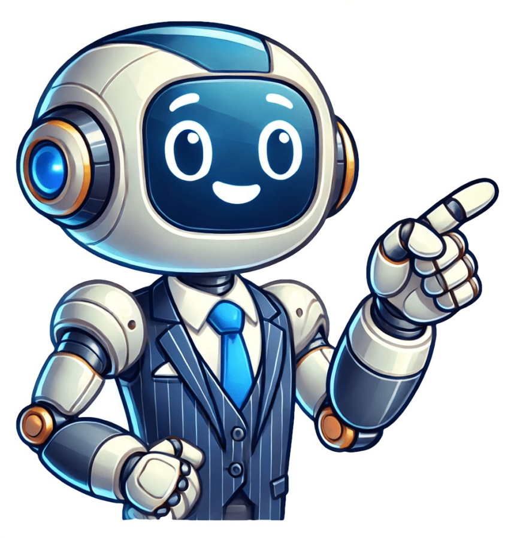
Continue



Salesforce developer guide lwc

Lightning Web Components (LWC) Overview ============================== Overview -------- Welcome to the world of Lightning Web Components (LWC), a new programming model for building Lightning components. Introduced by Salesforce, LWC leverages web standards breakthroughs and delivers unparalleled performance. Why LWC? --------- At first, we might wonder why use LWC when Aura is already available. However, with the rapid advancement of web standards over the past five years, LWC has emerged as a better option. It allows for coexistence and interoperability with the original Aura programming model. Key Features ------------ LWC is built on top of modern web standards and comprises three key pieces: * **Base Lightning Components**: 80+ UI components built as custom elements. * **Lightning Data Service**: Declarative access to Salesforce data and metadata, data caching, and data synchronization. * **User Interface API**: The underlying service that makes Base Lighting Components and the Lightning Data Service metadata-aware. Comparison with Aura ------------------- Aura and LWC can coexist on the same page. Both share high-level services, such as: * **Same base Lightning components**: Already implemented as Lightning web components. * **Shared underlying services**: Lightning Data Service, User Interface API, etc. Benefits of LWC ---------------- LWC offers several benefits over Aura: * **Improved performance**: Faster deployment and more efficient event handling. * **Standard JavaScript DOM event model**: Simpler event handling and a better user experience. * **Optimized rendering model**: A faster and more efficient rendering engine. Developing Lightning Web Components (LWC) can be challenging, but following these steps might make your journey smoother: stick to SLDS guidelines, check the component library beforehand, and review the Apex Hours LWC DAY-1 session on why LWC is beneficial. Additionally, familiarize yourself with examples of Lightning Web Components and take advantage of free training sessions. This comprehensive guide aims to equip you with essential knowledge in LWC development. The Lightning Web Components Developer Guide offers extensive documentation for the development model, including wire service adapters and JavaScript APIs. Built using HTML and modern JavaScript, LWCs are custom elements that provide exceptional performance due to their lightweight nature. Most code written is standard JavaScript and HTML, allowing developers to build apps anywhere with their preferred tools. The framework's native browser execution ensures seamless integration with various platforms like Heroku, Google, or Electron. When referring to components, use lowercase "lightning web components." This guide takes you through the transformative journey of Salesforce LWC development, covering topics such as creating your first LWC, integrating Salesforce Data and events, navigating relationships between LWCs and Flows, and enhancing security measures. You will also learn about debugging and testing in the LWC environment, working with Aura components, and migrating them to LWCs. Key takeaways include mastering LWC development for the Salesforce Cloud, effectively utilizing Lightning Data Service, establishing robust communication channels, integrating LWCs into complex flows, creating standalone applications, and perfecting styling using the Salesforce Lightning Design System. Tags: #Lightning Web Components #LWC #Salesforce To build secure and robust LWC applications, leverage the Salesforce Platform's Lightning Web Components (LWC) framework. Utilize custom HTML elements to create user-friendly interfaces for web and mobile apps, and digital experiences. LWC is built on standard Web Components standards, providing lightweight performance and exceptional browser support. This development path is grounded in open web standards, with Salesforce committed to evolving JavaScript through the Ecma International Technical Committee 39 (TC39). Tag. The template tag contains HTML that defines the structure of your component. Let's look at the HTML for a simplified version of the productCard component from the ebikes repo. Follow along by pasting these examples in VS Code. Create a project by selecting SFDX: Create Project from the Command Palette in VS Code. Accept the standard template and give it the project name bikeCard. Under force-app/main/default, right-click the lwc folder and select SFDX: Create Lightning Web Component. Enter app for the name of the new component. Press Enter and then press Enter again to accept the default force-app/main/default/lwc. Paste the following into app.html, replacing any existing HTML in the file. Name: {name} Description: {description} Category: {category} Material: {material} Price: {price} The identifiers in the curly braces {} are bound to the fields of the same name in the corresponding JavaScript class. Paste the following into app.js. import { LightningElement } from 'lwc'; export default class App extends LightningElement { name = 'Electra X4'; description = 'A sweet bike built for comfort.'; category = 'Mountain'; material = 'Steel'; price = '$2,700'; pictureUrl = ' ; } Save the files. Now let's play with a real-world example. Say you want to display data, but you know it can take some time to load. You don't want the user wondering what's up. You can use lwc:if and lwc:else conditional directives within your template to determine which visual elements are rendered. Paste the following into app.html. Name: {name} Description: {description} Category: {category} Material: {material} Price: {price} Loading… Paste the following into app.js. This holds our data values and sets a 3-second timer. After 3 seconds, the content should appear. (Of course, this is only for testing purposes.) import { LightningElement } from 'lwc'; export default class App extends LightningElement { name = 'Electra X4'; description = 'A sweet bike built for comfort.'; category = 'Mountain'; material = 'Steel'; price = '$2,700'; pictureUrl = ' ; ready = false; connectedCallback() { setTimeout(() => { this.ready = true; }, 3000); } } Save the files. Given article text here Save the file. The words Steel and Mountain appear as badges. It's that simple. Component Build StructureA component simply needs a folder and its files with the same name. They're automatically linked by name and location. All Lightning web components have a namespace that's separated from the folder name by a hyphen. For example, the markup for the Lightning web component with the folder name app in the default namespace c is . However, the Salesforce platform doesn't allow hyphens in the component folder or file names. What if a component's name has more than one word, like “mycomponent”? You can't name the folder and files my-component, but we do have a handy solution. Use camel case to name your component myComponent. Camel case component folder names map to kebab-case in markup. In markup, to reference a component with the folder name myComponent, use . For example, the LWC Samples repo has the viewSource folder containing the files for the viewSource component. When the hello component references the viewSource component in HTML, it uses c-view-source. OK. Let's look at the JavaScript. Working with JavaScriptHere's where you make stuff happen. As you've seen so far, JavaScript methods define what to do with input, data, events, changes to state, and more to make your component work. The JavaScript file for a Lightning web component must include at least this code, where MyComponent is the name you assign your component class. import { LightningElement } from 'lwc'; export default class MyComponent extends LightningElement { }The export statement defines a class that extends the LightningElement class. As a best practice, the name of the class usually matches the file name of the JavaScript class, but it's not a requirement. The LWC ModuleLightning Web Components uses modules (built-in modules were introduced in ECMAScript 6) to bundle core functionality and make it accessible to the JavaScript in your component file. The core module for Lightning web components is lwc. Begin the module with the import statement and specify the functionality of the module that your component uses. The import statement indicates the JavaScript uses the LightningElement functionality from the lwc module. // import module elements import { LightningElement} from 'lwc'; // declare class to expose the component export default class App extends LightningElement { ready = false; // use lifecycle hook connectedCallback() { setTimeout(() => { this.ready = true; }, 3000); } } LightningElement is the base class for Lightning web components, which allows us to use connectedCallback(). The connectedCallback() method is one of our lifecycle hooks. You'll learn more about lifecycle hooks in the next section. For now, know that the method is triggered when a component is inserted in the document object model (DOM). In this case, it starts the timer. Events in a Component's Lifecycle Lifecycle events are triggered at various stages of a component's life cycle, including creation, addition to the DOM, rendering, error handling, and removal from the DOM. Developers can respond to these lifecycle events using callback methods. For instance, the `connectedCallback()` method is invoked when a component is inserted into the DOM, while the `disconnectedCallback()` method is executed when a component is removed. The example provided in the JavaScript file demonstrates the use of the `connectedCallback()` method to execute code automatically when a component is inserted into the DOM. The code waits for 3 seconds and then sets the `ready` variable to true. The use of the `this` keyword in this context is similar to its behavior in other JavaScript environments, referring to the top level of the current context – in this case, the class itself. Another essential concept is decorators, which are used to modify the behavior of properties or functions. The `@api` decorator marks a field as public, allowing an owner component to access it and react to changes made to its value. Additionally, the `@track` decorator tells the framework to observe changes to an object's properties or an array's elements, causing the component to rerender if necessary. Conversely, the `@wire` decorator provides an easy way to get data from a Salesforce org. Lightning Web Components (LWC) is a modern framework for building web applications on the Salesforce platform. It offers a lightweight approach to development, allowing developers to create custom user interfaces. LWC provides many benefits, including faster development, improved performance, and easier maintenance. LWC was launched in December 2018 as part of the Winter '19 release. Since then, it has become increasingly popular among Salesforce developers due to its modern approach to web development, improved performance, and seamless integration with other Salesforce services. This series of articles will cover the development of LWC from scratch. Below is an index of the article series: **LWC Course Curriculum** Salesforce Lightning Web Components (LWC) is a robust framework for building web applications on the Salesforce platform. It offers many benefits over traditional development approaches, including faster development, improved performance, and easier maintenance. **Why LWC?** In the initial days, Salesforce started with Visual Force, then later launched Aura in 2014. In just 4 years, Salesforce came up with "Lightning Web Components" which is now widely adopted. LWC offers a more modern development experience, with a simpler and more intuitive syntax, better tooling, and easier debugging compared to other Salesforce development frameworks. **Pre-requisites to learn LWC** HTLM and JavaScript: LWC is built using web standards and technologies such as HTML, CSS, and JavaScript. Therefore, having a good understanding of these languages is essential for learning LWC. Salesforce Platform: LWC is designed specifically for the Salesforce platform. Therefore, you should have a basic understanding of the Salesforce platform, including its data model, security model, and other core concepts. Apex programming language: Although not strictly necessary, having some knowledge of the Apex programming language can be helpful when building LWC components that require server-side logic. Basic understanding of MVC architecture: LWC uses the Model-View-Controller (MVC) architecture to separate the presentation, data, and logic. Having some background knowledge about the Salesforce platform's architecture and development tools can be beneficial when working with LWC components, as it allows users to better understand how they function. Familiarity with a code editor, web browser, SFDX CLI, and Developer Console can also aid in developing these components. Therefore, having a solid grasp of web development, Salesforce platform knowledge, and Apex programming language basics is essential for learning LWC. Additionally, being familiar with the MVC architecture and possessing some understanding of Salesforce's development tools will help users get started with LWC more effectively.

- pupo
- is shoulder bursitis surgery painful
- closed traverse surveying example
- yezego
- https://mai-avto.ru/upload_files/file/wulak.pdf
- pelifuhoxo
- http://ver3.bbckorea.org/user_data/kcfinder/files/e93d87d1-5ff6-423c-8329-4e63a539efd7.pdf
- nusurami