NgZone

A zone is an execution context that persists across async tasks. You can think of it as thread-local storage for JavaScript VMs. This guide describes how to use Angular's NgZone to automatically detect changes in the component to update HTML.

Fundamentals of change detection

To understand the benefits of `NgZone`, it is important to have a clear grasp of what change detection is and how it works.

Displaying and updating data in Angular

In Angular, you can display data by binding controls in an HTML template to the properties of an Angular component.

src/app/app.component.ts

```typescript
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
    `
})
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

In addition, you can bind DOM events to a method of an Angular component. In such methods, you can also update a property of the Angular component, which updates the corresponding data displayed in the template.

src/app/click-me.component.ts

```
@Component({
  selector: 'app-click-me',
  template: `
    <button (click)="onClickMe()">Click
me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe() {
    this.clickMessage = 'You are my
hero!';
  }
}
```

In both of the above examples, the component's code updates only the property of the component.

However, the HTML is also updated automatically. This guide describes how and when Angular renders the HTML based on the data from the Angular component.

Detecting changes with plain JavaScript

To clarify how changes are detected and values updated, consider the following code written in plain JavaScript.

```
<html>
  <div id="dataDiv"></div>
  <button id="btn">updateData<btn>
  <canvas id="canvas"><canvas>
  <script>
    let value = 'initialValue';
    // initial rendering
    detectChange();

    function renderHTML() {

document.getElementById('dataDiv').inner
 = value;
    }

    function detectChange() {
      const currentValue =
document.getElementById('dataDiv').inner

      if (currentValue !== value) {
        renderHTML();
      }
    }

    // example 1: update data inside
button click event handler

document.getElementById('btn').addEventL
 () => {
      // update value
      value = 'button update value';
      // call detectChange manually
      detectChange();
    });

    // example 2: Http Request
    const xhr = new XMLHttpRequest();
    xhr.addEventListener('load',
function() {
      // get response from server
      value = this.responseText;
      // call detectChange manually
      detectChange();
    });
    xhr.open('GET', serverUrl);
    xhr.send();

    // example 3: setTimeout
    setTimeout(() => {
      // update value inside setTimeout
callback
      value = 'timeout update value';
      // call detectChange manually
      detectChange();
    }, 100);

    // example 4: Promise.then
    Promise.resolve('promise resolved a
value').then((v) => {
      // update value inside Promise
thenCallback
      value = v;
      // call detectChange manually
      detectChange();
    }, 100);

    // example 5: some other
asynchronous APIs

document.getElementById('canvas').toBlob
 => {
      // update value when blob data is
created from the canvas
      value = `value updated by canvas,
size is ${blog.size}`;
      // call detectChange manually
      detectChange();
    });
  </script>
</html>
```

After you update the data, you need to call `detectChange()` manually to check whether the data changed. If the data changed, you render the HTML to reflect the updated data.

In Angular, this step is unnecessary. Whenever you update the data, your HTML is updated automatically.

When apps update HTML

To understand how change detection works, first consider when the application needs to update the HTML. Typically, updates occur for one of the following reasons:

1. Component initialization. For example, when bootstrapping an Angular application, Angular loads the bootstrap component and triggers the ApplicationRef.tick() to call change detection and View Rendering. Just as in the displaying data sample, the `AppComponent` is the bootstrap component. This component has the properties `title` and `myHero`, which the application renders in the HTML.

2. Event listener. The DOM event listener can update the data in an Angular component and also trigger change detection, as in the following example.

src/app/click-me.component.ts

```
@Component({
  selector: 'app-click-me',
  template: `
    <button (click)="onClickMe()">Click
me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';

  onClickMe() {
    this.clickMessage = 'You are my
hero!';
  }
}
```

3. Http Data Request. You can also get data from a server through an Http request. For example:

```
@Component({
  selector: 'app-root',
  template: '<div>{{data}}</div>';
})
export class AppComponent implements
OnInit {
  data = 'initial value';
  constructor(private httpClient:
HttpClient) {}

  ngOnInit() {

this.httpClient.get(serverUrl).subscribe
 => {
      // user does not need to trigger
change detection manually
      data = response.data;
    });
  }
}
```

4. MacroTasks, such as `setTimeout()`/`setInterval()`. You can also update the data in the callback function of `macroTask` such as `setTimeout()`. For example:

```
@Component({
  selector: 'app-root',
  template: '<div>{{data}}</div>';
})
export class AppComponent implements
OnInit {
  data = 'initial value';

  ngOnInit() {
    setTimeout(() => {
      // user does not need to trigger
change detection manually
      data = 'value updated';
    });
  }
}
```

5. MicroTask, such as `Promise.then()`. Other asynchronous APIs return a Promise object (such as `fetch`), so the `then()` callback function can also update the data. For example:

```
@Component({
  selector: 'app-root',
  template: '<div>{{data}}</div>';
})
export class AppComponent implements
OnInit {
  data = 'initial value';

  ngOnInit() {
    Promise.resolve(1).then(v => {
      // user does not need to trigger
change detection manually
      data = v;
    });
  }
}
```

6. Other async operations. In addition to `addEventListener()`/`setTimeout()`/`Promise.the` there are other operations that can update the data asynchronously. Some examples include `WebSocket.onmessage()` and `Canvas.toBlob()`.

The preceding list contains most common scenarios in which the application might change the data. Angular runs change detection whenever it detects that data could have changed. The result of change detection is that DOM is updated with new data. Angular detects the changes in different ways. For component initialization, Angular calls change detection explicitly. For asynchronous operations, Angular uses a Zone to detect changes in places where the data could have possibly mutated and it runs change detection automatically.

Zones and execution contexts

A zone provides an execution context that persists across async tasks. Execution Context is an abstract concept that holds information about the environment within the current code being executed. Consider the following example.

```
const callback = function() {
  console.log('setTimeout callback
context is', this);
}

const ctx1 = {
  name: 'ctx1'
};
const ctx2 = {
  name: 'ctx2'
};

const func = function() {
  console.log('caller context is',
this);
  setTimeout(callback);
}

func.apply(ctx1);
func.apply(ctx2);
```

The value of `this` in the callback of `setTimeout` might differ depending on when `setTimeout` is called. Thus you can lose the context in asynchronous operations.

A zone provides a new zone context other than `this`, the zone context persists across asynchronous operations. In the following example, the new zone context is called `zoneThis`.

```
zone.run(() => {
  // now you are in a zone
  expect(zoneThis).toBe(zone);
  setTimeout(function() {
    // the zoneThis context will be the
same zone
    // when the setTimeout is scheduled
    expect(zoneThis).toBe(zone);
  });
});
```

This new context, `zoneThis`, can be retrieved from the `setTimeout()` callback function, and this context is the same when the `setTimeout()` is scheduled. To get the context, you can call `Zone.current`.

Zones and async lifecycle hooks

Zone.js can create contexts that persist across asynchronous operations as well as provide lifecycle hooks for asynchronous operations.

```
const zone = Zone.current.fork({
  name: 'zone',
  onScheduleTask: function(delegate,
curr, target, task) {
    console.log('new task is scheduled:
', task.type, task.source);
    return
delegate.scheduleTask(target, task);
  },
  onInvokeTask: function(delegate,
curr, target, task, applyThis,
applyArgs) {
    console.log('task will be invoked',
task.type, task.source);
    return delegate.invokeTask(target,
task, applyThis, applyArgs);
  },
  onHasTask: function(delegate, curr,
target, hasTaskState) {
    console.log('task state changed in
the zone', hasTaskState);
    return delegate.hasTask(target,
hasTaskState);
  },
  onInvoke: function(delegate, curr, target, callback, applyThis, applyArgs) {
    console.log('the callback will be invoked', callback);
    return delegate.invoke(target, callback, applyThis, applyArgs);
  }
});
zone.run(() => {
  setTimeout(() => {
    console.log('timeout callback is invoked.');
  });
});
```

The above example creates a zone with several hooks.

`onXXXTask` hooks trigger when the status of Task changes. The Zone Task concept is very similar to the Javascript VM Task concept.

- `macroTask`: such as `setTimeout()`.
- `microTask`: such as `Promise.then()`.
- `eventTask`: such as `element.addEventListener()`.

The `onInvoke` hook triggers when a synchronize function is executed in a Zone.

These hooks trigger under the following circumstances:

- `onScheduleTask`: triggers when a new asynchronous task is scheduled, such as when you call `setTimeout()`.
- `onInvokeTask`: triggers when an asynchronous task is about to execute, such as when the callback of `setTimeout()` is about to execute.
- `onHasTask`: triggers when the status of one kind of task inside a zone changes from stable to unstable or from unstable to stable. A status of stable means there are no tasks inside the Zone, while unstable means a new task is scheduled in the zone.
- `onInvoke`: triggers when a synchronize function is going to execute in the zone.

With these hooks, `Zone` can monitor the status of all synchronize and asynchronous operations inside a zone.

The above example returns the following output.

```
the callback will be invoked () => {
  setTimeout(() => {
    console.log('timeout callback is
invoked.');
  });
}
new task is scheduled:  macroTask
setTimeout
task state changed in the zone {
microTask: false,
  macroTask: true,
  eventTask: false,
  change: 'macroTask' }
task will be invoked macroTask
setTimeout
timeout callback is invoked.
task state changed in the zone {
microTask: false,
  macroTask: false,
  eventTask: false,
  change: 'macroTask' }
```

All of the functions of Zone are provided by a library called zone.js. This library implements those features by intercepting asynchronous APIs through monkey patching. Monkey patching is a technique to add or modify the default behavior of a function at runtime without changing the source code.

NgZone

While Zone.js can monitor all the states of synchronous and asynchronous operations, Angular additionally provides a service called NgZone. This service creates a zone named `angular` to automatically trigger change detection when the following conditions are satisfied:

1. When a sync or async function is executed.
2. When there is no `microTask` scheduled.

NgZone `run()`/`runOutsideOfAngular()`

`Zone` handles most asynchronous APIs such as `setTimeout()`, `Promise.then()`, and `addEventListener()`. For the full list, see the Zone Module document. Therefore in those asynchronous APIs, you don't need to trigger change detection manually.

There are still some third party APIs that Zone does not handle. In those cases, the NgZone service provides a `run()` method that allows you to execute a function inside the angular zone. This function, and all asynchronous operations in that function, trigger change detection automatically at the correct time.

```
export class AppComponent implements
OnInit {
  constructor(private ngZone: NgZone)
{}
  ngOnInit() {
    // New async API is not handled by
Zone, so you need to
    // use ngZone.run() to make the
asynchronous operation in the angular
zone
    // and trigger change detection
automatically.
    this.ngZone.run(() => {
      someNewAsyncAPI(() => {
        // update the data of the
component
      });
    });
  }
}
```

By default, all asynchronous operations are inside the angular zone, which triggers change detection automatically. Another common case is when you don't want to trigger change detection. In that situation, you can use another NgZone method: runOutsideAngular().

```
export class AppComponent implements
OnInit {
  constructor(private ngZone: NgZone)
{}
  ngOnInit() {
    // You know no data will be
updated,
    // so you don't want to trigger
change detection in this
    // specified operation. Instead,
call ngZone.runOutsideAngular()
    this.ngZone.runOutsideAngular(() =>
{
      setTimeout(() => {
        // update component data
        // but don't trigger change
detection.
      });
    });
  }
}
```

Setting up Zone.js

To make Zone.js available in Angular, you need to import the zone.js package. If you are using the Angular CLI, this step is done automatically, and you will see the following line in the `src/polyfills.ts`:

```
/***************************************

 * Zone JS is required by default for
Angular itself.
 */
import 'zone.js/dist/zone';  //
Included with Angular CLI.
```

Before importing the `zone.js` package, you can set the following configurations:

- You can disable some asynchronous API monkey patching for better performance. For example, you can disable the `requestAnimationFrame()` monkey patch, so the callback of `requestAnimationFrame()` will not trigger change detection. This is useful if, in your application, the callback of the `requestAnimationFrame()` will not update any data.
- You can specify that certain DOM events not run inside the angular zone; for example, to prevent a `mousemove` or `scroll` event to trigger change detection.

There are several other settings you can change. To make these changes, you need to create a `zone-flags.ts` file, such as the following.

```
(window as
any).__Zone_disable_requestAnimationFram
 = true; // disable patch
requestAnimationFrame
(window as
any).__zone_symbol__UNPATCHED_EVENTS =
['scroll', 'mousemove']; // disable
patch specified eventNames
```

Next, import `zone-flags` before you import `zone` in the `polyfills.ts`.

```
/*****************************************


 * Zone JS is required by default for
Angular.
 */
import `./zone-flags`;
import 'zone.js/dist/zone';  //
Included with Angular CLI.
```

For more information of what you can configure, see the zone.js documentation.

NoopZone

`Zone` helps Angular know when to trigger change detection and let the developers focus on the application development. By default, `Zone` is loaded and works without additional configuration. However, you don't have to use `Zone` to make Angular work, instead opting to trigger change detection on your own.

> ### Disabling `Zone`
>
> **If you disable `Zone`, you will need to trigger all change detection at the correct timing yourself, which requires comprehensive knowledge of change detection.**

To remove `zone.js`, make the following changes.

1. Remove the `zone.js` import from `polyfills.ts`.

```
/**********************************

 * Zone JS is required by default
for Angular itself.
 */
// import 'zone.js/dist/zone';
// Included with Angular CLI.
```

2. Bootstrap Angular with `noop zone` in `src/main.ts`.

```
platformBrowserDynamic().bootstra
 {ngZone: 'noop'})
  .catch(err =>
console.error(err));
```