Angular

Qinshift Academy

Signals

Qinshift Academy

What are signals?

A signal is a wrapper around a value that notifies interested consumers when that value changes. Signals can contain any value, from primitives to complex data structures.

You read a signal's value by calling its getter function, which allows Angular to track where the signal is used.

Signals may be either writable or read-only.

Writable signals

Writable signals provide an API for updating their values directly. You create writable signals by calling the signal function with the signal's initial value:

```
const count = signal(0);

// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());
```

To change the value of a writable signal, either .set() it directly:

```
count.set(3);
```

or use the .update() operation to compute a new value from the previous one:

```
// Increment the count by 1.
count.update(value => value + 1);
```

Writable signals have the type WritableSignal.

Computed signals

Computed signal are read-only signals that derive their value from other signals. You define computed signals using the computed function and specifying a derivation:

```
const count: WritableSignal<number> = signal(0);
const doubleCount: Signal<number> = computed(() => count() * 2);
```

The doubleCount signal depends on the count signal. Whenever count updates, Angular knows that doubleCount needs to update as well.

Computed signals are both lazily evaluated and memoized

doubleCount's derivation function does not run to calculate its value until the first time you read doubleCount. The calculated value is then cached, and if you read doubleCount again, it will return the cached value without recalculating.

If you then change count, Angular knows that doubleCount's cached value is no longer valid, and the next time you read doubleCount its new value will be calculated.

As a result, you can safely perform computationally expensive derivations in computed signals, such as filtering arrays.

Computed signals are not writable signals

You cannot directly assign values to a computed signal. That is,

```
doubleCount.set(3);
```

produces a compilation error, because doubleCount is not a WritableSignal.

Computed signal dependencies are dynamic

Only the signals actually read during the derivation are tracked. For example, in this computed the count signal is only read if the showCount signal is true:

```
const showCount = signal(false);
const count = signal(0);
const conditionalCount = computed(() => {
  if (showCount()) {
    return `The count is ${count()}.`;
  } else {
    return 'Nothing to see here!';
  }
});
```

When you read conditionalCount, if showCount is false the "Nothing to see here!" message is returned without reading the count signal. This means that if you later update count it will not result in a recomputation of conditionalCount.

If you set showCount to true and then read conditionalCount again, the derivation will re-execute and take the branch where showCount is true, returning the message which shows the value of count. Changing count will then invalidate conditionalCount's cached value.

Note that dependencies can be removed during a derivation as well as added. If you later set showCount back to false, then count will no longer be considered a dependency of conditionalCount.

Reading signals in OnPush components

When you read a signal within an OnPush component's template, Angular tracks the signal as a dependency of that component. When the value of that signal changes, Angular automatically marks the component to ensure it gets updated the next time change detection runs.

Effects

Signals are useful because they notify interested consumers when they change. An effect is an operation that runs whenever one or more signal values change. You can create an effect with the effect function:

```
effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

Effects always run at least once. When an effect runs, it tracks any signal value reads. Whenever any of these signal values change, the effect runs again. Similar to computed signals, effects keep track of their dependencies dynamically, and only track signals which were read in the most recent execution.

Effects always execute asynchronously, during the change detection process.

Use cases for effects

Effects are rarely needed in most application code, but may be useful in specific circumstances. Here are some examples of situations where an effect might be a good solution:

- Logging data being displayed and when it changes, either for analytics or as a debugging tool.
- Keeping data in sync with window.localStorage.
- Adding custom DOM behavior that can't be expressed with template syntax.
- Performing custom rendering to a <canvas>, charting library, or other third party UI library.

When not to use effects

Avoid using effects for propagation of state changes. This can result in ExpressionChangedAfterItHasBeenChecked errors, infinite circular updates, or unnecessary change detection cycles.

Because of these risks, Angular by default prevents you from setting signals in effects. It can be enabled if absolutely necessary by setting the allowSignalWrites flag when you create an effect.

Instead, use computed signals to model state that depends on other state.

Injection context

By default, you can only create an effect() within an injection context (where you have access to the inject function). The easiest way to satisfy this requirement is to call effect within a component, directive, or service constructor:

```
@Component({...})
export class EffectiveCounterComponent {
  readonly count = signal(0);
  constructor() {
    // Register a new effect.
    effect(() => {
      console.log(`The count is: ${this.count()}`);
    });
  }
}
```

Injection context

Alternatively, you can assign the effect to a field (which also gives it a descriptive name).

```
@Component({...})
export class EffectiveCounterComponent {
  readonly count = signal(0);

  private loggingEffect = effect(() => {
    console.log(`The count is: ${this.count()}`);
  });
}
```

To create an effect outside of the constructor, you can pass an Injector to effect via its options:

```
@Component({...})
export class EffectiveCounterComponent {
  readonly count = signal(0);
  constructor(private injector: Injector) {}

  initializeLogging(): void {
    effect(() => {
      console.log(`The count is: ${this.count()}`);
    }, {injector: this.injector});
  }
}
```

Destroying effects

When you create an effect, it is automatically destroyed when its enclosing context is destroyed. This means that effects created within components are destroyed when the component is destroyed. The same goes for effects within directives, services, etc.

Effects return an EffectRef that you can use to destroy them manually, by calling the .destroy() method. You can combine this with the manualCleanup option to create an effect that lasts until it is manually destroyed. Be careful to actually clean up such effects when they're no longer required.

Reading without tracking dependencies

Rarely, you may want to execute code which may read signals within a reactive function such as computed or effect without creating a dependency.

For example, suppose that when currentUser changes, the value of a counter should be logged. you could create an effect which reads both signals:

```
effect(() => {
  console.log(`User set to ${currentUser()} and the counter is ${counter()}`);
});
```

This example will log a message when either currentUser or counter changes. However, if the effect should only run when currentUser changes, then the read of counter is only incidental and changes to counter shouldn't log a new message.

Reading without tracking dependencies

You can prevent a signal read from being tracked by calling its getter with untracked:

```
effect(() => {
  console.log(`User set to ${currentUser()} and the counter is ${untracked(counter)}`);
});
```

untracked is also useful when an effect needs to invoke some external code which shouldn't be treated as a dependency:

```
effect(() => {
  const user = currentUser();
  untracked(() => {
    // If the `loggingService` reads signals, they won't be counted as
    // dependencies of this effect.
    this.loggingService.log(`User set to ${user}`);
  });
});
```

Effect cleanup functions

Effects might start long-running operations, which you should cancel if the effect is destroyed or runs again before the first operation finished. When you create an effect, your function can optionally accept an onCleanup function as its first parameter. This onCleanup function lets you register a callback that is invoked before the next run of the effect begins, or when the effect is destroyed.

```
effect((onCleanup) => {
  const user = currentUser();

  const timer = setTimeout(() => {
    console.log(`1 second ago, the user became ${user}`);
  }, 1000);

  onCleanup(() => {
    clearTimeout(timer);
  });
});
```

Questions?

Trainer Name

Trainer

trainer@mail.com

Assistant Name

Assistant

assistant@mail.com

Qinshift Academy