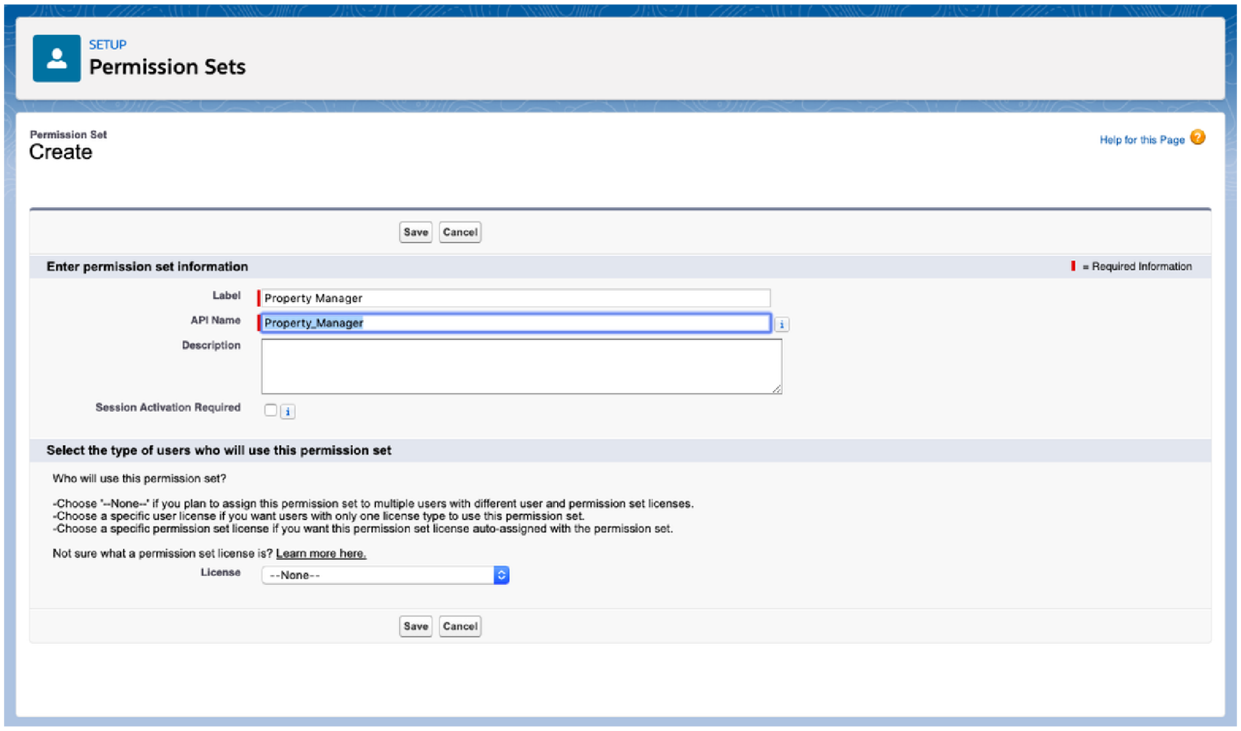
On this page you can create, view, and manage permission sets.

In addition, you can use the SalesforceA mobile app to assign permission sets to a user. Download SalesforceA from the App Store or Google Play: iOS | Android

Action	Permission Set Label ↑	Description	License
Clone	CRM User	Denotes that the user is a Sales Cloud or Service Cloud user.	CRM User
Clone	Sales Cloud User	Denotes that the user is a Sales Cloud user.	Sales User
Clone	Salesforce Console User	Enable Salesforce Console User	Sales Console User
Clone	Service Cloud User	Denotes that the user is a Service Cloud user.	Service User
Clone	Standard Einstein Activity Capture	Access to Standard Einstein Activity Capture	Standard Einstein Activity Capture User

Figure 3.6: Setup Permission Sets page.

2. Click the New button.
3. On the Permission Set Create page, enter “Property Manager” in the Label field. The API Name field should automatically populate with “Property_Manager.” You can enter a description in the Description box, but it is not necessary. See Figure 3.7.



Permission Set
Create

Enter permission set information — ▌ = Required Information

Label	Property Manager
API Name	Property_Manager
Description	
Session Activation Required	☐

Select the type of users who will use this permission set

Who will use this permission set?

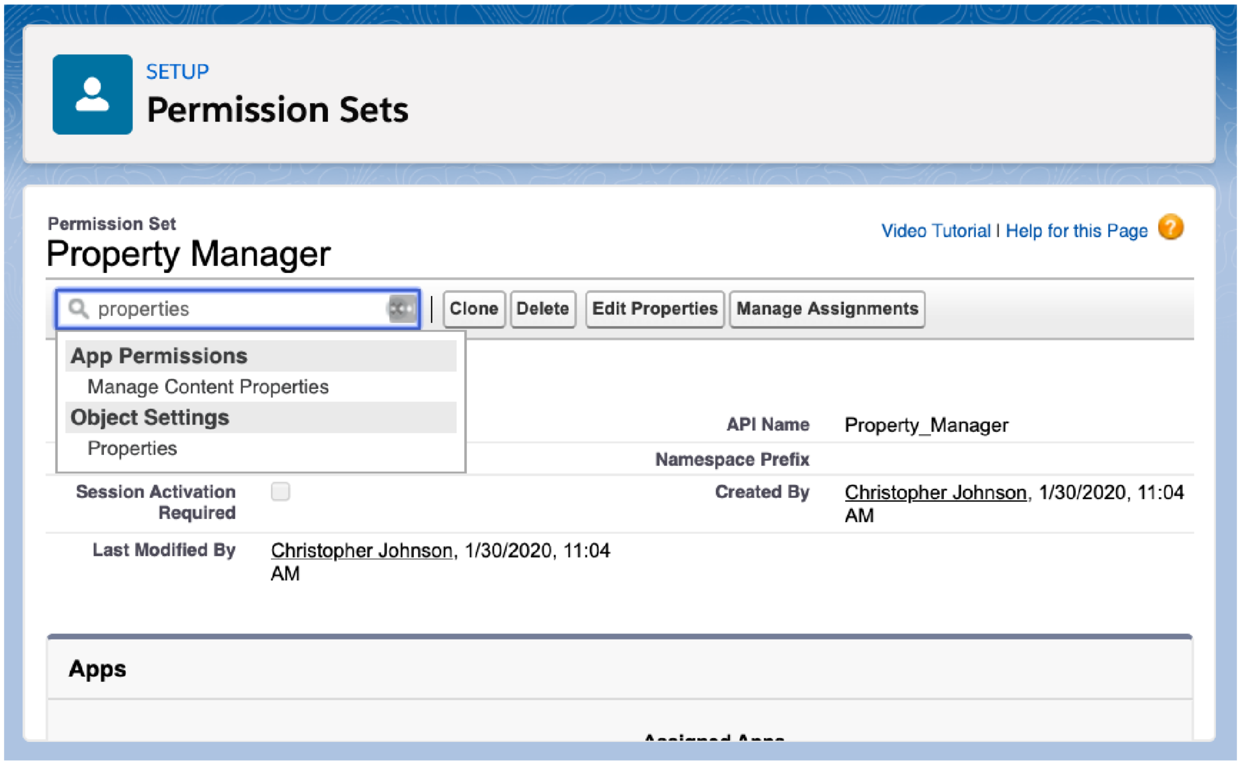
-Choose '--None--' if you plan to assign this permission set to multiple users with different user and permission set licenses.
-Choose a specific user license if you want users with only one license type to use this permission set.
-Choose a specific permission set license if you want this permission set license auto-assigned with the permission set.

Not sure what a permission set license is? Learn more here.

License: --None--

Figure 3.7: Type “Property Manager” in the Label box and leave License set to None.

4. Leave the “License” drop down set to “None.” This option allows ISVs to have more stringent control over exactly which users can be allocated our permission set, should they wish. In the real world, you may wish to choose an option, such as “Salesforce,” to prevent users with more limited licenses from accessing your app. Leaving this as “None” allows administrators at customer instances to provision the licenses to whomever they wish.
5. Click the Save button. You will return to the Setup Permission Sets page.
6. Enter “Properties” into the Find Setting box and then select Object Settings (Figure 3.8).


SETUP
Permission Sets

Permission Set
Property Manager

Video Tutorial | Help for this Page

properties | Clone | Delete | Edit Properties | Manage Assignments

App Permissions
Manage Content Properties
Object Settings
Properties

		API Name	Property_Manager
		Namespace Prefix	
Session Activation Required	☐	Created By	Christopher Johnson, 1/30/2020, 11:04 AM
Last Modified By	Christopher Johnson, 1/30/2020, 11:04 AM		

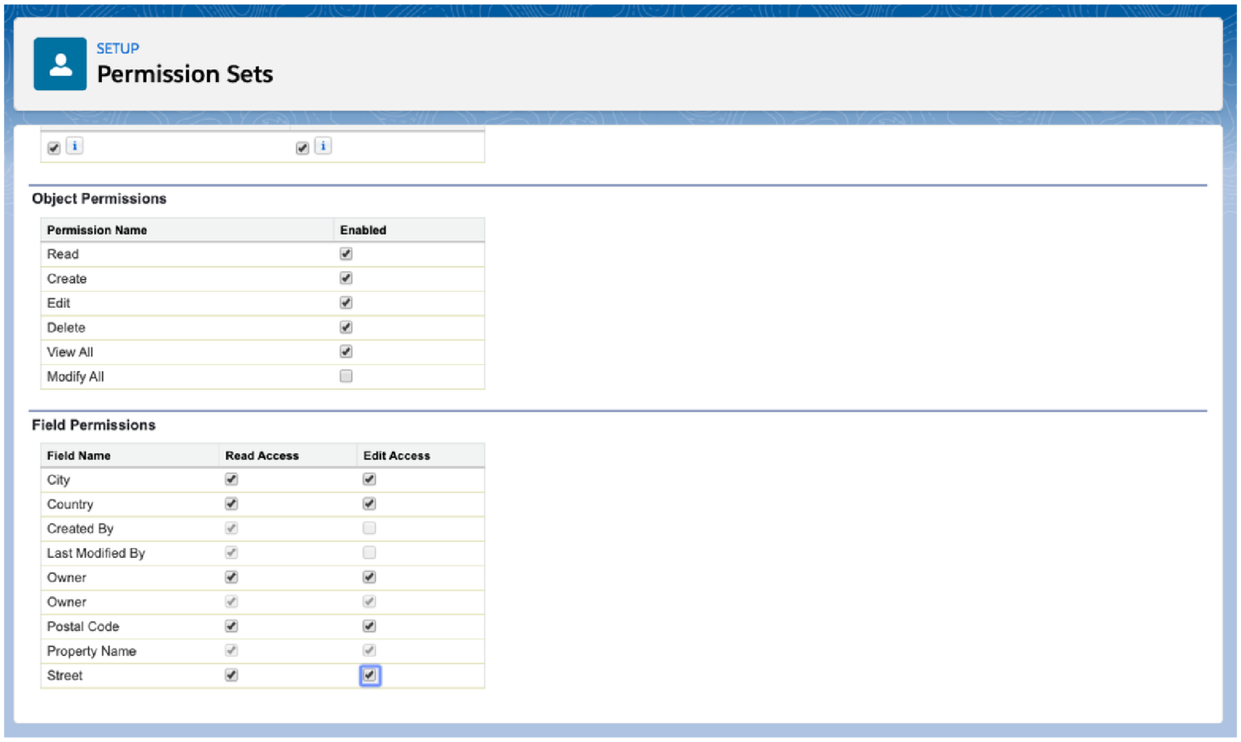
Apps

Figure 3.8: Type “properties” in the Find Settings... box and then select Object Settings.

7. On the App Permissions screen, click the Edit button.
8. Check the boxes next to these permissions:

- Tab Settings: Visible & Available
- Object Permissions: Read, Create, Edit, Delete, View All
- Field Permissions: Edit on City, Country, Post Code, State, Street, Owner

When you have finished, your screen should look like Figure 3.9.


SETUP
Permission Sets

✓	✓

Object Permissions

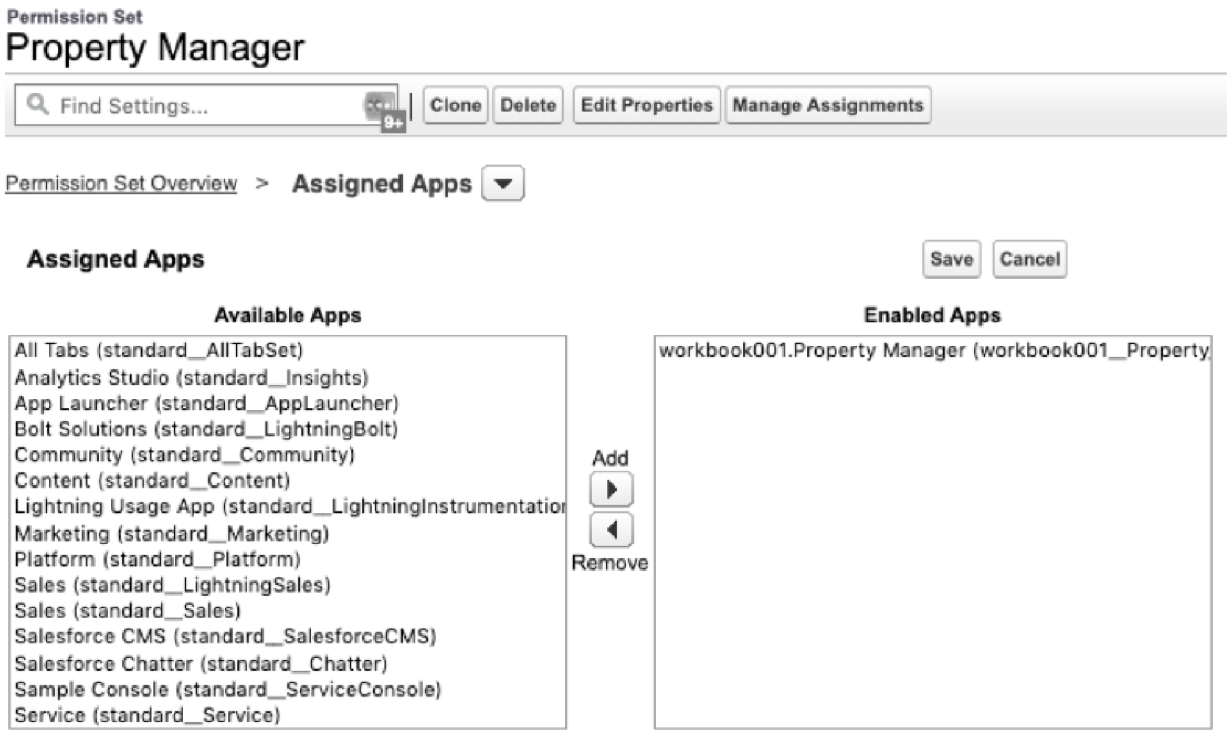
Permission Name	Enabled
Read	✓
Create	✓
Edit	✓
Delete	✓
View All	✓
Modify All	

Field Permissions

Field Name	Read Access	Edit Access
City	✓	✓
Country	✓	✓
Created By	✓	
Last Modified By	✓	
Owner	✓	✓
Owner	✓	✓
Postal Code	✓	✓
Property Name	✓	✓
Street	✓	✓

Figure 3.9: Set permissions for Tab, Object, and Field.

9. Return to the “Property Manager” permission set, select “Assigned Apps,” and click the Edit button.
10. On the Assigned App screen, add “Property Manager” to “Enabled Apps,” as shown in Figure 3.10.



Available Apps
All Tabs (standard__AllTabSet)
Analytics Studio (standard__Insights)
App Launcher (standard__AppLauncher)
Bolt Solutions (standard__LightningBolt)
Community (standard__Community)
Content (standard__Content)
Lightning Usage App (standard__LightningInstrumentatio
Marketing (standard__Marketing)
Platform (standard__Platform)
Sales (standard__LightningSales)
Sales (standard__Sales)
Salesforce CMS (standard__SalesforceCMS)
Salesforce Chatter (standard__Chatter)
Sample Console (standard__ServiceConsole)
Service (standard__Service)

Enabled Apps
workbook001.Property Manager (workbook001__Property

Figure 3.10: Add the Property Manager app to Enabled Apps.

Assign the Permission Set to a User

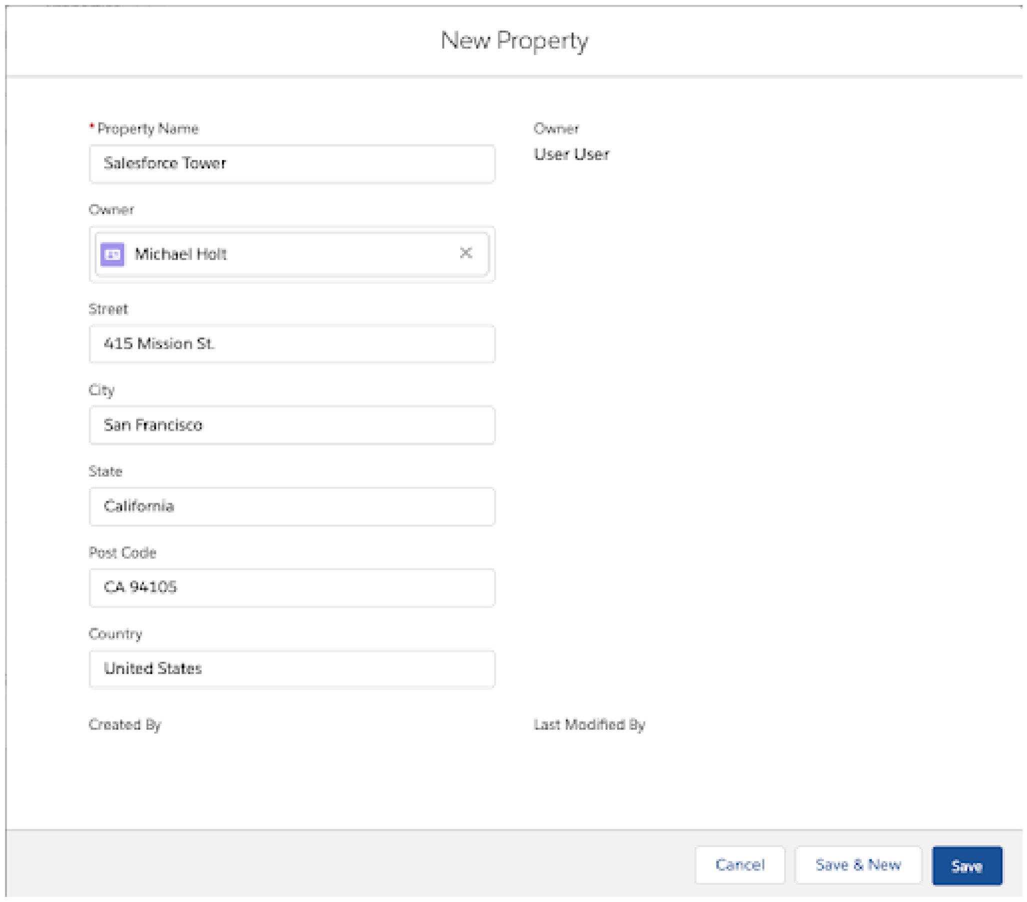
So that we can test that everything is working correctly, let's assign the permission set to a user in this scratch org.

1. On the permission set, click the “Manage Assignments” button and then select “Add Assignments.”
2. Check the box next to “User, User.” This is the default name given to the user who created the scratch org - that's us!
3. Click the Assign button and then Done.

Create a Property Record

To complete this step and ensure everything is going to plan, let's create a property record!

1. Use the App Launcher in the top left corner to open all the apps we have permission to use and you'll see the “Property Manager” app. Select the App and you'll be dropped into the “Properties” tab, which we added to the app earlier.
2. Click “New” and give the property a name and an address. When we go to add a Property Owner, you'll see the option to create a new Contact, that's because we don't have any Contacts in this scratch org.
3. Click Create Contact and create ourselves within Salesforce. Once done, save the Contact. Your screen should look similar to Figure 3.11.
4. Click Save once you're happy with the data.


New Property

* Property Name: Salesforce Tower	Owner: User User
Owner: Michael Holt	
Street: 415 Mission St.	
City: San Francisco	
State: California	
Post Code: CA 94105	
Country: United States	
Created By	Last Modified By

Figure 3.11: A new property record.

Using the power of declarative programming, you have quickly created a custom object–a database table–complete with a user interface.

But that's not all, by creating a custom object in Salesforce, the object is immediately available for API calls, too. you wanted to create these records from an external system, you will be able to do that right away! For this project though, we're going to stick to the platform and extend our property manager natively.

Part 4 | Use Apex for Server Side Development

In order for our customers to visualise the real estate properties they're managing, we want to provide them with a way to locate the properties on Google maps. In order to do that, we'll need to do two things:

1. Build out some logic on the Salesforce server that will query the property location, based on the record the customer is currently viewing.
2. We'll need to show that data via a sleek and modern user interface--a responsive web page that displays the locations on a map for our customers.

To do this, we will use Apex. Apex is a server side development language similar to many other object-oriented programming languages. It enables you to execute flow and transaction control statements on Salesforce servers in conjunction with calls to the API. Apex also enables you to add business logic to most system events, including button clicks, related record updates, and Visualforce pages.

Apex is like Java for Salesforce. It enables you to add and interact with data in the Lightning Platform persistence layer. It uses classes, data types, variables, and if-else statements. You can make it execute based on a condition, or have a block of code execute repeatedly. You should use Apex when you need functionality that is not supported by existing functionality in the Salesforce platform.

Create an Apex Class

Let's get started by returning to VSCode and creating an Apex Class.

1. In VSCode, right click the "classes" folder and then select "SFDX: Create Apex Class" from the pop-up menu (Figure 4.1). The classes folder is under force-app\main\default.
2. In the desired filename box, type "PropertyUtils" and press Enter.
3. At Select classes for the default folder location and press Enter. The result will be a new PropertyUtils class with a default Class definition and Constructor.

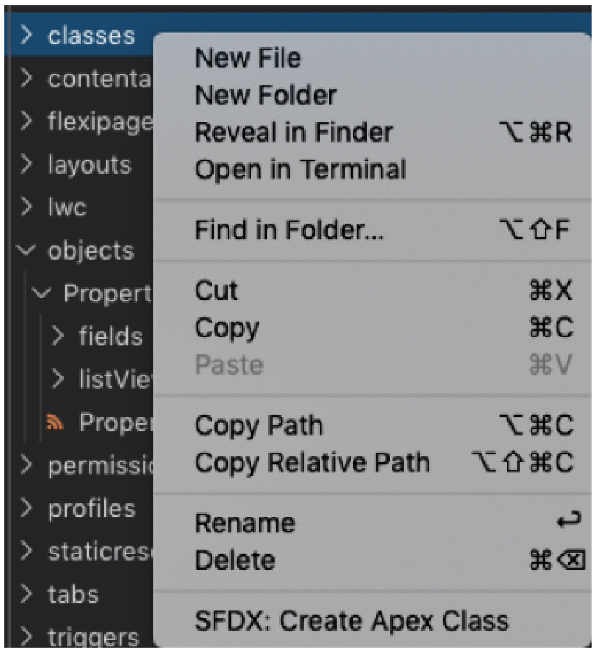


Figure 4.1: To create an Apex Class, right click the classes folder and then select SFDS: Create Apex Class.

4. We won't be needing the constructor public PropertyUtils(), so delete it. We'll create a static method that will be called by our UI component. The The PropertyUtils class should look like this.

```
public with sharing class PropertyUtils {

}
```

The "with sharing" declaration means that our class will respect the sharing rules defined within our customers' org. This is exactly what we want to do for the majority of the Apex code we write in our applications.

Write an Apex Method

Next, we'll create a new Apex method called "getPropertyAddress," which is going to take a record ID as a parameter and use a method to query the address fields related to that record. We'll do that using the Salesforce Object Query Language (SOQL), like so:

```
public static void getPropertyAddress(id recId) {
workbook001__Property__c prop = [SELECT workbook001__street__c, workbook001__city__c,
system.debug(prop);
}
```

The __c after the field names indicate that you created custom fields. Standard fields that are provided by default in Salesforce are accessed using the same type of dot notation but without the __c.

Note: Remember to replace the namespace prefix "workbook001" with your own namespace. Once done, run the following command in your terminal (replacing "workbookScratch" with your own scratch org name, if different, or removing it completely if you set the org as the default earlier):

`sfdx force:source:push -u workbookScratch`

Check Your Work with the Salesforce Developer Console

The Salesforce Developer Console provides tools for developing your components and app. You can check your work so far by opening up the Developer Console in your scratch org and using anonymous Apex to pass in a record ID for our Property object.

1. Open up your scratch org and navigate to the record we created earlier. In the URL for the record, we'll be able to get the ID. It should look something like this:

https://drive-force-3622-dev-ed.lightning.force.com/lightning/r/workbook001__Property__c/a002500000AFc4oAAD/view

2. Now open up the developer console. Click on the cog icon in the top right hand corner of Salesforce and then click the Developer Console option as shown in Figure 4.2.

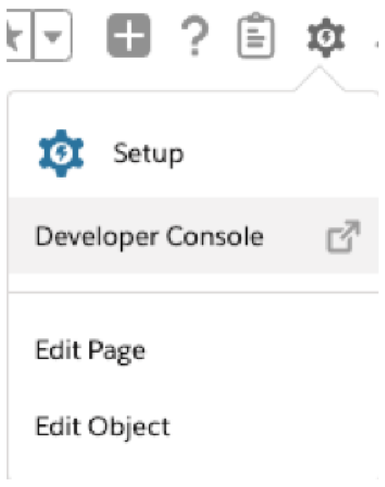
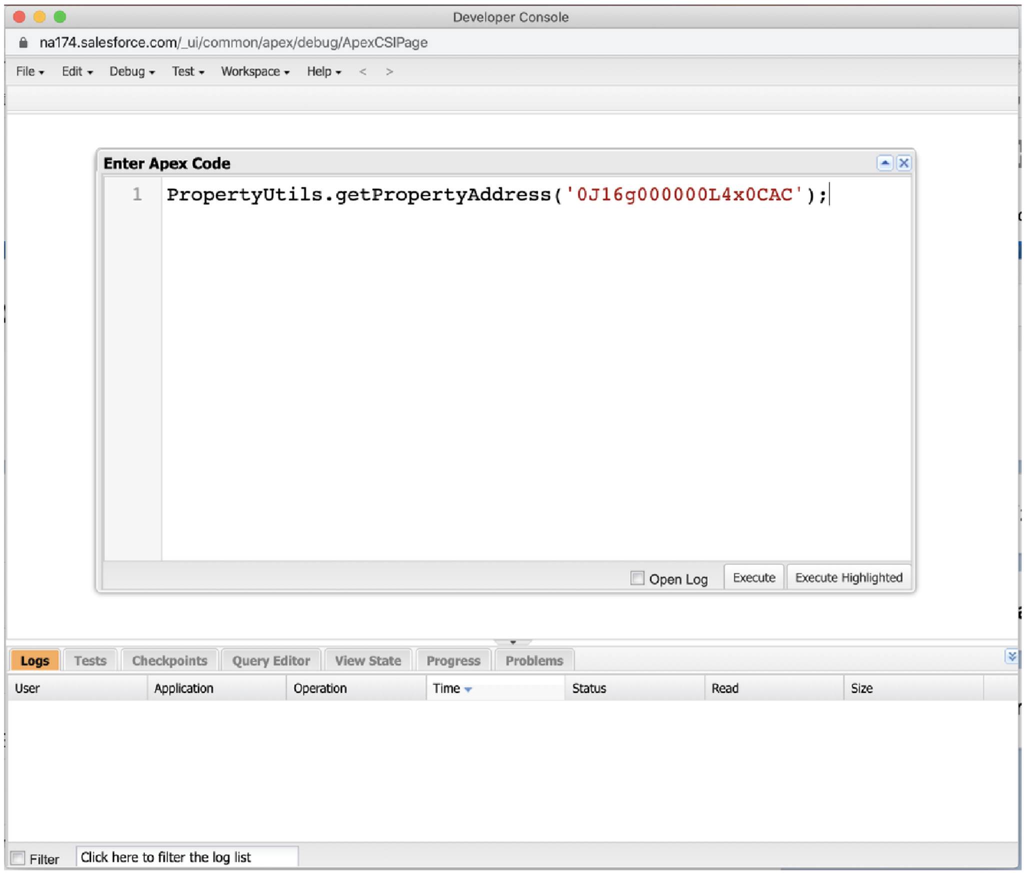


Figure 4.2: To open the Developer Console, click the cog in the upper right corner of the page and then click Developer Console from the menu.

3. Open Debug > Open Execute Anonymous Window (Figure 4.3). A new development window will open and we can enter some simple code, let's enter the following snippet, replacing the ID with the ID you've copied from the URL in the step above:

```
PropertyUtils.getPropertyAddress('a002500000AFc4oAAD');
```


Developer Console
na174.salesforce.com/_ui/common/apex/debug/ApexCSIPage
File Edit Debug Test Workspace Help < >
Enter Apex Code
Open Log
Execute
Execute Highlighted
Logs Tests Checkpoints Query Editor View State Progress Problems
User Application Operation Time Status Read Size
Filter
Click here to filter the log list

```
PropertyUtils.getPropertyAddress('0J16g000000L4x0CAC');
```

Figure 4.3: The Developer Console.

4. A new Log will appear in your developer console and double-clicking the log will bring up the Apex execution.There's a lot of information here, so let's check the "Debug Only" checkbox to see our debug line.
5. Double click the line that's shown in the Execution log and see the record we've queried - it works!

```
12:42:11:020 USER_DEBUG [5]|DEBUG|workbook001__Property__c:{workbook001__Street__c=415 Mission St,
workbook001__City__c=San Francisco, workbook001__State__c=California, workbook001__Country__c=United
States, workbook001__Post_Code__c=CA 94105, Id=a002500000AFc4oAAD}
```

Create a Data Structure

Now, when we implement our map, we'll need to provide the data in a structure that the base Lightning Web Component that we're going to use (lightning-map), is going to accept. We'll get into more detail on Lightning Web Components later, but for now, know that we can use the Lightning Web Component library to see the base component specifications, documentation and even an example implementation here.

If we look at the Documentation, we can see that we need to pass map-markers the following information:

- location (an object, consisting of...)
    - City
    - Country
    - PostalCode
    - State
    - Street
    - Or a set of coordinates *(we'll be using the fields above, since we already created input fields for them on our Property Object!)*
- title
- description
- icon

Since there's no standard structure within Salesforce to hold this data structure, we'll need to create it. We'll do that by creating a couple of inner Classes, inside the "PropertyUtils" class.

1. Let's start with that Location data structure and give it a Constructor to help us populate it, like so:

```
public class Location{
 @AuraEnabled
 public String Street;
 @AuraEnabled
 public String PostalCode;
 @AuraEnabled
 public String City;
 @AuraEnabled
 public String State;
 @AuraEnabled
 public String Country;

 public Location(String strt, String cit, String stat, String pc, String ctry){
  Street = strt;
  City = cit;
  State = stat;
  PostalCode = pc;
  Country = ctry;
 }
 }
```

2. We've marked these as "AuraEnabled," and we'll come back to why that is later on. For now, let's move on and create a structure that Map-Marker can consume. It is going to contain the Location as one of its properties.

```

public class MapMarker{
 @AuraEnabled
 public String icon;
 @AuraEnabled
 public String title;
 @AuraEnabled
 public String description;
 @AuraEnabled
 public Location location;

 public MapMarker(String i, String t, String d, Location l){
  icon = i;
  title = t;
  description = d;
  location = l;
 }
 }
```

Great. Now we have the structure for an object that map-marker can accept, all that's left is for us to create an instance of a MapMarker and give it the address data from the record we queried earlier.

3. We also need to provide the icon and description data. For this example we're not using those so I'll populate this with dummy data. Lastly, we need our method to give this data to the UI component that's going to call it, so let's have the method return a list of MapMarkers (although we're only returning one at this point, it'll make our lives easier later). Your method should end up looking like this:

```
public static List<MapMarker> getPropertyAddress(id recId) {
 workbook001__Property__c prop = [SELECT Name, workbook001__street__c, workbook001__ci

 Location loc = new Location(prop.workbook001__street__c, prop.workbook001__city__c, p
 MapMarker mark = new MapMarker('icon',prop.Name, 'description', loc);
 List<MapMarker> locs = new List<MapMarker>{mark};
 return locs;
 }
```

4. Push this to the scratch org using the following command in the terminal:

```
sfdx force:source:push -u workbookScratch
```

5. Once again, we can test this by jumping into the developer console and calling the method with our record. This time, let's write the debug line in the execute anonymous window, since I've removed that from my method for cleanliness. Enter the following (once again replacing the ID with the ID of your own record):

```
system.debug(PropertyUtils.getPropertyAddress('a002500000AFc4oAAD'));
```

6. Follow the same process as before to view our debug line:

```
13:16:33:025 USER_DEBUG [1]|DEBUG|(MapMarker:[description=description, icon=icon, location=Location:
[City=San Francisco, Country=United States, PostalCode=CA 94105, State=California, Street=415
Mission St], title=title])
```

Excellent! Our server side Apex code is able to retrieve the address details, given a Property record, and then provide the results in a structure that can be interpreted by a Lightning Web Component, but there's one last piece we're missing.

Use @AuraEnabled Annotation to Expose Lightning Components

In order for an Apex method to be accessed by a component, the method needs to make itself reachable by the Aura framework, something we won't be going into detail on here. We do this by applying the @AuraEnabled annotation to our method. What we'd also like to do to help our users utilise our apps quickly, is to cache the data that's returned by this method, using the optional (cacheable=true) parameter. The final result of our method will look like this:

```
@AuraEnabled(cacheable=true)
 public static List<MapMarker> getPropertyAddress(id recId) {
 workbook001__Property__c prop = [SELECT workbook001__street__c, workbook001__city__c,

 Location loc = new Location(prop.workbook001__street__c, prop.workbook001__city__c, p
 MapMarker mark = new MapMarker('icon', 'title', 'description', loc);
 List<MapMarker> locs = new List<MapMarker>{mark};
 return locs;
 }
```

If you have any doubts on the above, refer to the full Class, here:

```
public with sharing class PropertyUtils {

 @AuraEnabled(cacheable=true)
 public static List<MapMarker> getPropertyAddress(id recId) {
 system.debug(recId);
 workbook001__Property__c prop = [SELECT Name, workbook001__street__c, workbook001__ci

 Location loc = new Location(prop.workbook001__street__c, prop.workbook001__city__c, p
 MapMarker mark = new MapMarker('icon', prop.Name, 'description', loc);
 List<MapMarker> locs = new List<MapMarker>{mark};
 return locs;
 }


 public class MapMarker{
 @AuraEnabled
 public String icon;
 @AuraEnabled
 public String title;
 @AuraEnabled
 public String description;
 @AuraEnabled
 public Location location;

 public MapMarker(String i, String t, String d, Location l){
  icon = i;
  title = t;
  description = d;
  location = l;
 }
 }

 public class Location{
 @AuraEnabled
 public String Street;
```

```
  @AuraEnabled
  public String PostalCode;
  @AuraEnabled
  public String City;
  @AuraEnabled
  public String State;
  @AuraEnabled
  public String Country;

   public Location(String strt, String cit, String stat, String pc, String ctry){
   Street = strt;
   City = cit;
   State = stat;
   PostalCode = pc;
   Country = ctry;
  }
  }
 }
```

Unit Testing Your Apex Code

In order to publish AppExchange apps, Apex code needs test coverage. The solution to this is to write Apex Unit Tests. Apex Unit Test are classes which exist purely to test the outcomes of our Apex methods and help to ensure that future changes don't introduce bugs. If we attempt to upload a package which has under 75% Apex code coverage, then we'll be blocked from uploading it. We currently have 0% test coverage, so let's correct that by creating another Apex class!

Create Apex Unit Tests

To create a class for Apex Unit Tests:

1. Right click the "classes" folder in our project and select "SFDX: Create Apex Class."
2. In the text box , enter "PropertyUtils_Test" and hit Enter. On the final prompt, choose the default location and press Enter.
3. At the top of the file, above the class, write "@isTest." @isTest indicates that PropertyUtils_Test is a test class.

`@isTestpublic class PropertyUtils_Test {`

Create a Test Setup Method

In the context of a test method, a dummy Salesforce database is provided with no data whatsoever. There is a means of accessing the live data, but this is generally considered bad practice, since we can't guarantee what data will be available to us (particularly in the context of a managed package, where customer data will vary).

The test should be self-contained and we should be able to predict its outcome. For that reason, we'll introduce a Test Setup method, which will execute before all other test methods in the class and provide us with the data we need. Within this setup method, we'll create an instance of our property record and insert it into the empty test database, so we have a record that our test methods can use. Remember to replace the namespace with your own:

```
  @testSetup
  static void propertyUtilsTestSetup(){
  workbook001__property__c property = new workbook001__property__c(
   workbook001__street__c = '415 Mission St',
   workbook001__City__c = 'San Francisco',
   workbook001__Post_code__c = 'CA 94105',
   workbook001__Country__c = 'United States');
  insert property;
  }
```

To ensure the correct behaviour occurs in a wide variety of scenarios, unit tests should cover a range of scenarios for our Apex code. Because this is a demo application that is designed to prove the concept of Apex unit testing, we're only going to create a single test that ensures that our method provides us with the output we'd expect.

We'll start by querying the database to retrieve the record inserted in our test method. We'll then run the test case, by calling the method we wrote above. We'll then assert that the response provided by the method is in line with our expectations:

```
@isTest
static void getPropertyAddressTest(){
workbook001__property__c property = [SELECT id FROM workbook001__property__c][0];
test.startTest();
 propertyUtils.MapMarker result = PropertyUtils.getPropertyAddress(property.id)[0];
test.stopTest();

system.assert(result.location.Street == '415 Mission St');
system.assert(result.location.City == 'San Francisco');
system.assert(result.location.PostalCode == 'CA 94105');
system.assert(result.location.Country == 'United States');
}
```

Run Tests in the Developer Console

We can run these tests from VSCode, but instead, let's open the developer console to get some insights.

1. Open the developer console by clicking the cog within Salesforce and selecting Developer Console.
2. Select Test > New Run.
3. In the window which opens, click on our new test class.
4. Next, click on the checkbox beside our test method and select Run (Figure 4.4).

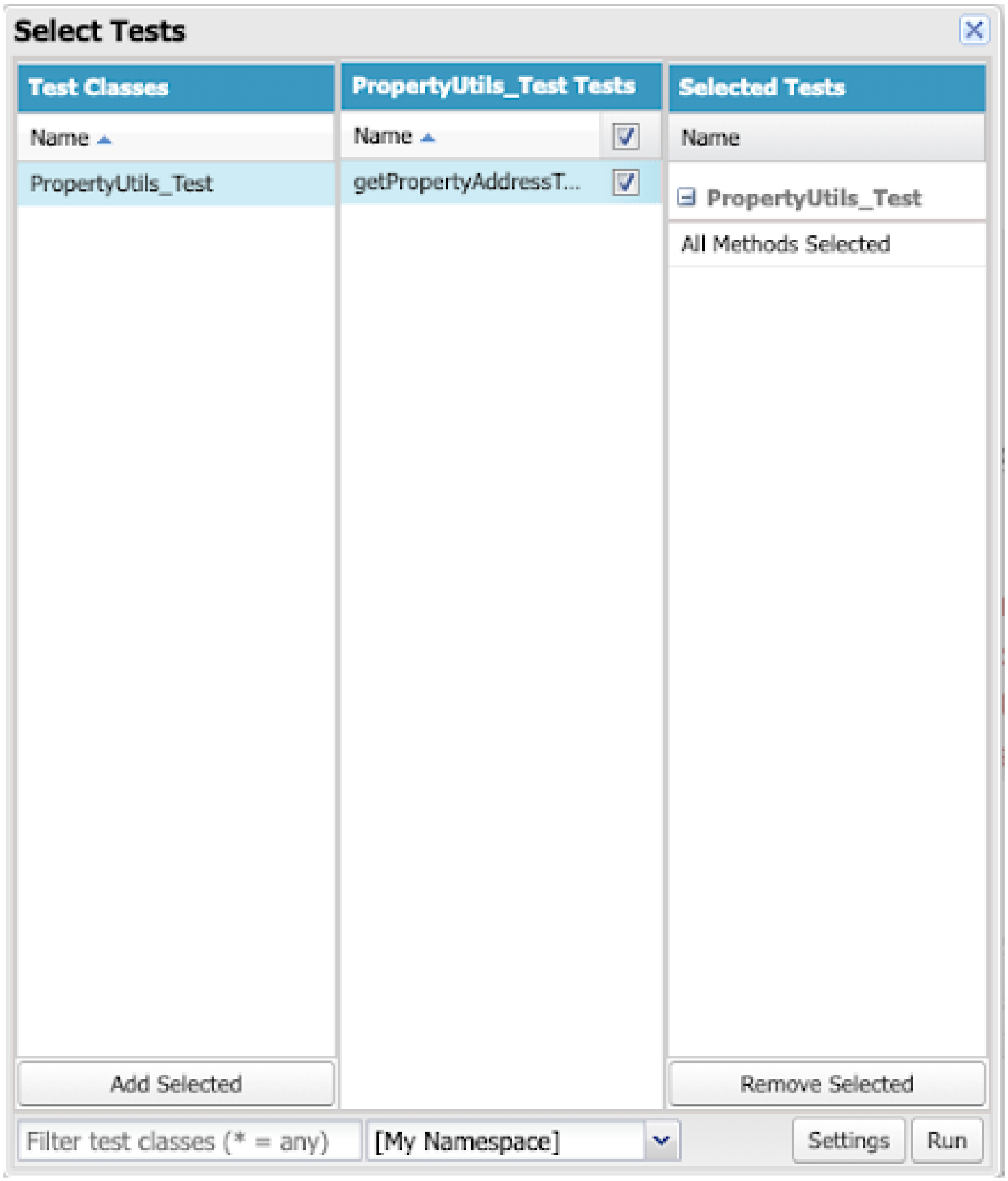
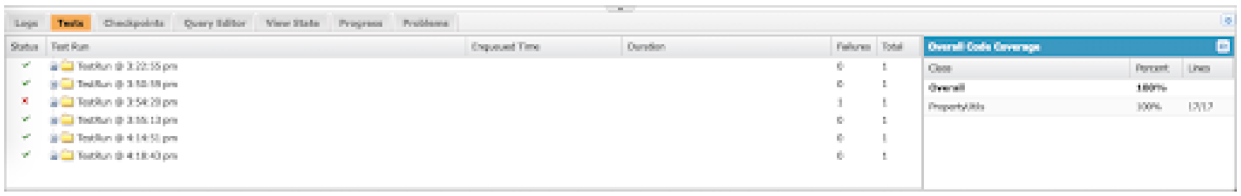


Figure 4.4: In the Developer Console, check the test method you want to run and then click the Run button.

5. Under the "Test" tab, at the bottom of the developer console, you'll find that your test ran successfully. Try altering some of the values in the assertion and run the test again, you'll see that it then fails. All tests need to pass in order to ensure a successful upload of our package later on.

We mentioned that 75% test coverage is required in order for us to upload our package. Fortunately, we can check that value in the Developer Console too! Look to the bottom right of the Test tab and you'll find the overall test coverage (Figure 4.5).



Status	Test Run	Enqueued Time	Duration	Failures	Total
✓	TestRun @ 3:22:55 pm			0	1
✓	TestRun @ 3:50:59 pm			0	1
✗	TestRun @ 3:54:20 pm			1	1
✓	TestRun @ 3:56:13 pm			0	1
✓	TestRun @ 4:14:51 pm			0	1
✓	TestRun @ 4:18:40 pm			0	1

Overall Code Coverage

Class	Percent	Lines
Overall	100%	
PropertyUtils	100%	17/17

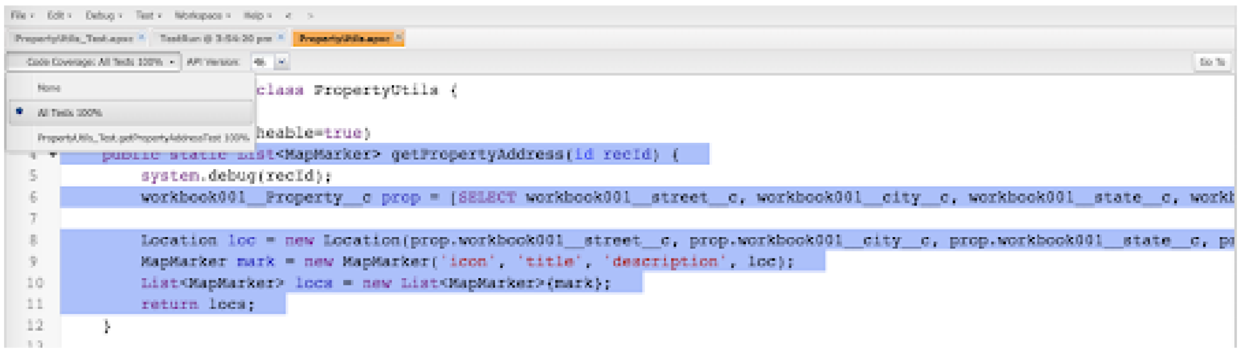
Figure 4.5: Look at the bottom right of the Developer Console to see overall test coverage.

Since we have a single Class within our project, we can see 100% for the Class and 100% for the overall coverage. As you can imagine, with multiple Classes this gives us a great window into which Classes need unit testing in order for our package to be as reliable as possible.

View Tested Code Line-by-Line

The last thing to be aware of within the Developer Console is the ability to see the granular line-by-line view of code coverage.

1. In the Developer Console, go to File > Open > Classes > Property Utils and then select "Open."
2. Select the "Code Coverage" drop down in the top left corner and then select one of the 2 options (Not "None"). The Developer Console will highlight our code coverage for us and give us a view into exactly which lines have been covered (Figure 4.6). For uncovered lines, the Developer Console would highlight these in red. Fortunately, we have 100% coverage so we can be sure our package will not fail for this reason!



```
                  class PropertyUtils {

                  heable=true)
    public static List<MapMarker> getPropertyAddress(id recId) {
        system.debug(recId);
        workbook001__Property__c prop = [SELECT workbook001__street__c, workbook001__city__c, workbook001__state__c, workb

        Location loc = new Location(prop.workbook001__street__c, prop.workbook001__city__c, prop.workbook001__state__c, p
        MapMarker mark = new MapMarker('icon', 'title', 'description', loc);
        List<MapMarker> locs = new List<MapMarker>{mark};
        return locs;
    }
```

Figure 4.6: The Developer Console highlights the code that is covered.

So, now that we've built and tested the backend, we better show the data to our users, using a Lightning Web Component!

Part 5 | Create a User Interface Using Lightning Web Components

Lightning Web Components, or LWCs, are the latest in modern, responsive, custom user interfaces built on the Salesforce platform. Based on the web standards of today, LWCs consist primarily of HTML, Javascript and CSS and can be leveraged to build extremely complex and intuitive user experiences on the platform. You've already seen a glimpse of these if you opened the lightning-map example in our documentation earlier on.

Build a User Interface

In the previous section we developed a backend method, using Apex, to provide a frontend with the data it needs to output a location on a map. In this section, we'll build that frontend component!

So let's dive into VSCode and start building.

1. As we did with our Apex Class previously, let's right click on the lwc folder in VSCode and select SFDX: Create a Lightning Web Component (Figure 5.1).
2. In the text box, type "propertyMap," which is what we'll call the component Leave the containing folder as the default and press Enter.

In the lwc folder, you will see a new folder called propertyMap containing 3 files that we'll use to build our user interface (Figure 5.2):

propertyMap.html
propertyMap.js
propertyMap.js-meta.xml

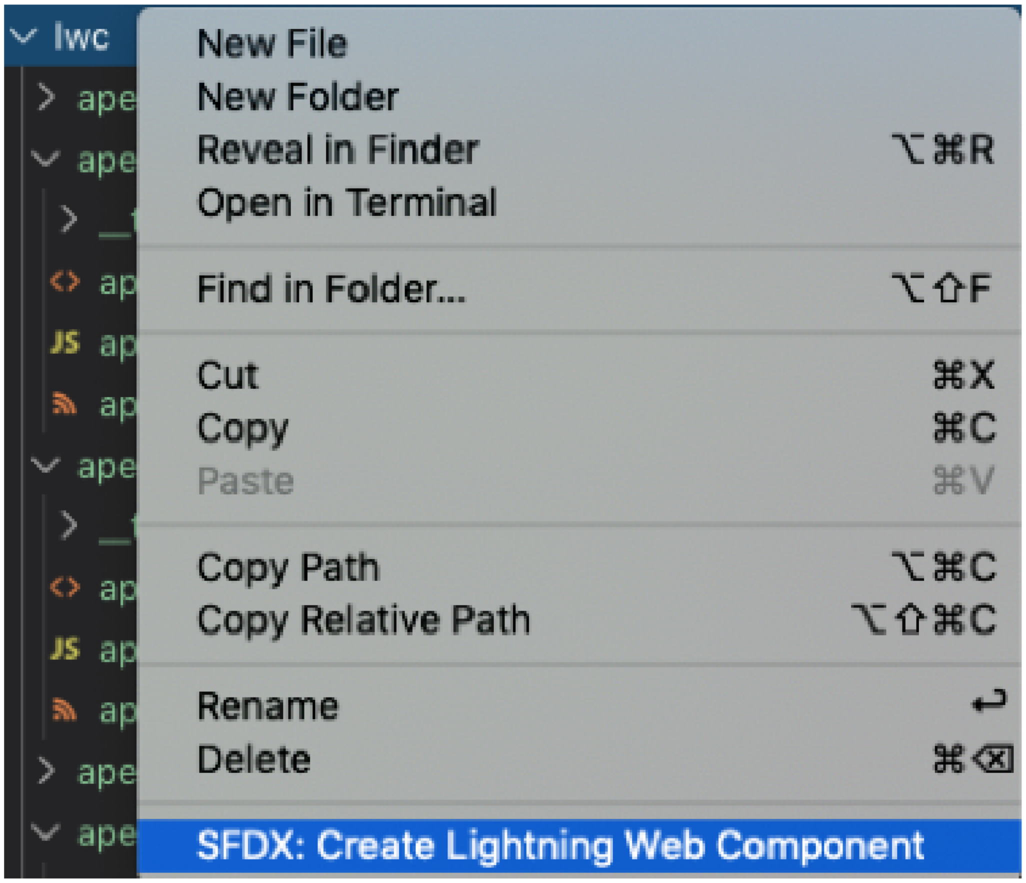


Figure 5.1: Right click the lwc and select SFDX: Create Lightning Web Component.

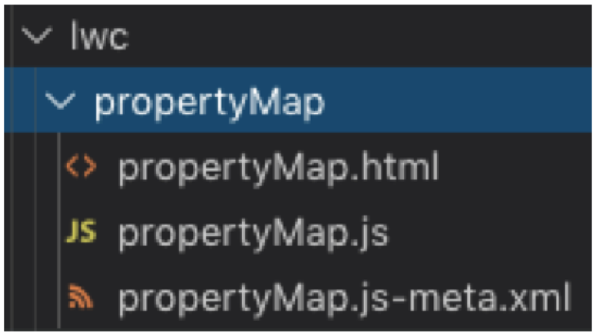


Figure 5.2: New Lightning Web Component with 3 files.

3. Let's start by opening the html file. Currently, it should consist only of a template opening and closing tag. Let's put some dummy text in there, "Hello World!" feels appropriate.
4. Next, open the propertyMap.js-meta.xml and you'll see some default data:

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata" fqn="propertyMap"> <apiVer
sion>46.0</apiVersion> <isExposed>false</isExposed></LightningComponentBundle>
```

5. Change the isExposed value to true. This exposes the file to the Lightning App Builder, which is the tool we'll be using to customize our user interface.
6. Next, we need to indicate on which pages our component should be available; record pages, app pages or home pages. Since our component requires the ID of the record our user is looking at, enter "lighting__RecordPage" as the target value for our component:

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata" fqn="propert
 <apiVersion>46.0</apiVersion>
 <isExposed>true</isExposed>
 <targets>
 <target>lightning__RecordPage</target>
 </targets>
</LightningComponentBundle>
```

7. Push this to your scratch org, once again using the following command and open the record we created earlier.

```
sfdx force:source:push -u workbookScratch
```

8. Now, we'll need to use the Lightning App Builder to add our component to the record page. We can do this by going to the cog in the top right hand corner and selecting "Edit Page." This tool allows us to completely customise the view of our Property record screen, using Standard and Custom components.
9. Let's customise by adding our brand new component. Scroll to the "Custom" section of the Lightning Components pane and drag our newly created component from the left hand pane of the component, as shown in Figure 5.3.

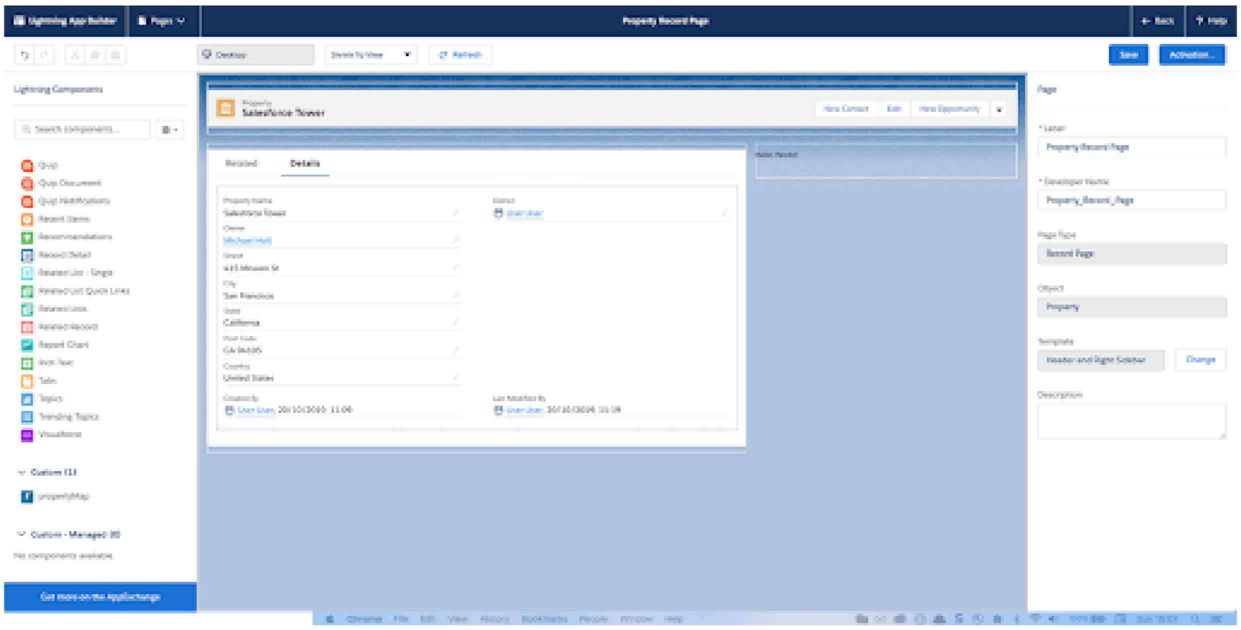
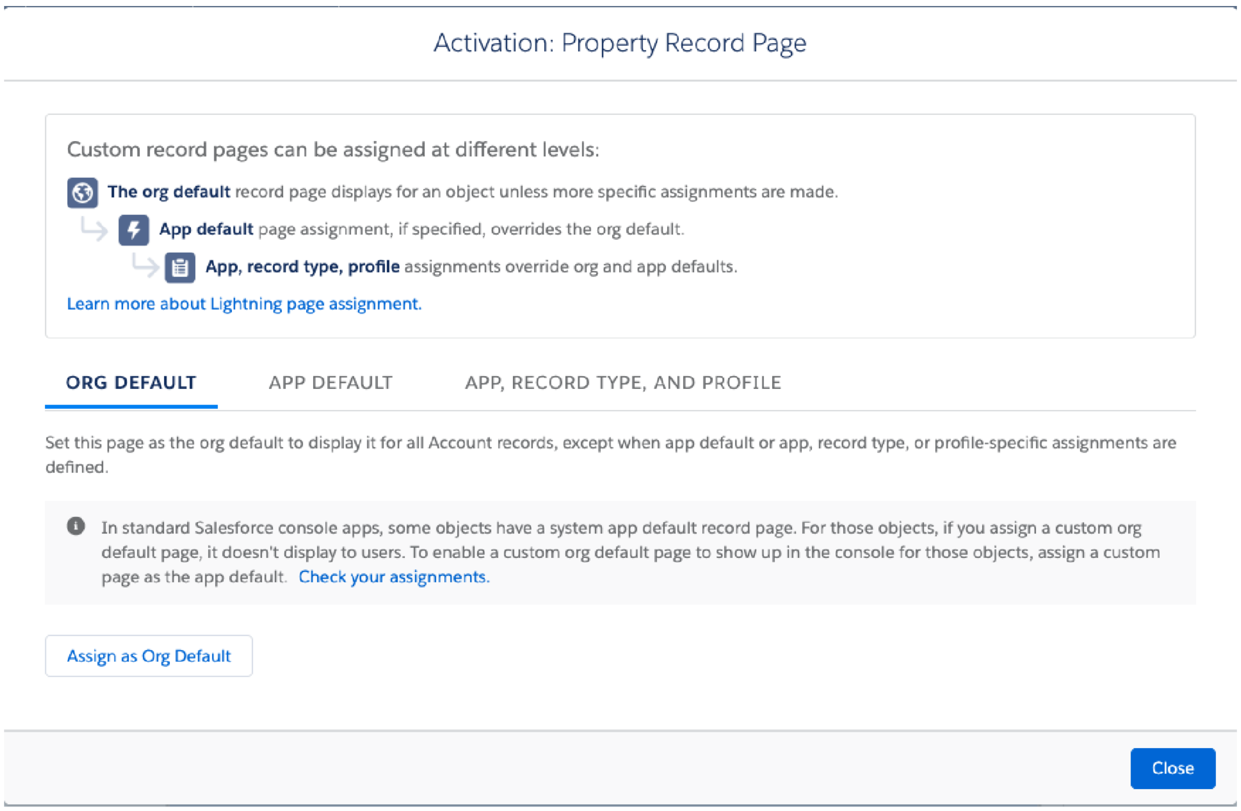


Figure 5.3: Drag the custom component into the right-hand column of the page.

10. Save the page.
11. On the Activation: Property Record Page, click the Assign as Org Default button. This sets the current page to be the default for all the Property records in our app (Figure 5.4).



Activation: Property Record Page

Custom record pages can be assigned at different levels:

The org default record page displays for an object unless more specific assignments are made.

App default page assignment, if specified, overrides the org default.

App, record type, profile assignments override org and app defaults.

Learn more about Lightning page assignment.

ORG DEFAULT APP DEFAULT APP, RECORD TYPE, AND PROFILE

Set this page as the org default to display it for all Account records, except when app default or app, record type, or profile-specific assignments are defined.

In standard Salesforce console apps, some objects have a system app default record page. For those objects, if you assign a custom org default page, it doesn't display to users. To enable a custom org default page to show up in the console for those objects, assign a custom page as the app default. Check your assignments.

Assign as Org Default

Close

Figure 5.4: To set the current page as the default for all Property records, click the Assign as Org Default button.

12. Click save on the confirmation page and then use the Back button to return to the record screen. Hey presto! We can now see "Hello World!" on the record page.

Whilst this is great in itself, our users are going to want a little bit more. Let's pull that data from the Apex method we created and show our users an intuitive map on which they can view a property's location.

Use JavaScript to Display Map

We'll need to use JavaScript to do that, so go ahead and open the propertyMap.js file. At the moment, we're just importing a custom wrapper of the standard HTML element, called LightningElement and then extending it to create a JavaScript class. We export this class so that it can be used by other components within our app (components within components is a key concept of lightning component development!).

We know that we'll need the ID of the record we're currently looking at. Fortunately, Salesforce provides us with a super easy way to retrieve that using @api recordId. The @api decorator defines the property as public, meaning that any parent components in this component tree can read this value.

Before we can use @api, we need to import it, alongside LightningElement. So let's add that to our JavaScript.
Our component now has visibility of our record ID. If we wanted to test this, we could output recordId using console.log and push it to our scratch org, but let's press on with the development.

```
import {LightningElement, api} from 'lwc';

export default class LightningExampleMapSingleMarker extends LightningElement {
 @api recordId;
}
```

Access the Apex Method with @wire

Our next step is to access our Apex method and we're going to do that using another annotation; @wire. @wire uses a reactive wire service which Salesforce developers can use to build components that can gather data from Salesforce in a variety of different ways; the adapter can read object metadata, pull data directly from records without using Apex, retrieve lists of records using listview IDs and much much more. What we can also use @wire for is calling Apex methods, which is what we'll need to do here.

1. Firstly, we need to import our Apex method into our component, like so:

```
import getPropertyAddress from '@salesforce/apex/PropertyUtils.getPropertyAddress';
```

2. We'll then need to define a variable which can store the response from our Apex Method, let's call this "propertyLocs" and import and use another annotation @track. This annotation allows us to rerender the component if it's value changes, as @api does, but tacked properties are private and cannot be read or set by other components. Our JavaScript file should now look like this:

```
import {LightningElement, api, wire, track} from 'lwc';
import getPropertyAddress from '@salesforce/apex/PropertyUtils.getPropertyAddress';


export default class LightningExampleMapSingleMarker extends LightningElement {
 @api recordId;
 @track propertyLocs;
}
```

3. We'll now need to implement the JavaScript function to read the data from our method. We'll start by defining the method to call using the wire adapter:

`@wire(getPropertyAddress, {recId: '$recordId'})And beneath it, define our method, called "wiredProperties" wiredProperties({error,data}) {`

4. If our method call returns data, then let's push the response into propertyLocs, which we defined earlier. We can also turn the response into a human-readable String and output it to the console, so we can be sure the function is working.

```
 @wire(getPropertyAddress, {recId: '$recordId'})
 wiredProperties({error,data}) {
 if (data) {
  this.propertyLocs = data;
  console.log(JSON.stringify(data, null, '\t'));
```

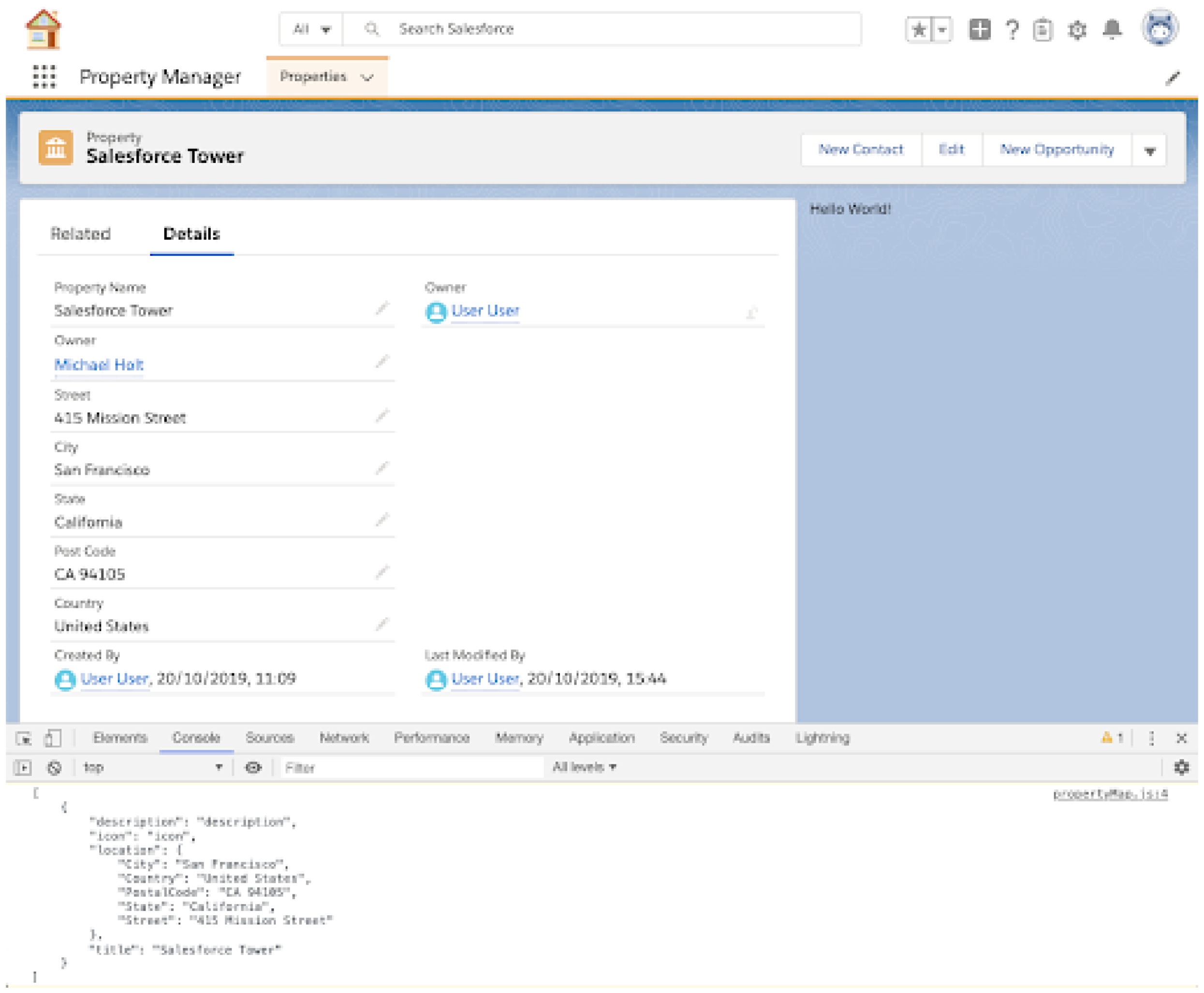
5. Lastly, should our wire adapter bring back any errors, we can push those into a tracked "error" property which, in a real world scenario, we may want to expose to a user or take some other action off the back of. We won't be handling this error here, but let's at least log it to the Developer Console so we have a view of any problems which may occur. The final resulting JavaScript file should look like so:

```

import {LightningElement, api, wire, track} from 'lwc';
import getPropertyAddress from '@salesforce/apex/PropertyUtils.getPropertyAddress';


export default class PropertyMap extends LightningElement {
 @api recordId;
 @track propertyLocs;
 @track error;


 @wire(getPropertyAddress, {recId: '$recordId'})
 wiredProperties({error,data}) {
 if (data) {
  this.propertyLocs = data;
  console.log(JSON.stringify(data, null, '\t'));
 } else if (error) {
  console.log('error');
  console.log(JSON.stringify(error, null, '\t'));
  this.error = error;
 }
 }
}
```

6. Let's push this to our Scratch org and open our test record:

`sfdx force:source:push -u workbookScratch`

We can see our JavaScript in action using the Developer Console within the browser. I'm using Google Chrome, but no matter your browser, so long as it's supported by Salesforce, you should be able to see something similar to Figure 5.5.



Property Name: Salesforce Tower
Owner: Michael Holt
Street: 415 Mission Street
City: San Francisco
State: California
Post Code: CA 94105
Country: United States
Created By: User User, 20/10/2019, 11:09
Owner: User User
Last Modified By: User User, 20/10/2019, 15:44

```
[
  {
    "description": "description",
    "icon": "icon",
    "location": {
      "City": "San Francisco",
      "Country": "United States",
      "PostalCode": "CA 94105",
      "State": "California",
      "Street": "415 Mission Street"
    },
    "title": "Salesforce Tower"
  }
]
```

Figure 5.5: JavaScript communicating with Apex, as seen in Salesforce Developers Console.

Fantastic! Our JavaScript is communicating with our Apex Class and able to retrieve the data from the record we're looking at. Of course, we can't expect our users to open the developer console in their browsers, so let's implement that final step to get this data onto their screens!

Open your HTML file and remove the "Hello World!" that we entered earlier. You'll remember we're using the lightning-map base LWC and if we review the documentation, we only have 1 required parameter; map-markers. Let's implement lightning-map and associate the map-markers parameter with our propertyLocs property. Our HTML file should look like this:

```
<template>
 <lightning-map
  map-markers={propertyLocs}>
 </lightning-map>
</template>
```

Unfortunately, if we push this to our scratch org we'll see an error message when we try to load the page. Go ahead and try it:

```
sfdx force:source:push -u workbookScratch
```

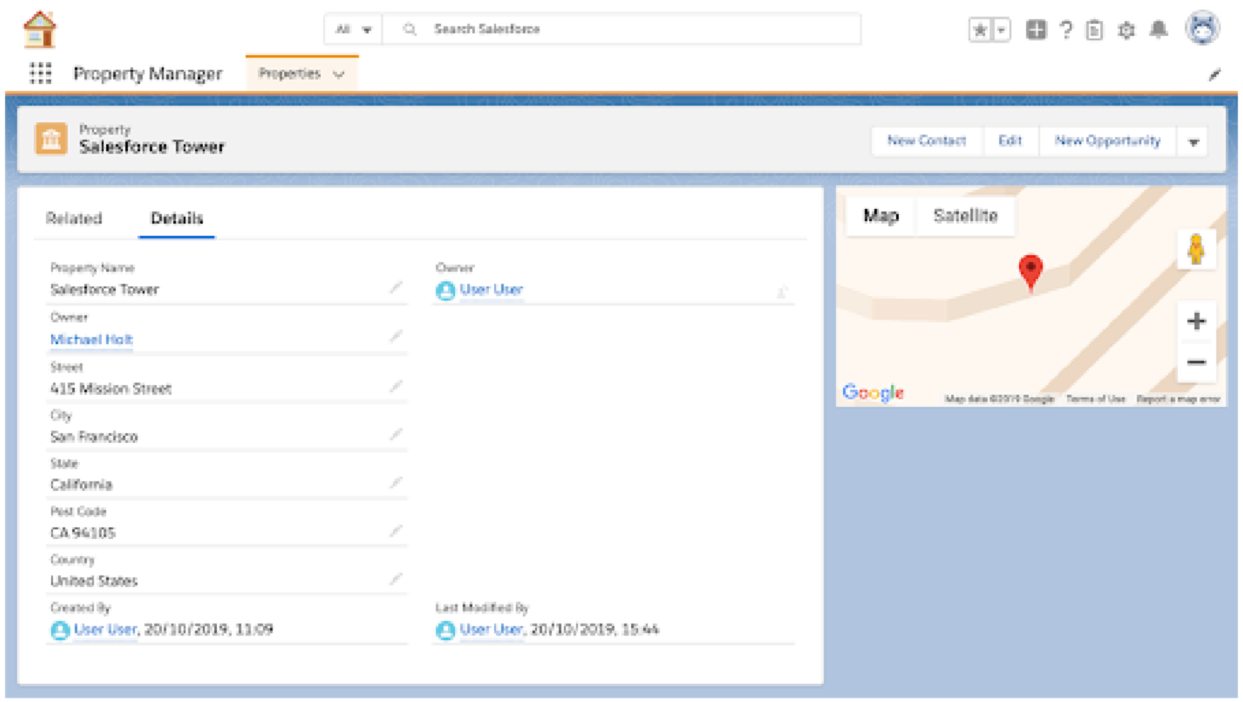
This is happening because when the *page* initially loads, propertyLocs has no value at all and when the *component* initially loads propertyLocs is empty. This is okay, since it only takes a second for our wire adapter to do its work and, remember we made propertyLocs tracked?

This means the component will re-render the second we retrieve the value. What we need to do is prevent the component from trying to display the map before propertyLocs has a value. We'll do that using a second HTML template, assess the value of propertyLocs and render the map only when it has a value, like so:

```
<template>
 <template if:true={propertyLocs}>
 <lightning-map
  map-markers={propertyLocs}>
 </lightning-map>
 </template>
</template>
```

Let's push this to our org, once again using the following command:

```
sfdx force:source:push -u workbookScratch
```



Property Name
Salesforce Tower

Owner
Michael Holt

Street
415 Mission Street

City
San Francisco

State
California

Post Code
CA 94105

Country
United States

Created By
User User, 20/10/2019, 11:09

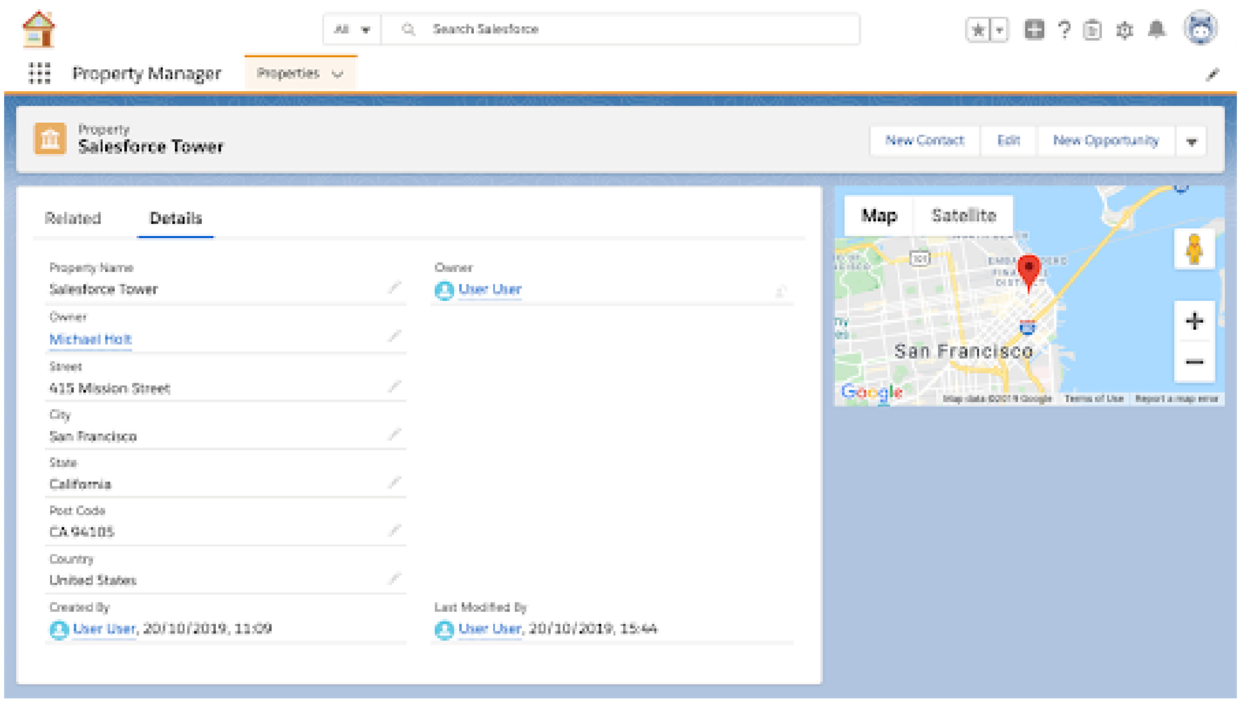
Owner
User User

Last Modified By
User User, 20/10/2019, 15:44

Figure 5.6: So far, your map should appear like this.

The only issue you might be noticing from your implementations, is that the map is highly zoomed in. This zoom level is currently being determined by the Google APIs but is too close to provide the value for our property management app. Fortunately, the lightning-map component has a solution. Let's set the zoom level to something more useful, I'm going to choose 12:

```
<template>
 <template if:true={propertyLocs}>
 <lightning-map
  map-markers={propertyLocs}
  zoom-level='12'>
 </lightning-map>
 </template>
</template>
```



Property Name
Salesforce Tower

Owner
Michael Holt

Street
415 Mission Street

City
San Francisco

State
California

Post Code
CA 94105

Country
United States

Created By
User User, 20/10/2019, 11:09

Owner
User User

Last Modified By
User User, 20/10/2019, 15:44

Figure 5.7: The map in an app with the adjusted zoom-level.

Incredible! With the backing of our base Lightning Web Component provided by Salesforce, we are able to render an interactive Google map with just 7 lines of HTML and a small JavaScript function, combined with an Apex Class. This is a true demonstration of the power of the Lightning Platform.

Part 6 | Create Engaging Customer Experiences with Lightning Flow

The next thing we'd like to add to our property management app is a simple wizard that will enable property managers to easily and conveniently book maintenance, inspections, and add other tasks related to managing their properties.

To do this, we are going to use Lightning Flow. Lightning Flow enables you to quickly automate business processes by building applications, known as flows, that collect, update, edit, and create Salesforce information. Then make those flows available to the right users or systems.

Create a Flow with Flow Builder

Flow Builder is one of the development tools within Lightning Flow. For our project, Flow Builder will enable you to use point-and-click to create a flow declaratively. The Property Management wizard will be created with a screen flow–a collection of screens and connectors that step the property managers a business process.

We'll be using the standard Salesforce object "Task" to hold this data.

To create the wizard:

1. Open the Setup Menu, type in "Flow" in the Quick Find box, and select the "Flows" option that appears under "Process Automation."
2. In the Setup Flows window, click the New Flow button.
3. When the New Flow window appears (Figure 6.1), select Screen Flow and then click the Create button. This will launch Flow Builder.

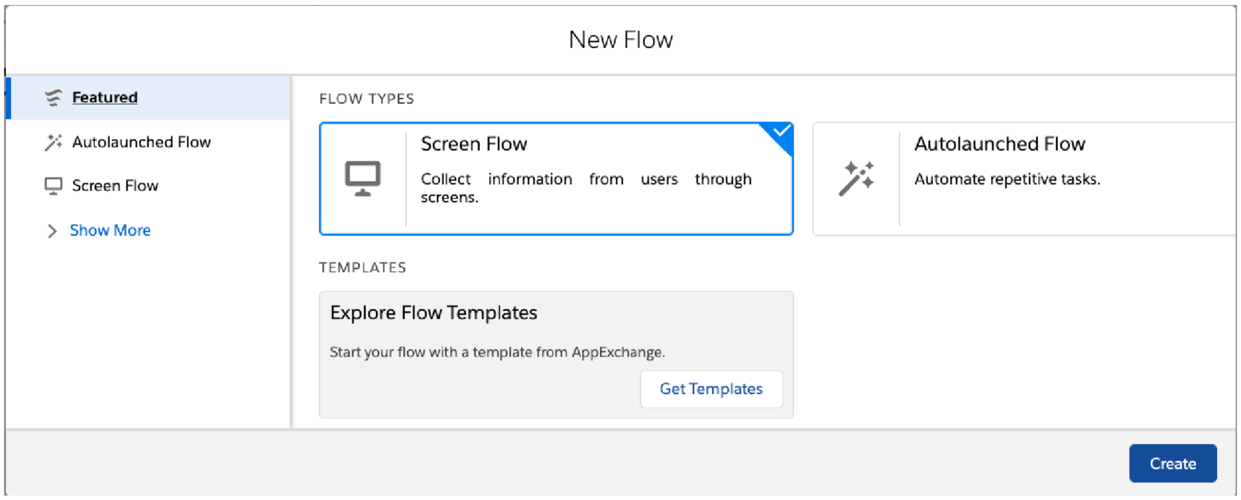
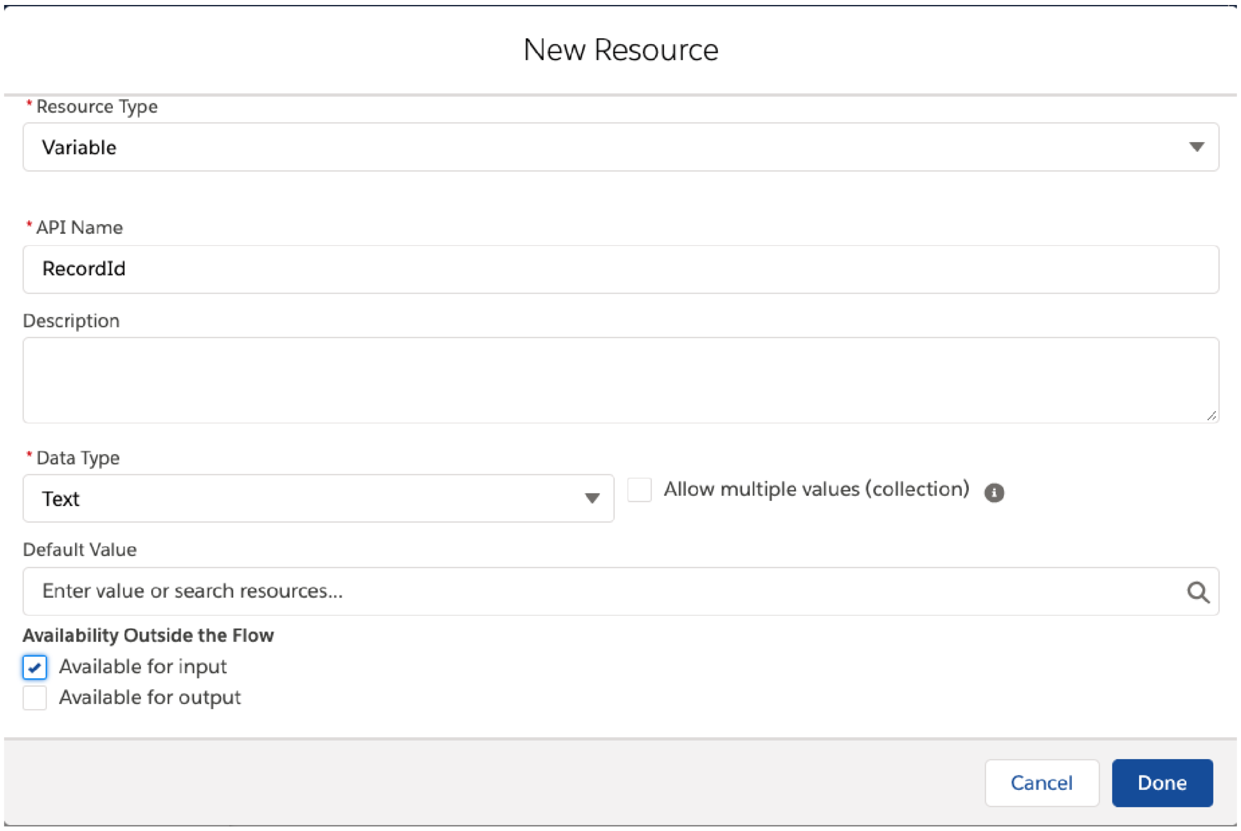


Figure 6.1: To create a new flow, select Screen Flow and then click the Create button.

4. In Flow Builder, click the Manager tab in the Toolbox and click the New Resource button.
5. In the New Resource box, select Variable from the Resource type pull-down menu. This will reveal several additional fields in the new Resource box. In the API Name field, type "RecordId." Select Text from the Data Type pull-down menu and check the Available for input box (Figure 6.2). If everything looks correct, click the Done button.

RecordId is a specific variable that we can create in Lightning Flow which will automatically detect the ID of the record we're currently looking at. This functionality is provided by the platform when creating this specific resource, so we don't need to create any custom functionality to do this for us.



New Resource

* Resource Type

Variable

* API Name

RecordId

Description

* Data Type

Text

Allow multiple values (collection)

Default Value

Enter value or search resources...

Availability Outside the Flow

- [x] Available for input
- [ ] Available for output

Figure 6.2: In the New Resource screen, the Resource type should be Variable, enter “RecordId” for the API Name, select Text as the Data Type, and check the Available for input box.

6. Next, we want to define the fields which the user can input data into. Switch back over to the “Elements” tab in the toolbox and drag a “Screen” anywhere on the canvas.
7. When the New Screen box appears, click on [Flow Label]. Under Screen Properties, located on the right side, enter “Quick Task Create” into the Label field. (The API Name will auto-populate with the same name. Expand the “Control Navigation” section of Screen Properties (you may need to scroll down a bit) and uncheck “Pause” box.
8. From the “Screen Components” pane on the left side of the New Screen box, drag and drop a Text field component under [Flow Label]. Make sure the new text field is selected and then enter “Subject” in the Label field. (Again, the API Name will auto-populate with the same text as the Label.) Check the Require box.
9. Next, drag a Long Text Area component under the Subject field and label it “Description.” If done correctly, your screen should look like Figure 6.3. If everything looks good, click the Done button.

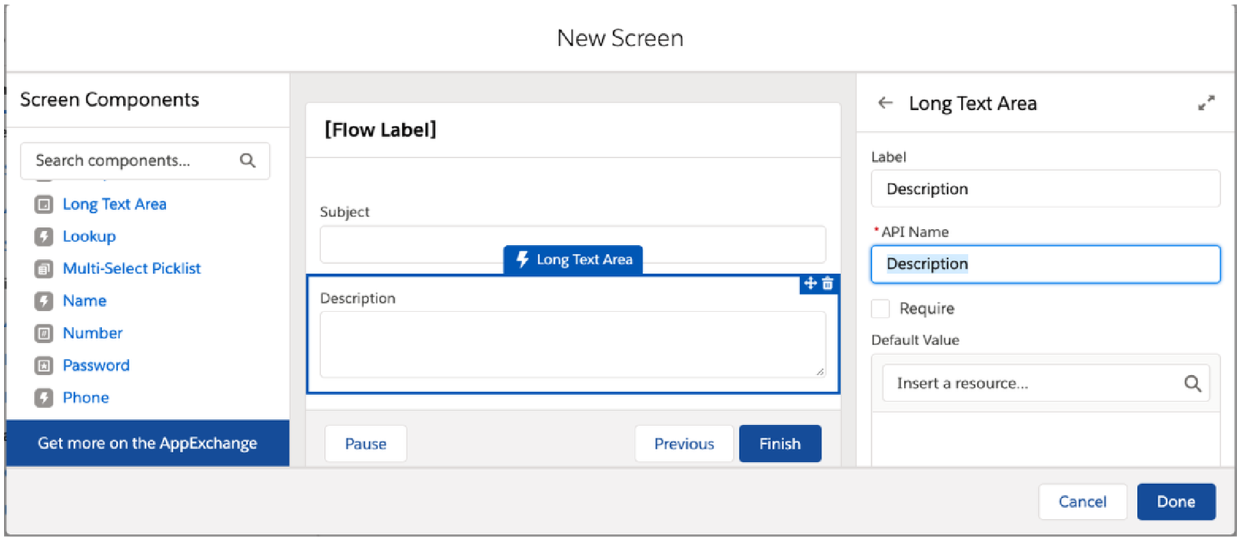


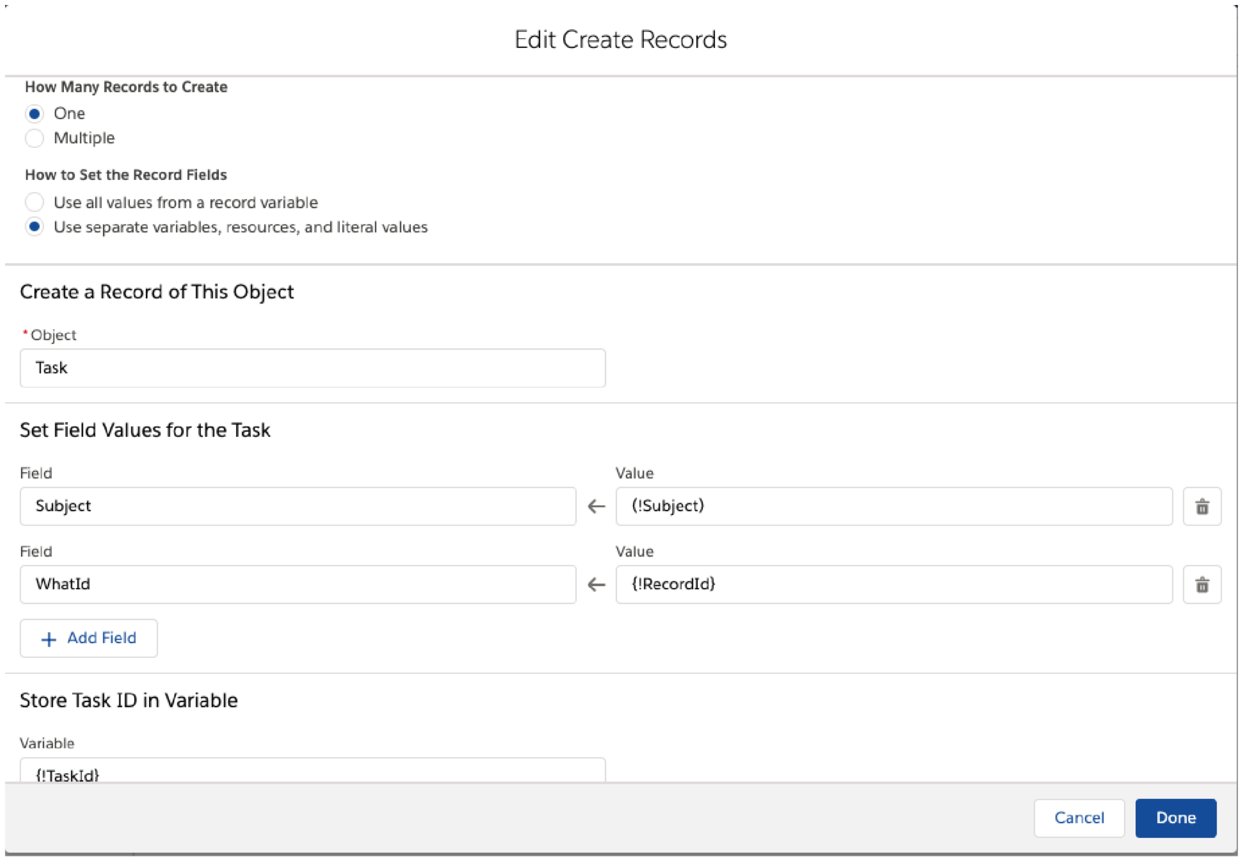
Figure 6.3: From the Screen Components column, drag the Text component and Long Text Area component onto the screen.

10. Back at the main Flow Builder screen, click Elements under the Toolbox and then drag the “Create Records” onto the screen. In the New Create Records box, enter “Create Task” into the Label field. and then select Use separate variables, resources and literal values.
11. Under “Create a Record of This Object,” search for and choose “Task.” Then, under Set Field Values for the Task, search for and select “Subject.” In the Value field, search for and enter {!Subject}. Then click the Add Field button and add two more fields:

- Description > {!Description}
- WhatId > {!RecordId}

Don’t click Done yet; stay in the New Create Records screen.

12. In the “Store Task ID in Variable” section, click the lookup field. Select “+ New Resource.” In the New Resource box, choose Variable From the Resource Type pull-down menu.
13. Type “TaskId” into the API Name field and then select Text from the Data Type pull-down menu. Leave the Available for input and Available for Output boxes unchecked.
14. Click the Done button. You will return to the New Create Records screen, which should now look Figure 6.4 If it looks correct, click the Done button.


Edit Create Records

How Many Records to Create

[✓ One] Multiple

How to Set the Record Fields

Use all values from a record variable [✓ Use separate variables, resources, and literal values]

Create a Record of This Object

*Object: Task

Set Field Values for the Task

Field	Value
Subject	(!Subject)
WhatId	{!RecordId}

Store Task ID in Variable

Variable: {!TaskId}

Figure 6.4: Your Edit Create Records screen should look like this.

15. The last thing we want to do is to be able to associate a file to the new Task, so let's drag another screen onto the Canvas. Under Control Navigation, once again uncheck the "Pause" value and then select the header and enter "Quick Task Create" as the label, followed by "Quick_Task_Create2" as the API Name. Next, drag the "File Upload" screen component onto the Screen and fill out the following values:

- API Name > File_Upload
- File Upload Label > File Upload
- Related Record ID > {!TaskId}

16. Link the flow together by clicking on the circle at the bottom of the "Start" icon and dragging the connector to our first "Quick Task Create." Do the same from Quick Task Create to Create Task. Lastly, from Create Task to the second Quick Task Create. Your canvas should look similar to Figure 6.5.

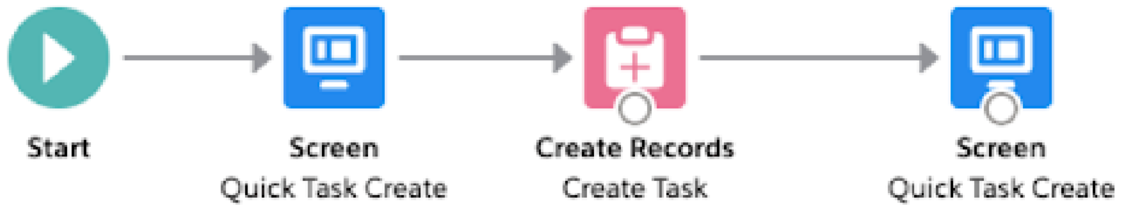


Figure 6.5: Your new Flow should look like this!

17. Once done, save the Flow and call it “Quick Task Create.” Note: You may receive some errors, if these errors say the following, then they can be ignored": *“These issues don't prevent activation, but can cause problems when you run the flow. XXXX (Screen) - This screen includes screen components that require Lightning runtime.*

If you are seeing other errors, you will likely need to fix them in line with the above instructions.

Debug and Activate the Flow

We can debug the Flow by selecting the “Debug” button. This will give us the option to associate a record with the flow. Copy and paste the ID from our dummy record from earlier and follow the Flow through, attaching a file at the end.

If this was successful, activate the flow by returning to the Flow Builder and selecting the “Activate” button in the top right corner.

Now we’ll add the Wizard and the tasks related list to our app screen.

1. Return to our dummy record and open the App Builder again, by selecting the cog > Edit Page.
2. Drag the standard “Flow” component anywhere onto the screen and then select the “Pass record ID into this variable” option in the right hand pane.
3. Next, select the “Related” tab on the layout and drag the standard “Activities” component onto the screen.
4. Save the page and then return to our record.

Selecting the “Related View” will now give users information on tasks associated with the property, as shown in Figure 6.6.

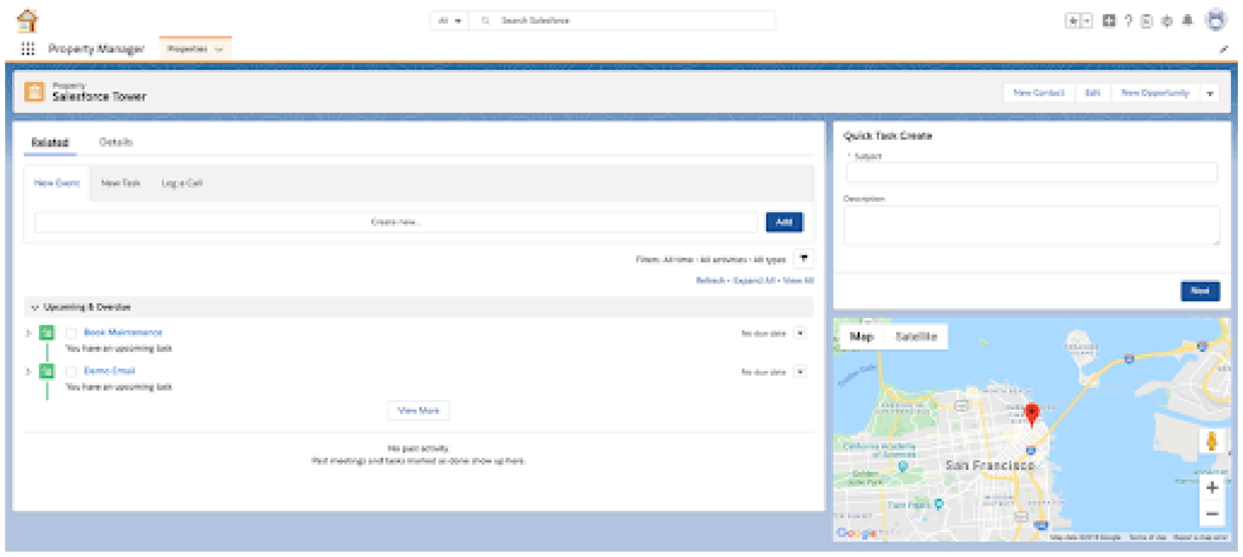


Figure 6.6: Now customers can view properties on an interactive map.

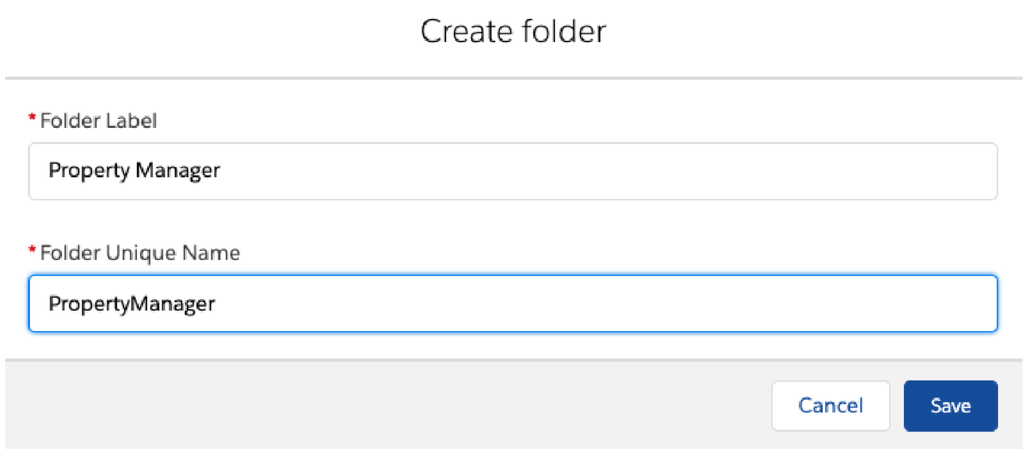
Part 7 | Use Reports to Give Users Insights Natively on the Platform

This final section of app building on the Salesforce platform surrounds reports. Reporting on Salesforce is provided. If your users want deeper business intelligence insights, use Einstein Analytics.

For this quick example, we're going to focus on Reports and provide our users with a view of how many properties a particular Contact owns.

Create a Report

1. Open the App Launcher and search for "Reports,"
2. Select Reports and then Select "New Folder." Give the folder an appropriate name, such as "Property Manager" and then hit the Save button (Figure 7.1).



Create folder

* Folder Label

Property Manager

* Folder Unique Name

PropertyManager

Figure 7.1: in the Create folder screen, enter the name of the folder.

3. Inside the new folder, select "New Report." From the menu, search for and select "Properties" (Figure 7.2).

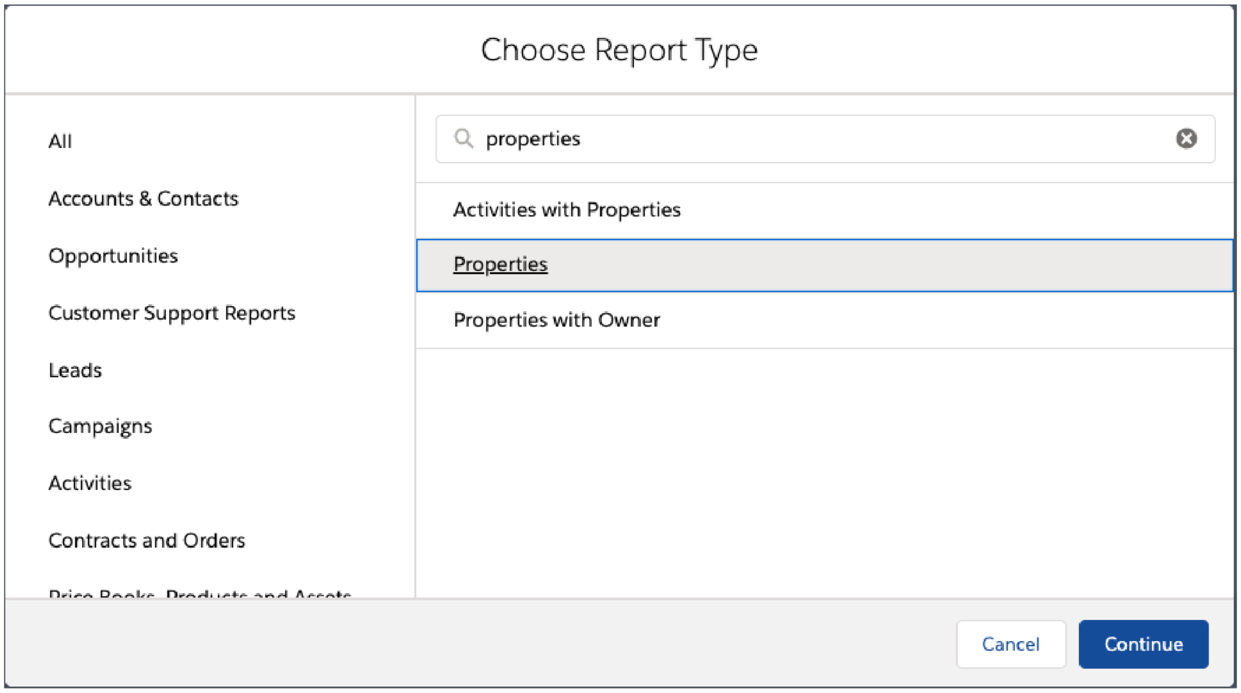
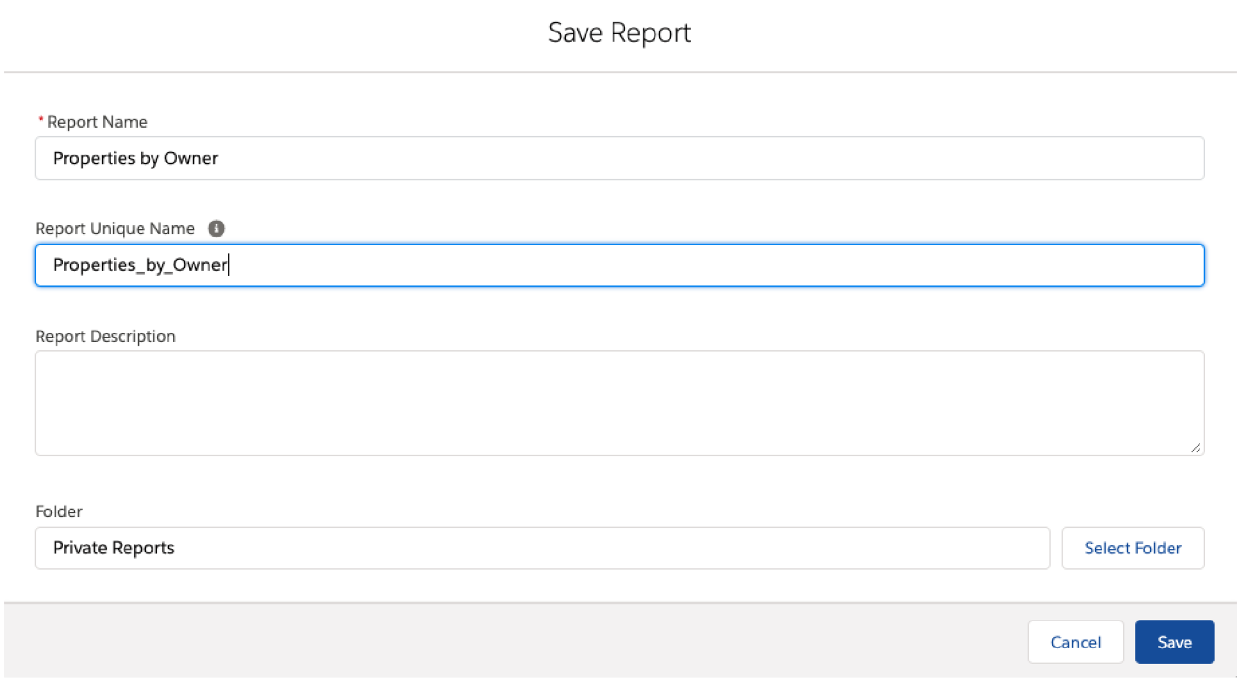


Figure 7.2: In the Choose Report Type screen, search for and then click on Properties.

4. On the page which appears, select the "Filters" menu on the left hand pane and change the "Show Me" option to "All properties" and the "Created Date" option to "All Time."
5. Expand the "Fields" pane and drag the "Owner" field onto the reporting canvas, next to "Property Name."
6. Click on the arrow next to "Owner" and then select "Group Rows by this Field."
7. Select the "Add Chart" button and then select the cog which appears next to the chart. Enter "Properties by Owner" as the Chart Title and select whichever chart that you like, before clicking "Save & Run."
8. Name the Report "Properties by Owner", change the folder to "Property Manager" and click Save.


Save Report

* Report Name
Properties by Owner

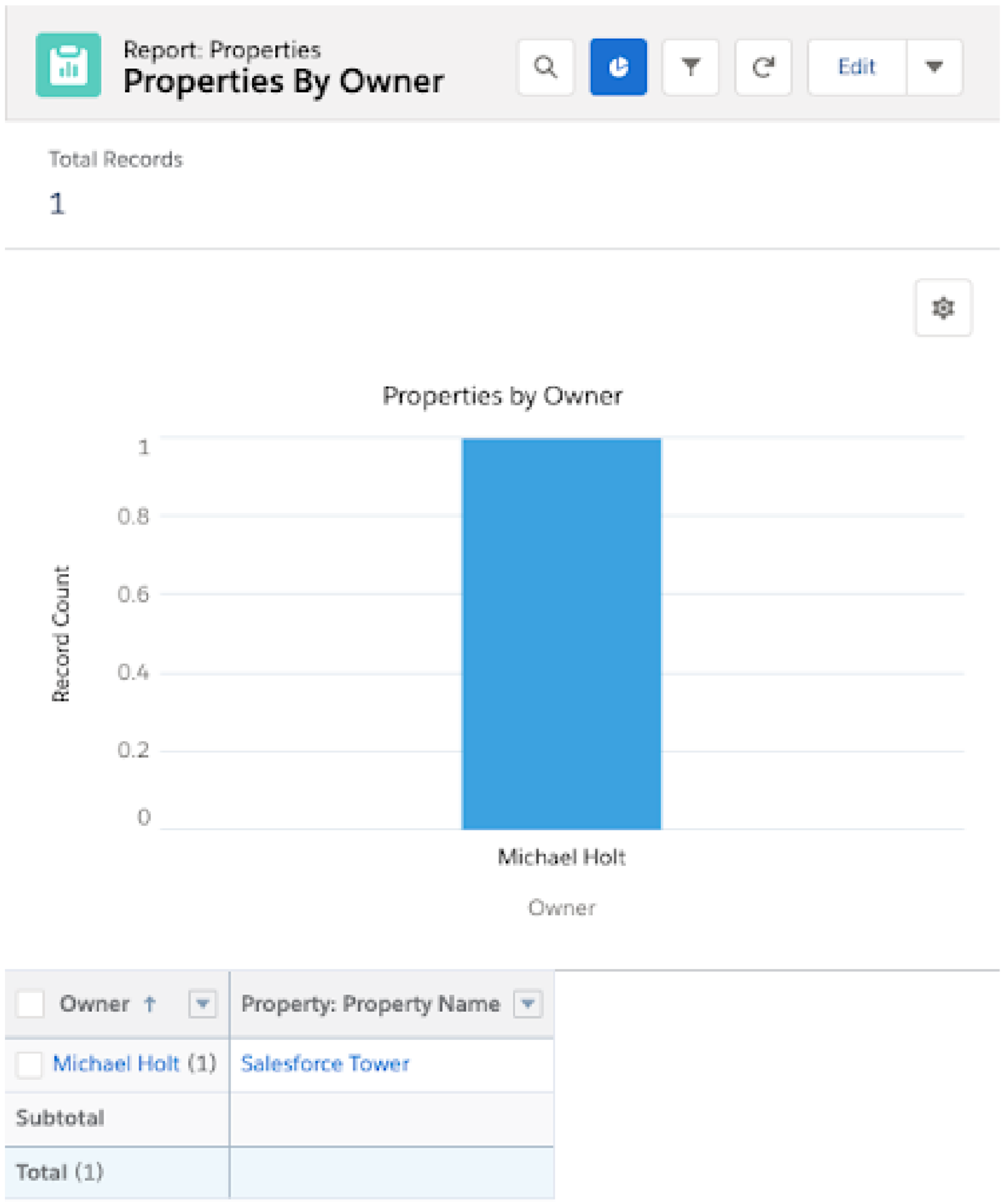
Report Unique Name
Properties_by_Owner

Report Description

Folder
Private Reports

Figure 7.3: Save Report screen.

You should have a very simple looking chart, similar to Figure 7.4.



Owner ↑	Property: Property Name
Michael Holt (1)	Salesforce Tower
Subtotal	
Total (1)	

Figure 7.4: Your chart should look like this.

Add a Utility Bar

For our users to have quick and easy access to our shiny new report, let's add this to the Utility Bar within our app, so users can always pop it open to see what the status of their owners looks like.

1. Head to the Setup Menu and search for "App Manager." Select it, then hit edit next to the "Property Manager" app.
2. Under the "Utility Items" menu, select "Add Utility Item" and choose an appropriate label.
3. Under the "Report" option, ensure you've chosen "Properties By Owner" and save it.

4. While we're here, we'll allow our users quick access to the tasks they've created within our app, too! Select "Navigation Items" and then add "Tasks" to our "Selected Items". Click Save and return to your Property Manager App.

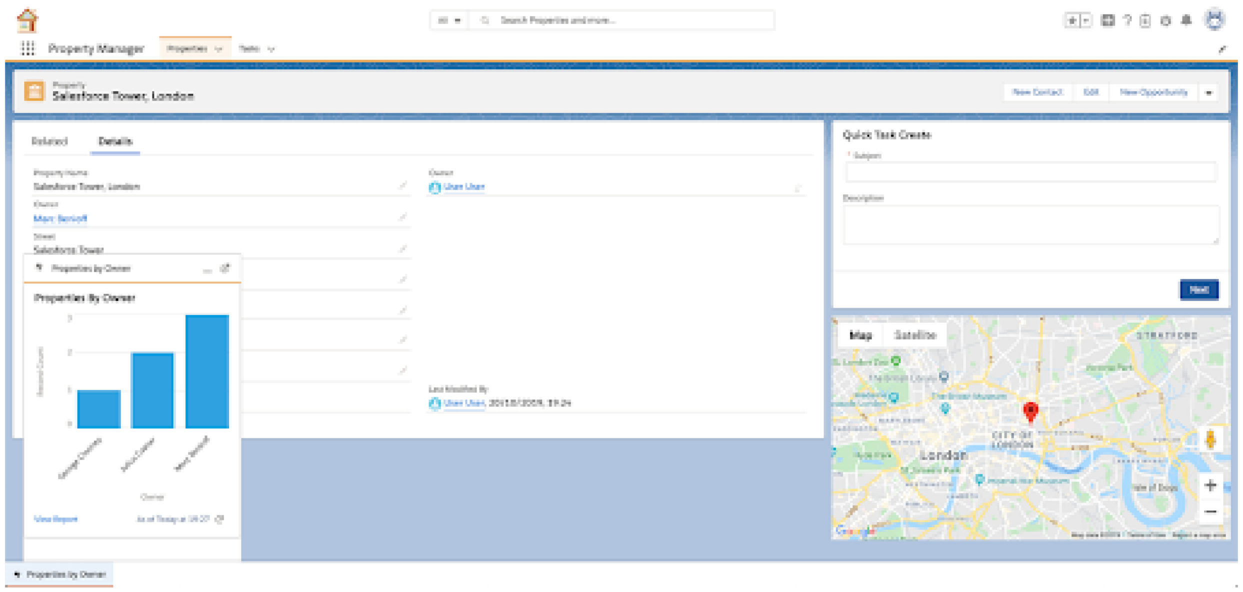


Figure 7.5: You have added a report to your Property Management app!

As far as our property management solution goes, that's it! We've achieved a great amount on the platform in a short period of time and hopefully you can begin to see how these examples might apply to your own use cases. For now though, take some time to play around and admire what you've built, before we move into the next part: Packaging the App.

Part 8 | Package and Distribute your Property Manager

It's time to take our solution and make it installable. By making the app installable, anyone in the Salesforce ecosystem can install our fantastic property management solution into their org. This is why packaging is important. By bundling your solution into a protected and replicable box, it can be delivered, downloaded, and installed by multiple customers.

We currently have our programmatic metadata locally on our machine. However, everything that we built declaratively is still within the scratch org. In order to package the solution, we'll need to ensure all of the metadata is held within our project.

Before we begin, we need to make some amendments to the ".forceIgnore" file in our project. This file tells SFDX to ignore certain components that we are not interested in. We mentioned previously that ISVs use permission sets, rather than profiles, to control access to functionality.

For that reason, we'll add profiles to the forceIgnore document, along with the app switcher and Task metadata, since we don't want any modifications we made to tasks to impact our customers' use of the Task object.
Let's add the following lines:

```
**/profiles/**
**AppSwitcher**
force-app/main/default/applications/Task**
force-app/main/default/layouts/Task-Task
force-app/main/default/flexipages/Task**
```

Your file should look something like the following:

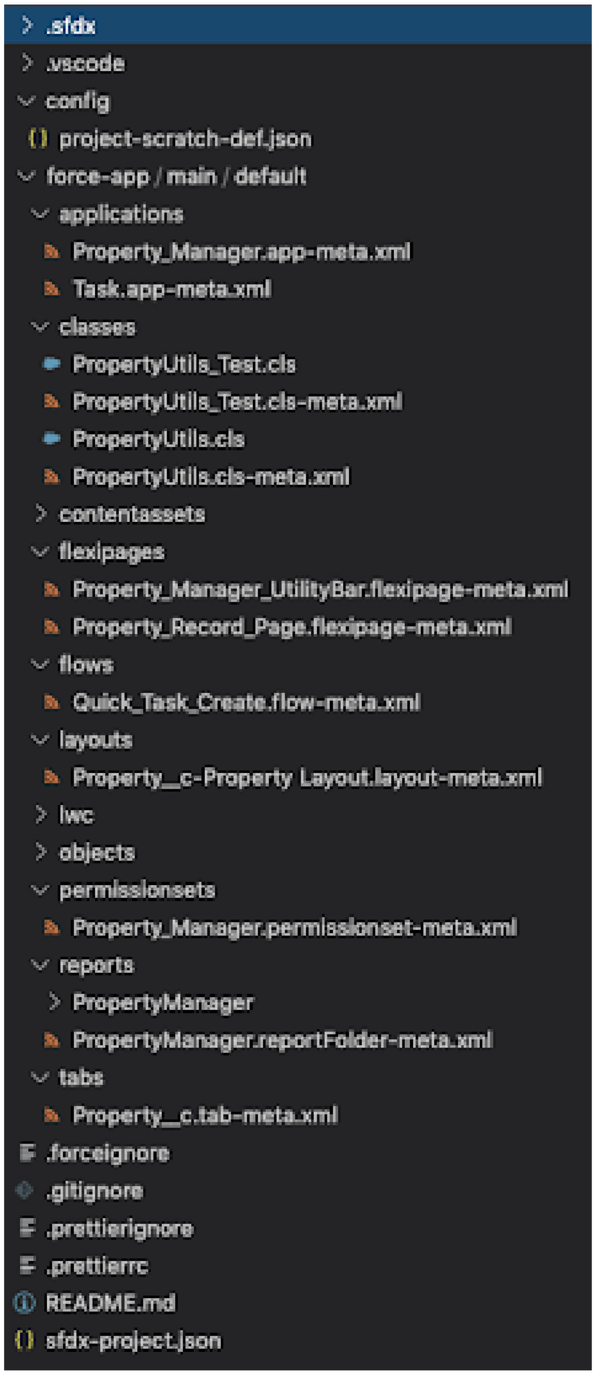
```
# List files or directories below to ignore them when running force:source:push, force:source:pull, and force:source:status# More information: https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_exclude_source.htm#package.xml# LWC configuration files**/jsconfig.json**/.eslintrc.json# LWC Jest**/__tests__/****/profiles/****AppSwitcher**force-app/main/default/applications/Task**force-app/main/default/layouts/Task-Taskforce-app/main/default/flexipages/Task**
```

Pull Metadata from the Scratch Org

Once we've amended the forceIgnore file, we can pull down any metadata from our scratch org that we've not yet got locally on our machine, we can do this using the following command:

```
sfdx force:source:pull -u appScratch
```

After a brief period of time, your terminal should suddenly be populated with all of the files not yet in the repository, including our flow, report, page layouts, Flexipages and permission set. If you've already run the pull command in this workbook, then you may see fewer results as part of this particular pull. Nevertheless, make sure that your project has all of the relevant metadata within it. It should look like Figure 8.1.



```
> .sfdx
> .vscode
v config
  {} project-scratch-def.json
v force-app / main / default
  v applications
      Property_Manager.app-meta.xml
      Task.app-meta.xml
  v classes
      PropertyUtils_Test.cls
      PropertyUtils_Test.cls-meta.xml
      PropertyUtils.cls
      PropertyUtils.cls-meta.xml
  > contentassets
  v flexipages
      Property_Manager_UtilityBar.flexipage-meta.xml
      Property_Record_Page.flexipage-meta.xml
  v flows
      Quick_Task_Create.flow-meta.xml
  v layouts
      Property__c-Property Layout.layout-meta.xml
  > lwc
  > objects
  v permissionsets
      Property_Manager.permissionset-meta.xml
  v reports
    > PropertyManager
      PropertyManager.reportFolder-meta.xml
  v tabs
      Property__c.tab-meta.xml
.forceignore
.gitignore
.prettierignore
.prettierrc
README.md
{} sfdx-project.json
```

Figure 8.1: Your project should look like this.

If you see any additional metadata within your own project, you can simply delete it from the project folder. You don't want to mistakenly add any unwanted metadata to your package.

Create the Package

Once all the metadata is cleanly stored within our project, it's time to create the package. This is done very simply, with the following command:

```
sfdx force:package:create --name PropertyManager --path force-app --packagetype Managed
```

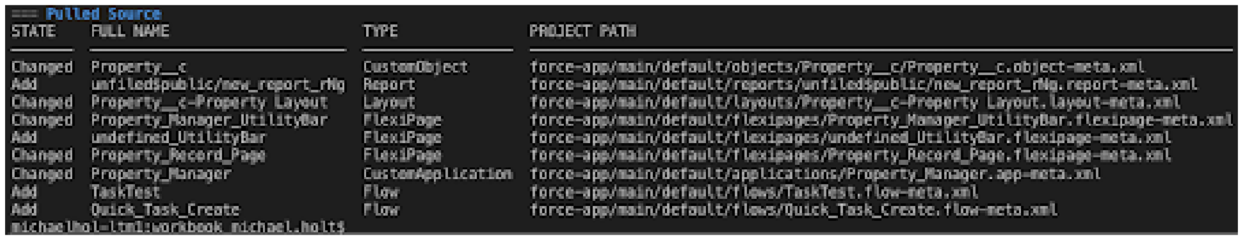
We need to specify 3 things as part of this command.

- First the package name. I've chosen "PropertyManager."
- Second, indicate the path where our metadata is held. In this case, the metadata is located in the force-app folder of our package.
- Third, indicate the type of package we wish to create. All AppExchange packages must be managed packages. There are many advantages to managed packages, which we won't delve into here.

Once you've executed the command, you'll receive the following output:

```
sfdx-project.json has been updated.Successfully created a package. 0Ho1n000000KykOCAS
```

Running the command has automatically associated our package with our project! Let's go and see exactly what updates have taken place within our sfdx-project.json file, as shown in Figure 8.2.



```
=== Pulled Source
STATE    FULL NAME                        TYPE               PROJECT PATH
───────  ───────────────────────────────  ─────────────────  ──────────────────────────────────────────────────────────────────────────────
Changed  Property__c                      CustomObject       force-app/main/default/objects/Property__c/Property__c.object-meta.xml
Add      unfiled$public/new_report_rNg    Report             force-app/main/default/reports/unfiled$public/new_report_rNg.report-meta.xml
Changed  Property__c-Property Layout      Layout             force-app/main/default/layouts/Property__c-Property Layout.layout-meta.xml
Changed  Property_Manager_UtilityBar      FlexiPage          force-app/main/default/flexipages/Property_Manager_UtilityBar.flexipage-meta.xml
Add      undefined_UtilityBar             FlexiPage          force-app/main/default/flexipages/undefined_UtilityBar.flexipage-meta.xml
Changed  Property_Record_Page             FlexiPage          force-app/main/default/flexipages/Property_Record_Page.flexipage-meta.xml
Changed  Property_Manager                 CustomApplication  force-app/main/default/applications/Property_Manager.app-meta.xml
Add      TaskTest                         Flow               force-app/main/default/flows/TaskTest.flow-meta.xml
Add      Quick_Task_Create                Flow               force-app/main/default/flows/Quick_Task_Create.flow-meta.xml
michaelhol-ltm1:workbook michael.holt$
```

Figure 8.2: Note that all of the IDs you see, will be different to my own, this is because they're all unique.

Generate a Package Version

In order to install our solution into our customer's orgs, we need to create an instance of a package called a package version. A package version allows us to capture the metadata at a particular point in time, facilitating upgrades and improvements to our product.

Let's create our first package version now, using the following command, ensuring you replace the package parameter with the ID of your own package, taken from the sfdx-project.json file:

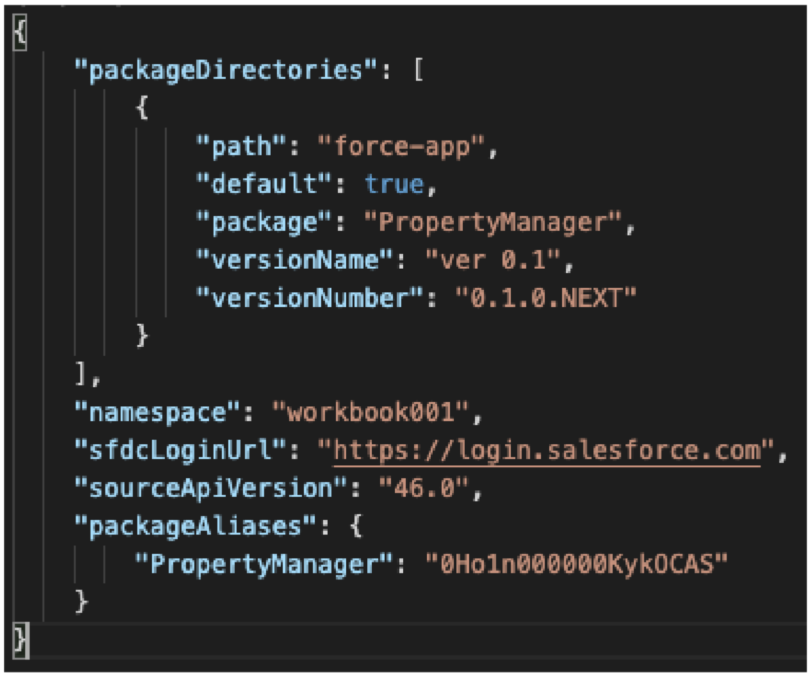
```
sfdx force:package:version:create --package 0Ho1n000000KykOCAS --definitionfile config/project-
scratch-def.json --codecoverage --installationkeybypass --wait 10
```

That's quite a hefty command, so let's break it down:

- Tell SFDX to create a package version associated with the package.
- Point the package version to the scratch org definition file. This specifies a list of features and org preferences our package depends upon.
- A –codecoverage parameter is required for managed packages. It forces our Apex tests to run and ensures that we have the necessary code coverage to promote our package (something we'll get to shortly).

- Because we want to distribute our solution far and wide via the AppExchange, we've added the installationkeybypass parameter here to *prevent* password protection. (It is possible to associate a key with our packages that will password-protects the installation, but we don't want to do that in this case.) Tell SFDX to wait up to 10 minutes for this request to complete.

Before too long, your terminal should look similar to my own. You'll also see another update to your sfdx-project.json file, to include the package version, as shown in Figure 8.3.:



```
{
    "packageDirectories": [
        {
            "path": "force-app",
            "default": true,
            "package": "PropertyManager",
            "versionName": "ver 0.1",
            "versionNumber": "0.1.0.NEXT"
        }
    ],
    "namespace": "workbook001",
    "sfdcLoginUrl": "https://login.salesforce.com",
    "sourceApiVersion": "46.0",
    "packageAliases": {
        "PropertyManager": "0Ho1n000000KykOCAS"
    }
}
```

Figure 8.3: You'll see the package version update in your sfdx-project.json file.

Promote the Package

The last stage of creating our distributable solution is to promote it. In real terms, this is the process of uploading the package to the Salesforce servers so that it an be replicated globally and available for installation into any Salesforce instance.

In order to promote the package, execute the following command. Make sure that the ID is replaced with your own.

```
sfdx force:package:version:promote -p [id number]
```

```
michaelhol-ltm1:ISVWorkbook michael.holt$ sfdx force:package:version:promote -p 04t1n0000028NeRAAU
```

You'll be prompted to confirm the action, as it cannot be reversed. Type "y" and return for the command to execute.

Part 9 | Install and Test Your App

The final section of this workbook, we'll install and test our app within demo environments. To test this, we'll use SFDX. However, this time we'll use it to create an Enterprise Edition test environment and remove our namespace from the project definition file.

1. Start by opening the project-scratch-def.json file and replacing “Partner Developer” with “Enterprise”:

```
{
 "orgName": "michael.holt Company",
"edition": "Enterprise",
"features": [],
"settings": {
"orgPreferenceSettings": {
"s1DesktopEnabled": true,
"s1EncryptedStoragePref2": false
  }
  }
}
```

2. Next, open the sfdx-project.json and remove the reference to our namespace, leaving it blank:

```
{
"packageDirectories": [
{
"path": "force-app",
"default": true, "package": "PropertyManager",
"versionName": "ver 0.1",
"versionNumber": "0.1.0.NEXT"
}
],
"sfdcLoginUrl": "https://login.salesforce.com",
"sourceApiVersion": "46.0",
"packageAliases": {
"PropertyManager": "0Ho1n000000KykOCAS",
"PropertyManager@0.1.0-1": "04t1n0000028NeMAAU",
"PropertyManager@0.1.0-2": "04t1n0000028NeRAAU"
 }
}
```

3. Create the org using:

```
sfdx force:package:install --package 04t1n0000028NeRAAU -u EnterpriseTesting
```

4. Once you receive the success message, we can install our app via the command line, like so:

```
sfdx force:package:install --package 04t1n0000028NeRAAU -u EnterpriseTesting
```

5. SFDX will then provide us with a command we can execute to retrieve the status. Wait a few minutes and then execute the following, with the username provided by the CLI, until you receive a success message:

```
sfdx force:package:install:report -i 0Hf7E0000000oPcSAI -u test-moo175cih2zy@example.com
```

6. Now that the package is installed, let’s use the CLI to assign ourselves the Property Manager permission set, so

that we can view the app, without assigning this via the UI:

```
sfdx force:user:permset:assign -n Property_Manager -u EnterpriseTesting
```

7. And finally, open the org:

```
sfdx force:org:open -u EnterpriseTesting
```

Congratulations, you've just installed your first Salesforce application into a "customer" org. You can now see your deployed solution (Figure 9.1).

Associate Contacts to Accounts

Because we've created an Enterprise Edition instance of Salesforce, we're required to associate Contacts to Accounts. The first thing to do is go to the App Launcher, search for "Account," and then create one.

Following that, we can proceed to our newly-deployed Property Manager app! Create a Property, and associate it with a Contact, using our new Account. From there, you'll be able to interact with the app just as you were able to whilst we were developing it.

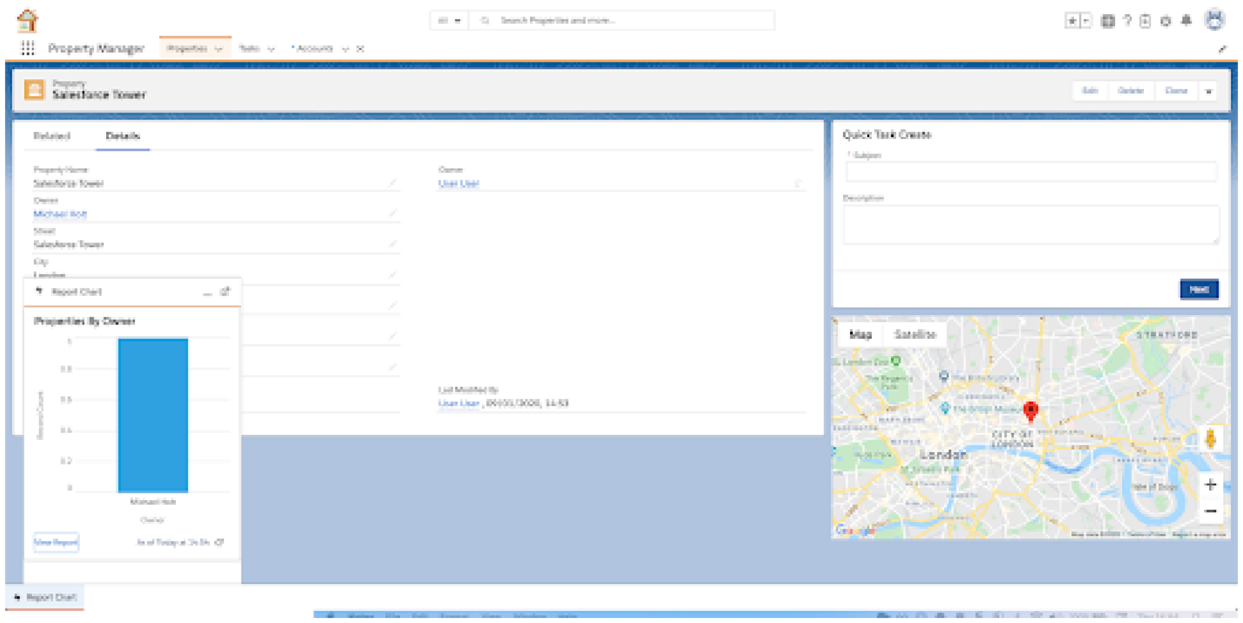


Figure 9.1: Congratulations. You've built your first AppExchange app.

To explore the setup menu, click the cog in the upper-right and select "Setup." In the search box, type "Apex Classes.

Open your classes to view your code. You'll see that the code is hidden in this subscriber org. One of the many advantages of managed packages being displayed here is the obfuscation of code, protecting the intellectual property of our partners.

You can also visit the "Installed Packages" menu within Setup and where customers can view all of their installed packages.