Workshop: 5 Things You Can Now Do with Lightning Web Components copy

Updated: Jan-28-2020

Abstract

Lightning Web Components (LWC) launched with a bang. Whether you have used them before or are brand new, join this workshop to learn five ways LWC improve developer productivity.

Setup

PER MACHINE

- git
- node/npm (for macs, install via brew)
- salesforce stuff: https://trailhead.salesforce.com/content/learn/projects/quick-start-lightning-web-components/set-up-visual-studio-code

BEFORE EACH WORKSHOP

Get an org from the deployer: https://lightning-platform-workshops.herokuapp.com/tdx19dev and click **Launch**

Keep the tab open for the username/password so you can link your SFDX to it


```
# get an org from the deployer, https://lightning-platform-workshops.herokuapp.com/df
# leave that page open so you can use the username/password to connect to it

git clone https://github.com/mshanemc/lwc-workshop
cd lwc-workshop
npm install

# open vscode in the current directory
code .
```

FROM INSIDE VSCODE:

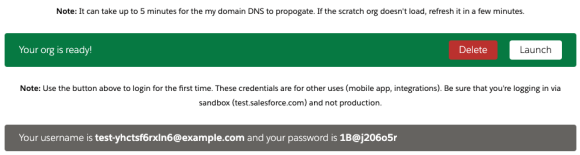
1. cmd-shift-p
2. Type enough of `default` or `Set a Default` so you can pick **SFDX: Set a Default Org**
3. Select `+ Authorize an Org`

4. Select **Sandbox**
5. Press enter to skip the alias.
6. Use your username and password from the Public Deployer to login on the browser tab that opens
7. Allow Access to let the cli connect to your org.

💬 ***Login pro-tip:*** make sure you aren't cutting and pasting a space onto the beginning/end of your username.

AFTER EACH WORKSHOP

```
sfdx force:auth:logout -p
cd ..
rm -rf lwc-workshop
# delete the org via the red button
```



Hands-On

Introduction

Lightning Web Components (LWC) let developers use modern, standard JavaScript to build lightning components that can be used in Lightning desktop, mobile, communities, and even off Salesforce since the entire framework is open source.

Today, we're going to get hands on with 5 awesome features of LWC. We assume that you've worked with our previous component framework (Aura) and know a bit of LWC's basic syntax. If not, you'll probably still be fine.

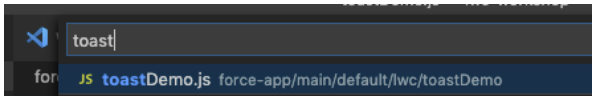
User Setup

If you haven't done this before, your machine has Visual Studio Code (VSCode), git, node, the Salesforce extensions for VSCode, and jest (a testing framework) installed. If you like what you see today, the instructions for setting this environment up are in the LWC dev guide.

#1 Dev Tooling

Since we pivoted from Eclipse to VSCode as our supported, free, open-source IDE that we want to build extensions for, we've built some cool features for coding.

Open the **ToastDemo** component (cmd-P, type toast until you find toastDemo.js).

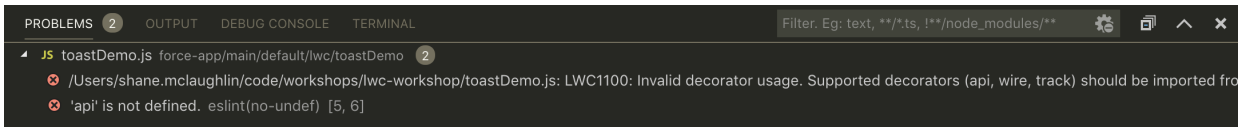


Uncomment out the **myProperty** line (cmd-/)

```
export default class ToastDemo extends LightningElement {
  @api myProperty;
```

In LWC, we have a js decorator called @api (other components can set this value as part of the component's public...api!)

Click on the **Problems** tab



You'll see eslint doing its job...and we've created LWC-specific ESlint rules. In LWC, you use static imports for everything you depend on. This helps the compiler check your code, but also helps the linter give you better feedback. By using standard import syntax and standard tools like eslint rules, we've made it possible for any IDE to support LWC.

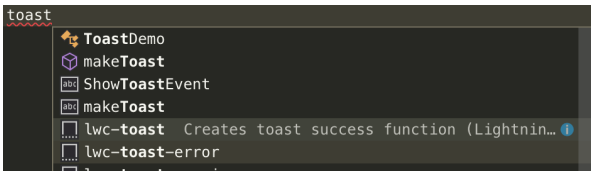
*Change the first line to add the **api** decorator, and the error disappears.*
`import { LightningElement, api } from 'lwc';`

Imports also give us access to all the salesforce data and ui services.
Add this line to our imports
`import {  ShowToastEvent } from 'lightning/platformShowToastEvent';`

```
import { LightningElement, api } from 'lwc';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
```

That's a toast event. How do we use it? Inside the makeToast method, type the word `toast`.

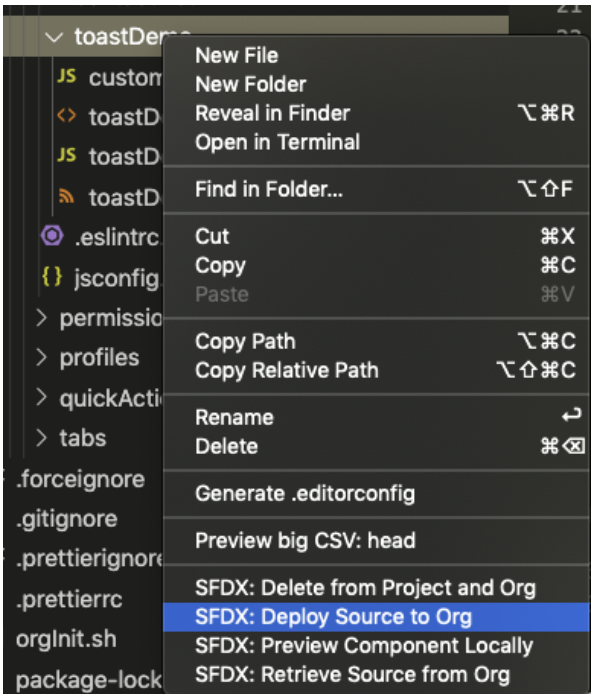
You'll see autocomplete snippets suggestion–find lwc-toast, which will build out the shell of what you can make your toast say.



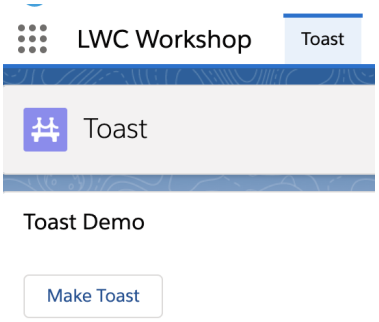
Fill out your toast

```
makeToast() {
    this.dispatchEvent(new ShowToastEvent({
        title: 'Toast',
        message: 'is really delicious',
        variant: 'success'
    }));
}
```

Right-click the component bundle folder in the sidebar and select `SFDX: Deploy source to org`



In the org, click the button on the component to see your toast.



Not only can you import all the Salesforce-specific modules, but you can even import your own custom modules. Open ***customModule.js.*** *Here's a function that uses raw JavaScript to format a date. We can import that in our LWC...and*

store the file in the component bundle, or in some other folder for reuse by other components

*Add this import statement to **toastDemo.js***
`import { getTomorrow } from './customModule';`

*Now we can use that function in our message. Change the **message** property of our **toast.** We're using es6 template literals (those backticks and ${} syntax), one of the many modern js things you can do in LWC that'll run on browsers that don't support them.*

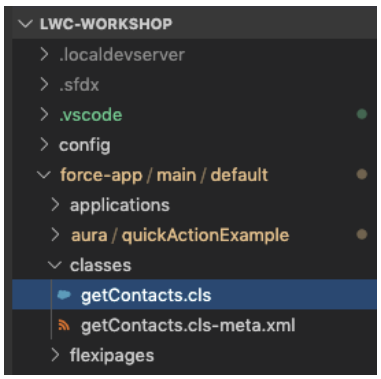
```
            message: `would be delicious on ${getTomorrow()}`,
```

And we've done so much...you'll see more as we continue.

```
import { LightningElement, api } from 'lwc';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
import { getTomorrow } from './customModule';

export default class ToastDemo extends LightningElement {
  @api myProperty;

  makeToast() {
    this.dispatchEvent(new ShowToastEvent({
      title: 'Toast',
      //message: 'is really delicious',
      message: `would be delicious on ${getTomorrow()}`,
      variant: 'success'
    }));
  }
}
```

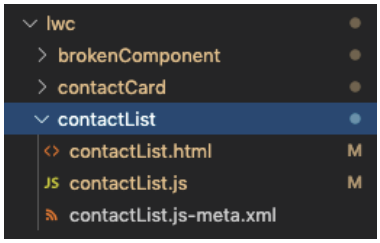
#2 Apex access with way less work

Open **getContacts.cls** in the **/classes** folder. Here's the simplest apex class we could think of...this isn't an apex workshop.



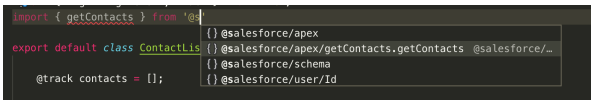
Ever called apex from Aura? You define an attribute, and a controller (one and only one!).
Here's the call...lots of boilerplate for each call/callback/error handling.

Let's do the same thing in LWC.
Open **contactList.js** in the **contactList** component



Import the apex class/method

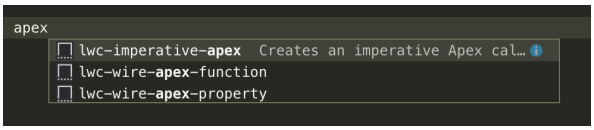
Wow...would you look at that amazing autosuggestion of our apex classes with AuraEnabled methods!




```
import getContacts from '@salesforce/apex/getContacts.getContacts';
```

- @wire a property to the method

Inside your component type `apex` and grab the **lwc-wire-apex-property** snippet



And modify it like this

```
@wire(getContacts, {})
contacts;
```

Between the curly brackets, we'd pass in an object for any parameters that our apex method needs...ours didn't need any. `contacts` *is now a variable we can use in our template.*

You'll need to have `wire` imported to use wires...but the linter will let you know that.

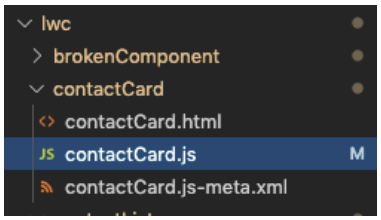
Seriously, that's it. There's more you can do (call apex imperatively, invalidate the cache, etc).

```
import { LightningElement, wire } from 'lwc';
import getContacts from '@salesforce/apex/getContacts.getContacts';

export default class ContactList extends LightningElement {
@wire(getContacts, {})
contacts;
}
```

#3 Data Service

We only queried the IDs in our apex method. What if we want more fields? We could get them from apex...but, in LWC, if you have the ID of a record, you can get other things really elegantly.

Open **ContactCard.js**



There's a well-named service called getRecord that we can import `import { getRecord } from 'lightning/uiRecordApi';`
Add this to your imports

```
import { LightningElement, api, wire } from 'lwc';
import { getRecord } from 'lightning/uiRecordApi';
```

*You'll see some **fields** already defined...they're here so you can easily add more.*

Type **wire** inside our component class and pick **lwc-wire-get-record-property** then set the fields and property name. *That special $ is a binding to the recordId property...when that changes (we're about to use this as a child of ContactList), this getRecord fires again.*

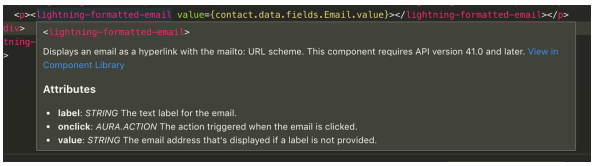
```
@wire(getRecord, { recordId: '$recordId', fields: FIELDS })
contact;
```

and, of course, we add **wire** to the imports `import { LightningElement, api, wire} from 'lwc';`

```
import { LightningElement, api, wire } from 'lwc';
import { getRecord } from 'lightning/uiRecordApi';

const FIELDS = [
    'Contact.Name',
    'Contact.Title',
    'Contact.Phone',
    'Contact.Email',
];

export default class ContactCard extends LightningElement {
    @api recordId;
    @wire(getRecord, { recordId: '$recordId', fields: FIELDS })
contact;


}
```

Open **contactCard.html**
You can see that we're taking the data that comes back form the contact and using it to populate the template.
You'll notice the use of some base components...there's lots of these in LWC and you should always use them. They'll give you styling, localization, accessibility and sometimes some fancy features without doing much work yourself.

Not only do they have autocomplete, but hover over one...it's the docs, along with a link to the component library!



Also notice the conditional rendering–there's actually error handling built in in case something weird happened.

*Back on **contactList.html**, let's take out our bullets and put in our new cards.*

Change the following lines

```
<li key={contact.Id}>
    {contact.Id}
</li>
```

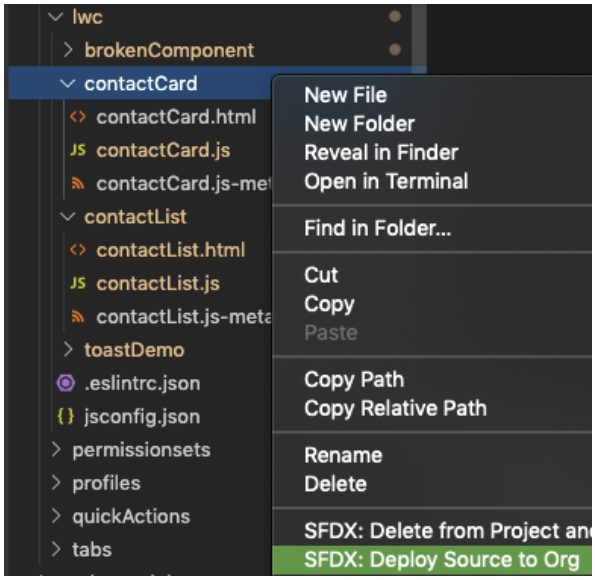
to (*and you'll see that amazing autocomplete here, too!*).

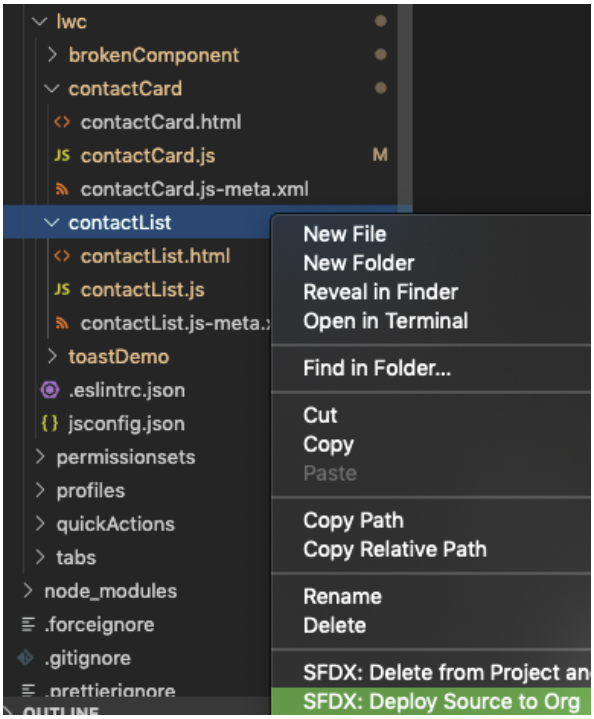
```
<c-contact-card record-id={contact.Id} key={contact.Id}></c-contact-card>
```

```
<template for:each={contacts.data} for:item="contact">
    <!--
    <li key={contact.Id}>
        {contact.Id}
    </li>
    -->
    <c-contact-card record-id={contact.Id} key={contact.Id}></c-contact-card>
</template>
```

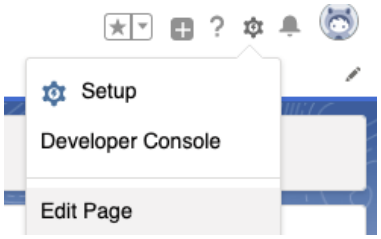
key *is not actually part of the component–it's related to the for:each directive and is used by LWC when mutating the array. LWC follows the web components standard for syntax (it-is-kabob-case) to things are slightly different from the js names (headsDownCamelCase)*

Deploy ***contactList*** *and* ***contactCard*** *up to the org, and let's see our contact cards come alive.*

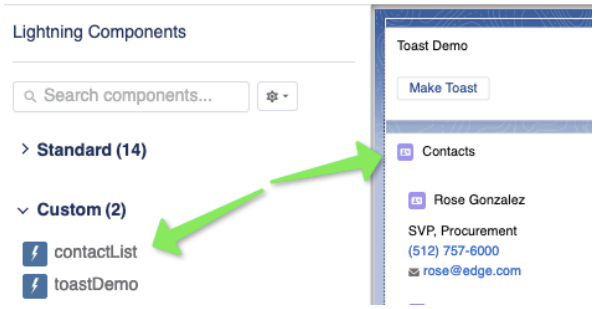




Edit the LWC Workshop > Toast page



Add the **contactList** custom component to the page under Toast Demo



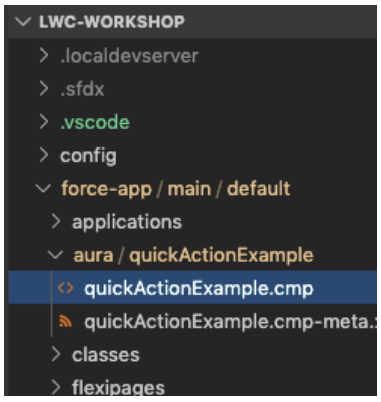
Click **Save** and **Back**

*There's a lot in @wire ui...you can import **createRecord** or **updateRecord** or get things like picklist values by recordType, or object metadata and layouts. But they all use the same pattern. Side note...we could have used the **getListView** data service to get our contacts and skipped apex entirely if we weren't trying to show off how easy it is now.*

#4 Interoperability

Got some complex Aura component that's not ready to move to LWC but you need to add a bit to it? Or need to use a container that LWC doesn't support yet? You may have to work with Aura. But, you can do as little Aura as possible by putting LWC in your aura.

Open **quickActionExample.cmp in aura/quickActionExample**



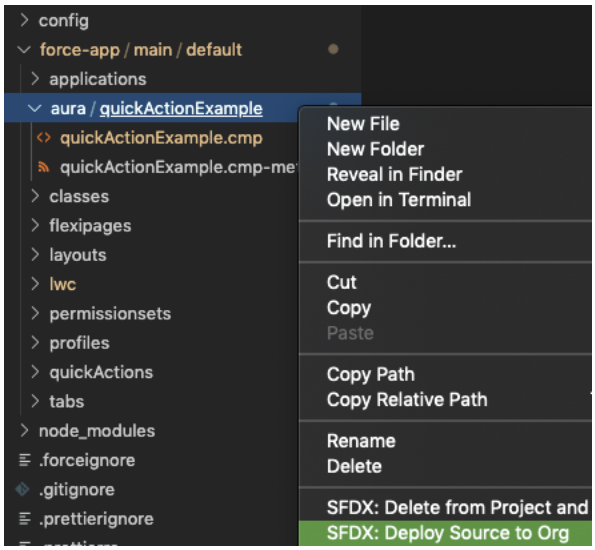
Here's an example of an aura component that can be used in a quickAction. We're just using Aura as a wrapper around our LWC because you can't use an LWC as a quickAction yet. It's a temporary, lightweight solution so you can use LWC now and not create more technical debt.

Add our component `<c:contactList/>` inside the Aura component. If it had any api attributes, you could pass those in.

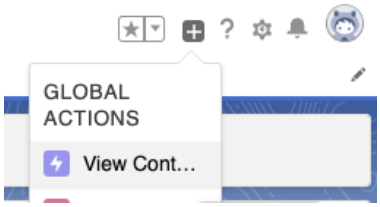
```
<aura:component implements="force:lightningQuickAction">
    <c:contactList/>
</aura:component>
```

*Notice that since we're in an Aura component, we use the aura syntax instead of the web component syntax (that would be **c-contact-list**)*

Right-click the component bundle and click `SFDX: Deploy Source to Org`



From the org, click on the global actions menu (+) and use the **View Contacts** action.



There's way more to interop–you can pass events up and out, and attributes down and in. If your LWC needs more context (like recordId for an action on a record page) get that context from Aura and pass the values in to your LWC. Salesforce has made it easy to use LWC in Aura, and leave your Aura until you're ready to migrate it.

#5 Local testing

Have you ever worked with a local test runner before when working on some non-Salesforce project? Your tests run anytime you change code, and you get instant feedback that things are OK or not. Or maybe you're into TDD.

Historically, you've had to save Salesforce metadata up to an org to be able to run tests. Apex tests were ok, but UI and component tests were more challenging. LWC brings modern testing to go with your modern JS and modern web standards, so you get instant feedback. No cloud necessary!

Open **BrokenComponent.html** in the **brokenComponent** component
It's nearly identical to our contactCard component we used earlier, but it's been broken on purpose.

Let's see the test. Open **BrokenComponent.test.js .**
You'll see that we import the component we want to test, and the data service (getRecord) that it uses.

```
import ContactCard from 'c/brokenComponent';
import { getRecord } from 'lightning/uiRecordApi';
```

You'll also see it tests the component's output.

```
expect(contactTitle).toBeTruthy();
expect(contactTitle.textContent).toBe('CEO');
```

Let's run our brokenComponent test and see it fail. In the VSCode Terminal (ctrl-`)
`npm run test:unit:watch`

To use mock data to run our test, we need to...

Import the test adapter and our fake data
Add the following imports

```
import { registerLdsTestWireAdapter } from '@salesforce/wire-service-jest-util';
import * as fakeData from './data/contactMock.json';
```

Tell our test that we'll be using the adapter
Add this line just inside the describe

```
const getRecordWireAdapter = registerLdsTestWireAdapter(getRecord);
```

Impersonate the wire service.
In our test, **add** this just above the returned Promise

```
getRecordWireAdapter.emit(fakeData);
```

This starts our test runner, and every code change will cause the tests to run again and give us feedback.

*Let's look at **brokenComponent.html**. One of these fields is not like the other...notice **title** isn't capitalized. If you're used to working in Apex, you're probably not case-sensitive, but JavaScript is case-sensitive.*

*Capitalize **Title** and **Save***

```
<p class="contact-title">{contact.data.fields.Title.value}</p>
```

Our test runs again, and we're passing.
Local testing, super fast. Also works on headless CI servers for when people who didn't test locally check in bad code.

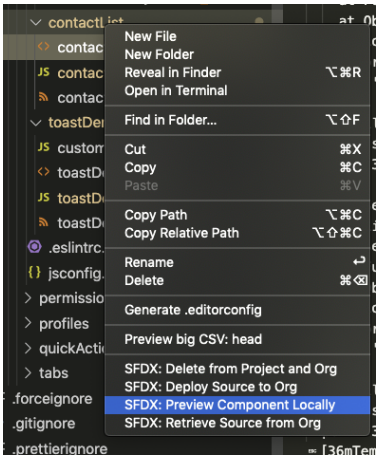
```
import ContactCard from 'c/brokenComponent';
import { getRecord } from 'lightning/uiRecordApi';
import { registerLdsTestWireAdapter } from '@salesforce/wire-service-jest-util';
import * as fakeData from './data/contactMock.json';



describe('@wire demonstration test', () => {
  const getRecordWireAdapter = registerLdsTestWireAdapter(getRecord);
  Run Test | Debug Test
  it('displays the correct title field', () => {
    const element = createElement('c-broken-component', { is: ContactCard });
    document.body.appendChild(element);

    getRecordWireAdapter.emit(fakeData);
    return Promise.resolve().then(() => {
      const contactTitle = element.shadowRoot.querySelector('.contact-title');
      expect(contactTitle).toBeTruthy();
      expect(contactTitle.textContent).toBe('CEO');
    });
  });

  afterEach(() => {
    while (document.body.firstChild) {
      document.body.removeChild(document.body.firstChild);
    }
  });
});
```

#6 Bonus material

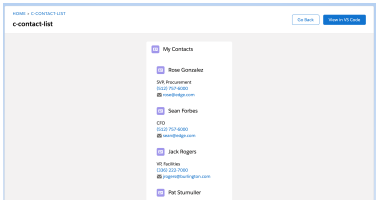
We have local development for LWC in beta (as of Winter '20)

Setup:
In the VSCode terminal run
`sfdx plugins:install @salesforce/lwc-dev-server`

Right-Click on your completed contactList folder and select **Preview Component Locally**



Boom...the component opens in a local dev server.
It's getting data from your org, but you don't have to push the code to preview it.



Change something on **ContactList** (ex: set the title to **My Contacts** instead of **Contacts**)

Boom...it's hot-reloading your changes so you can preview as you code.

In the browser, click **Go Back**.
You can see all the components in your source as you build.

* it's a separate plugin for now, will become part of the "main" sfdx when it goes GA

Wrap up

LWC gives developers some new features

- Better tooling
- Easy access to data via data service
- Simple syntax for apex
- Interoperability with Lightning Components that use Aura
- Local testing
- Local development

The developer guide, playground, and component library are great resources to learn more about LWC.