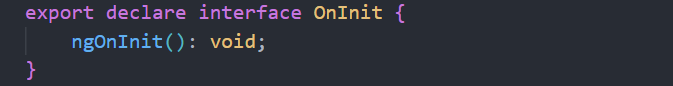
**Component Lifecycle Hooks**

Every component has a lifecycle, like us 🙂. The lifecycle starts when a component is instantiated, continues with change detection, and ends when the component template is removed from the DOM.

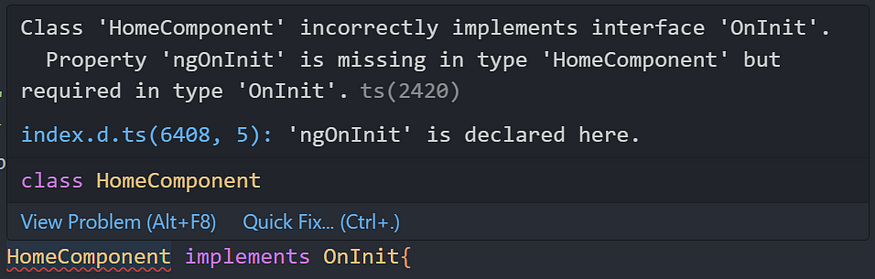
* Directives have a similar lifecycle.
* Angular gives us **lifecycle hook methods** to take advantage of events in the lifecycle.

**How can we use these lifecycle methods?**

* It’s simple; all you have to do is implement these lifecycle hook **interfaces**. You can import these interfaces from the **@angular/core** library.
* Every lifecycle interface has only 1 method. There is standardization for naming methods. It takes an **ng** prefix with the interface name. For example, look at the OnInit interface from the **@angular/core** library.



* If you implement an interface, TypeScript will inform you to use its method.



* You don’t have to implement all interfaces. **Only when you need it**, at initialization time or before deleting the component?
* you will be able to choose right one after reading this post 🙂

**Sequence of Lifecycle methods**

1. ngOnChanges
2. ngOnInit
3. ngDoCheck
4. ngAfterContentInit
5. ngAfterContentChecked
6. ngAfterViewInit
7. ngAfterViewChecked
8. ngOnDestroy

* When the component is rendered to the template, these methods will run at the appropriate time. Of course, above all, **the constructor runs first**.

**Let’s look at each method one by one**

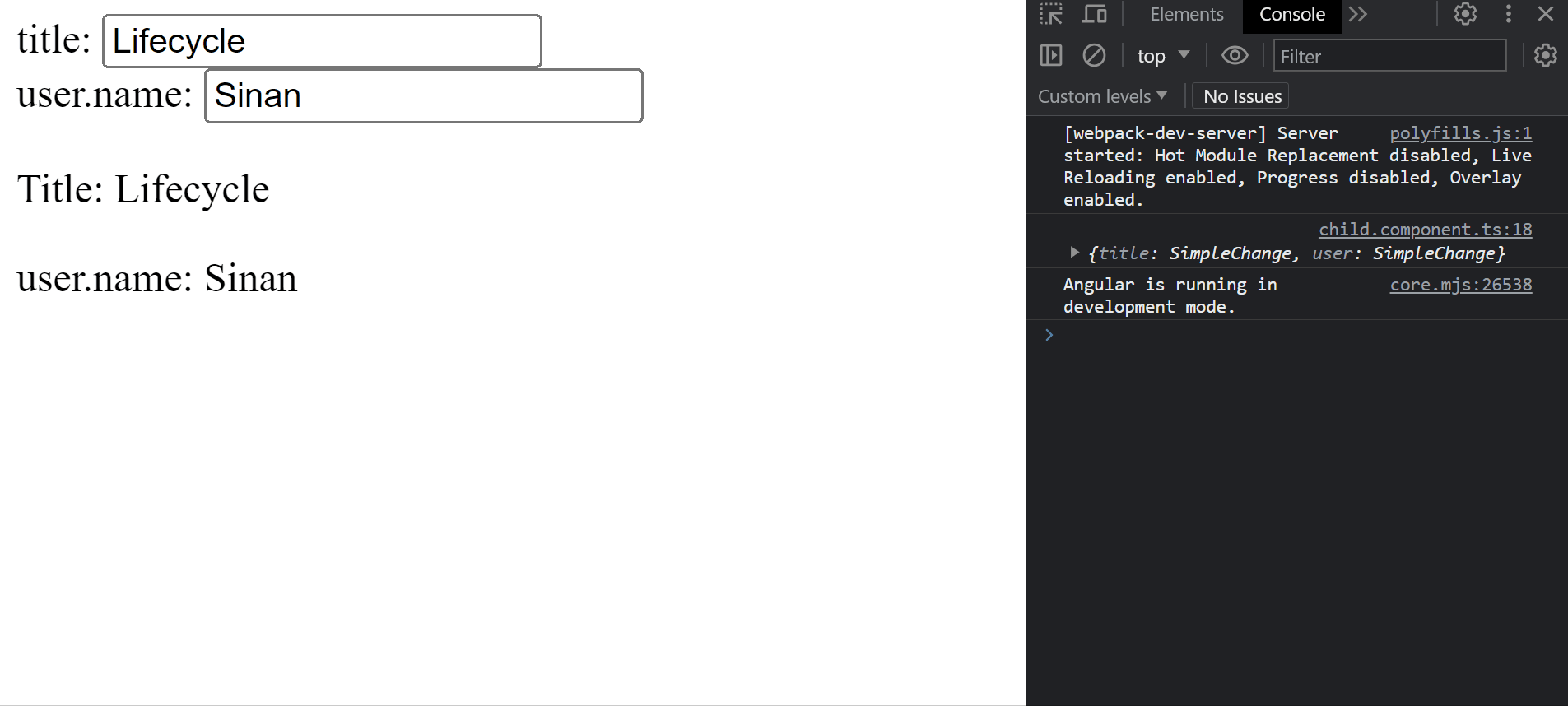
**ngOnChanges()**

* This one is called before **ngOnInit**, **only if you have an input value.** If you don't have an input value, this method won't run.
* When the input value resets or updates, this method will be triggered. ngOnChanges method gets a **SimpleChanges** object, which shows the **currentValue, firstChange, previousValue.**
* Your input values might change frequently, so making an HTTP request or something heavy might affect your application’s performance.
* By the way, **If your input is an object and if you change its property it wont catch by Angular, because it still referencing to the same address**. Let me show you an example.





* In the child component, we capture the input changes via the ngOnChanges method, **and we only receive changed inputs in the changes object, which has a SimpleChanges interface**.
* As you can see below, Angular couldn’t detect the user object’s name property changes.



* I will show you a way to detect it in the **ngDoCheck** lifecycle method.

**ngOnInit()**

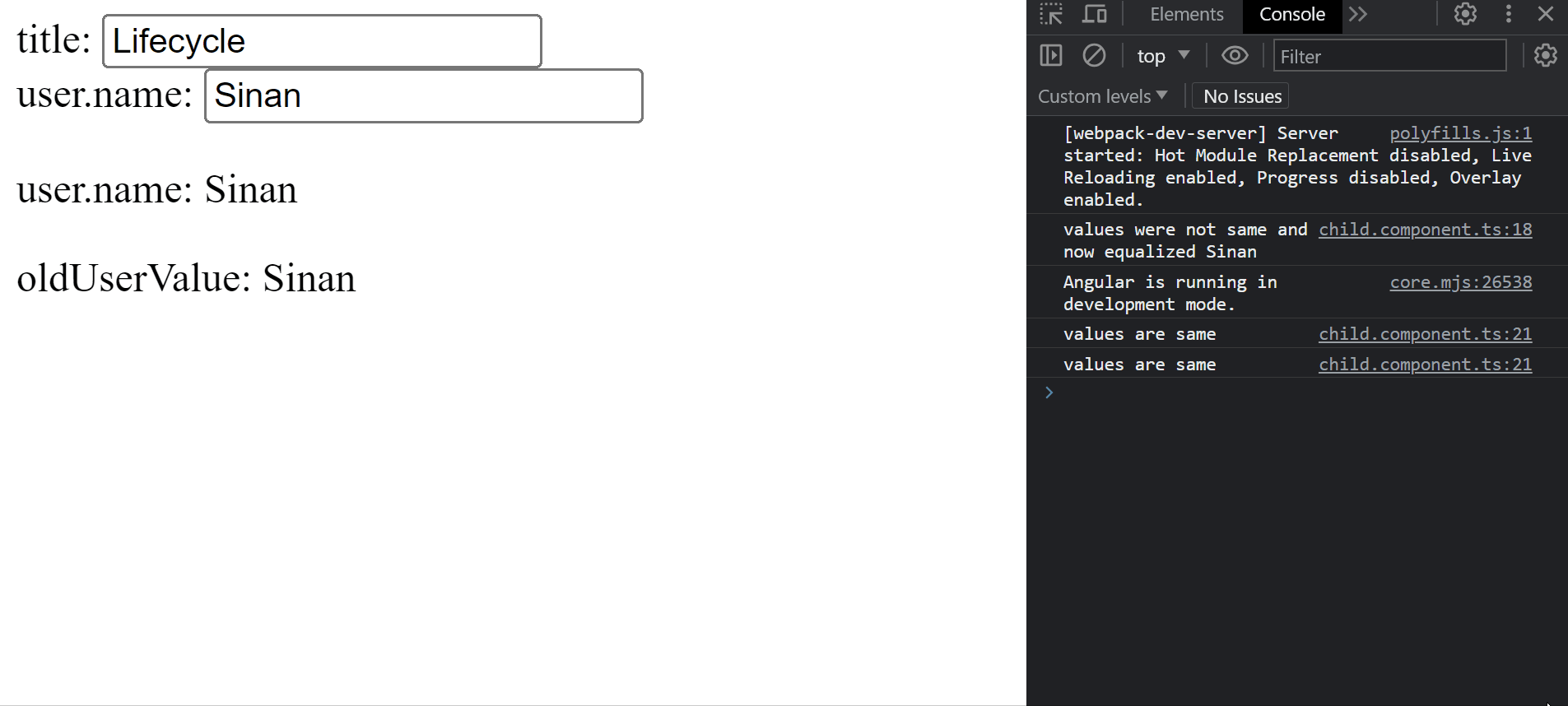
* This method is **called only once** during the component lifecycle and is the most used hook in Angular.
* It is called after the **ngOnChanges** method. Even if ngOnChanges is not called, it will still be called.
* We can use ngOnInit for initialization tasks. This is a good place for fetching data from a server, and **it’s a best practice**.
* You may ask why not the constructor? Because the constructor should be used only for binding values, **and you can’t reach input variables in the constructor.**

**ngDoCheck()**

* this method is **called almost every change detection.**
* In previous video i have shown you that angular couldn’t catch the input object property changes, **so we can implement our own custom change detection here.**



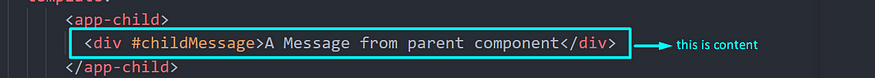
* When **user.name** value changes **we catch it via ngDoCheck method.**



* As shown at the end of the video, this method is **called almost every change detection, so you shouldn’t do heavy things here.** It will be bad for your website’s performance.

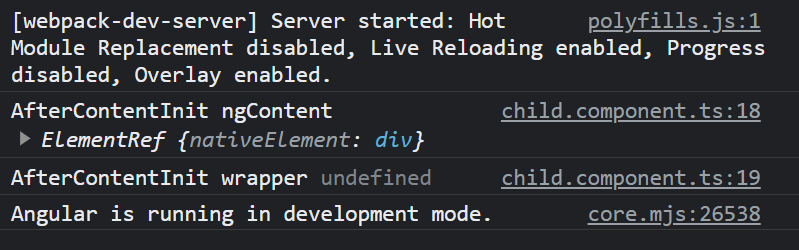
**ngAfterContentInit()**

* This hook is called only once after the first **ngDoCheck().**
* Before going in, I should explain what **content** is. If you are putting HTML elements inside the child component tags, it means you are **projecting content** to the child.
* Catch this content with the ng-content tag in the child component.



* So if you are working with ng-content and want to do something after this content is initialized you can use this lifecycle hook. Because this is the first time that we can access to the **ElementRef**of the**ContentChild**.
* **in this hook we only have access to the projected content**. while the template is not initialized yet we cannot access any other elements. it will be available to access on the ngAfterViewInit hook.
* Let me show you child component, and the logs.

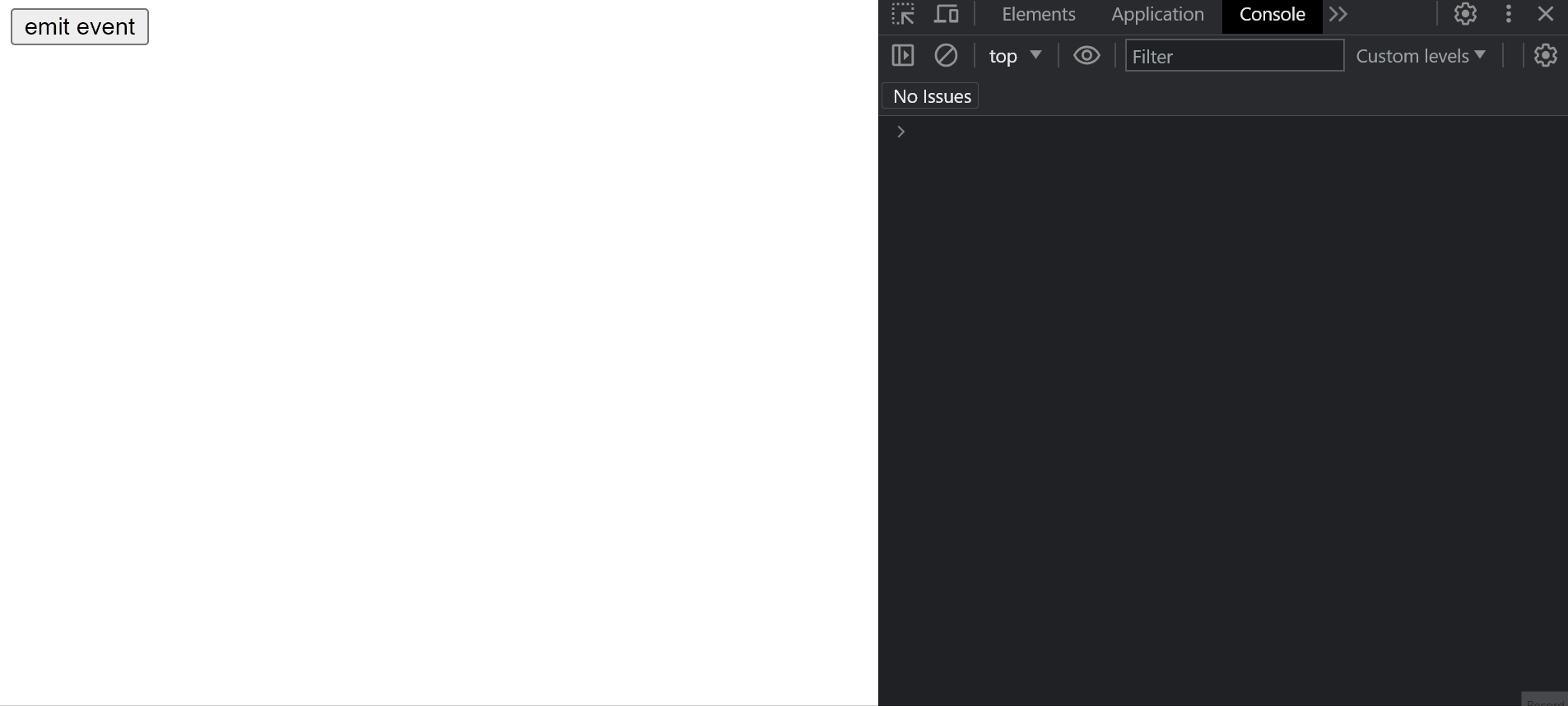




* As can be seen from the logs, we can access contents in this hook.

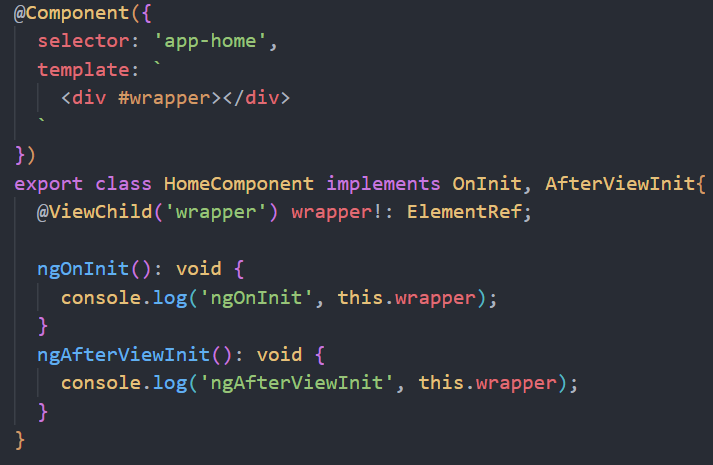
**ngAfterContentChecked()**

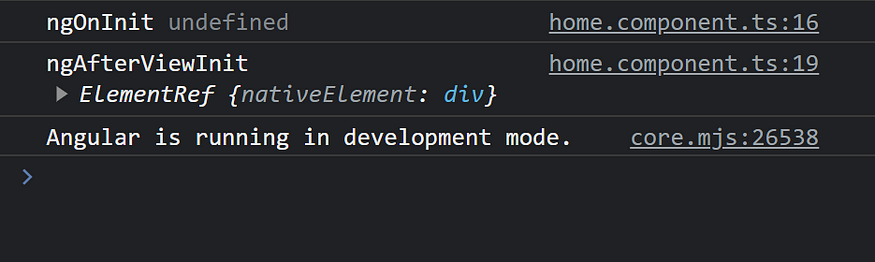
* This hook will be called after **ngAfterContentInit** and after every **ngDoCheck**.
* You may use this hook to react to changes in the projected content.
* **You shouldn’t do heavy things here** because **ngDoCheck** triggers this hook.



**ngAfterViewInit()**

* This hook is called only once after the first **ngAfterContentChecked**. With this hook, we understand that the component and child views are initialized.
* This is the first time that we can access the **ElementRef** of the **ViewChildren**.





* In Angular, the change detection starts from parent to child. If a child component tries to change something on the parent component, you get an **ExpressionChangedAfterItHasBeenCheckedError** error. So, make sure you don't do that in the ngAfterViewInit hook. 🙂

**ngAfterViewChecked()**

* This hook will be called after **ngAfterViewInit** and after every **ngAfterContentChecked**.
* This is called after angular checked component views and its child views.
* The triggering chain is like this;
* **ngDoCheck**→ **ngAfterContentChecked**→ **ngAfterViewChecked** so like others, this hook will be called almost every change detection. **You shouldn’t make heavy operations in this method.**

**ngOnDestroy()**

* **Called before Angular removes the component template from the DOM**.
* You can use this hook to unsubscribe from observables (**use async pipe as much as possible**), clean up your intervals, timeouts, local storage, etc. Eventually, this is a good place to handle memory leaks.

**The picture below shows the relationship between the hooks.**

