What every front-end developer should know about change detection in Angular and React

A comparison of the change detection mechanism by a developer who reverse-engineered both frameworks

Max Koretskyi aka Wizard Follow
Oct 9, 2018 · 12 min read

> *The only way to make sense out of change is to plunge into it, move with it, and join the dance—Alan Watts*

In my pursuit to grow as a developer I spend a lot of time reverse-engineering web technologies. I'm basically a reverse-engineering addict 😆 . The topic that fascinates me the most is change detection. You can find this mechanism in almost any web application. It's an integral part of the most popular web frameworks. Sufficiently advanced widgets, like datagrids or stateful jQuery plugins, have change detection. And there's a good chance that change detection lurks somewhere in your application's code base.

Every aspiring software architect must have a good understanding of this mechanism. I'd argue that change detection is the most important piece of an architecture since it's responsible for the visible part like DOM updates. It's also the area that significantly affects an application's performance. This article will greatly expand your knowledge in this domain.

We'll start by looking at the change detection in general. Then we'll implement a very basic change detection mechanism ourselves. And once we've established the essence of change detection, we'll take an in-depth look at how it's implemented in Angular and React.

I hope that this knowledge will awaken your curiosity. I want to inspire you to learn more about the web platform, architecture and programming. I want you to become an extraordinary engineer.

> I work as a developer advocate at **ag-Grid**. If you're curious to learn about data grids or looking for the ultimate Angular/React/Vue data grid solution, give it a try with our guide "**Get started in 5 minutes**" guide. I'm happy to answer any questions you may have. **And follow me to stay tuned**!

What is change detection?

So, let's start with the definition:

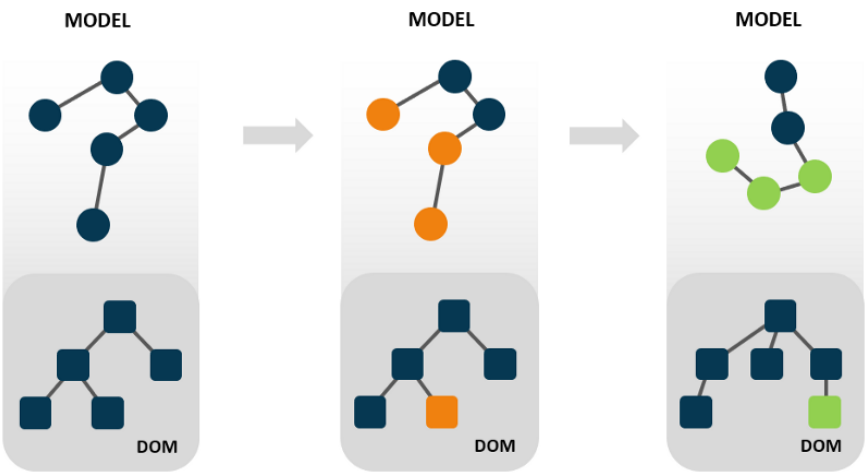
> Change detection is the mechanism designed to track changes in an application state and render the updated state on the screen. It ensures that the user interface always stays in sync with the internal state of the program.

We can infer from this definition that change detection has two main parts: tracking changes and rendering.

Let's take a look at rendering first. In any application, the process of rendering takes the internal state of the program and projects it into something we can see on the screen. In web development, we take data structures like objects and arrays and end up with a DOM representation of that data in the form of images, buttons and other visual elements. Although the implementation of rendering logic may not always be trivial, it's still pretty straightforward.

Things start to get a lot more sophisticated when we mix in data that changes over time. Web applications today are interactive. Which means that the application state can change any time as a result of a user interaction. There can be other reasons as well. Something else happens in the world that updates the server data. Then our client fetches the update.

As our state changes, we need to detect that and reflect the change.



This all might be a bit abstract, let's see a concrete example.

Rating widget

Suppose we want to implement a rating widget. The number of the solid stars on the screen reflects the current rating. This interactive widget allows a user to click on any star and set the new rating:

To track the rating, we need to store the current value somewhere. Let's define the private `_rating` property that will be part of our widget state:

```
export class RatingsComponent {
    constructor() {
        this._rating = 1;
    }
```

As we update the widget state we need to reflect these changes on the screen. Here's the DOM structure that we'll use to render the widget's UI:

```
<ul class="ratings">
    <li class="star solid"></li>
    <li class="star solid"></li>
    <li class="star solid"></li>
    <li class="star outline"></li>
    <li class="star outline"></li>
```

I'm using CSS classes `solid` and `outline` to render a corresponding star icon. The widget is initialized with all list items as star outlines. As the state changes the corresponding list items change to solid stars.

Initialization

First, we need to create all the required DOM nodes. This is our initialization logic that we'll put into the `init` method:

```
export class RatingsComponent {
    ...
    init(container) {
        this.list = document.createElement('ul');
        this.list.classList.add('ratings');
        this.list.addEventListener('click', (event) =>
            this.rating = event.target.dataset.value;
        });

        this.elements = [1, 2, 3, 4, 5].map((value) =>
            const li = document.createElement('li');
            li.classList.add('star', 'outline');
            li.dataset.value = value;
            this.list.appendChild(li);
```

In the code above we're creating an unordered list with items. Then we're adding CSS classes to the list items and registering an event listener for the `click` event.

Change detection

We need to get notified whenever the value of the `rating` property changes. In our basic implementation of change detection we'll use the setter functionality provided by JavaScript. So we define a setter for the `rating` property and trigger updates when its value changes. The DOM update is performed by swapping classes on list items. Here is the code:

```
export class RatingsComponent {
    ...
    set rating(v) {
        this._rating = v;

        // triggers DOM update
        this.updateRatings();
    }

    get rating() {
        return this._rating;
    }

    updateRatings() {
```

You can play with the implementation here. (The example uses the CSS classes `far` and `far` defined by font-awesome instead of the classes `solid` and `outline` used in the code above).

Now, look at the amount of code we needed to write to implement an extremely simple widget. Imagine a lot more elaborate functionality, possibly with multiple lists and conditional logic to show or hide some visual elements. The amount of code and complexity will grow dramatically. Ideally, in our everyday development, we want to focus on the application logic. We let someone else deal with the state tracking and screen updates. And this is where frameworks come in handy.

Frameworks

Frameworks take care of synchronization between the internal state of the application and the user interfaces for us. But not only do they take some weight off our shoulders, they do the state tracking and DOM updates very efficiently.

Here's how we can implement the same widget in Angular and in React. From a user's perspective with regards to UI, a template is the most important piece of a component configuration. It's interesting that we define templates in these frameworks in a similar manner.

Angular

```
<ul class="rating" (click)="handleClick($event)">
    <li [className]="'star ' + (rating > 0 ? 'solid' :
    <li [className]="'star ' + (rating > 1 ? 'solid' :
    <li [className]="'star ' + (rating > 2 ? 'solid' :
    <li [className]="'star ' + (rating > 3 ? 'solid' :
```

React

```
<ul className="rating" onClick={handleClick}>
    <li className={'star ' + (rating > 0 ? 'solid' : 'o
    <li className={'star ' + (rating > 1 ? 'solid' : 'o
    <li className={'star ' + (rating > 2 ? 'solid' : 'o
    <li className={'star ' + (rating > 3 ? 'solid' : 'o
```

The syntax is a bit different. The idea of using expressions as values for DOM element properties is the same. In the templates above, we're saying that the DOM property `className` depends on the component's property `rating`. So whenever the rating changes, the expression should be re-evaluated. If a change is detected, the `className` property should be updated.

> A quick note about the `click` event listener. Event listeners are not part of change detection in React or in Angular. They usually trigger change detection, but are never part of the process itself.

Change detection implementation

Although the idea of using expressions as values for DOM element properties is the same in both Angular and React, the underlying mechanisms are completely different. Let's now take a look under the hood of the change detection implementations in both of these frameworks.

Angular

When the compiler analyzes the template, it identifies properties of a component that are associated with DOM elements. For each such association, the compiler creates a binding in the form of instructions. A binding is the core part of change detection in Angular. It defines an association between a component's property (usually wrapped in some expression) and the DOM element property.

Once bindings are created, Angular no longer works with the template. The change detection mechanism executes instructions that process bindings. The job of these instructions is to check if the value of an expression with a component property has changed and perform DOM updates if necessary.

In our case, the `rating` property in the template is bound to the `className` property through the expression:

```
[className]="'star ' + ((ctx.rating > 0) ? 'solid' :
'outline')"
```

For this part of the template, the compiler generates instructions that set up a binding, perform dirty checks and update the DOM. Here's the code that is generated for our template:

```
if (initialization) {
    elementStart(0, 'ul');
        ...
        elementStart(1, 'li', ...);

        // sets up the binding to the className proper
        elementStyling();
        elementEnd();
        ...
    elementEnd();
}

if (changeDetection) {
```

> A quick note about the instructions you see on the screen: they are the output of the new compiler called Ivy. Previous versions of Angular use exactly the same idea of bindings and dirty checking, but they implement it a bit differently.

Let's suppose that Angular created the binding for the `className` and the current values of the binding are as follows:

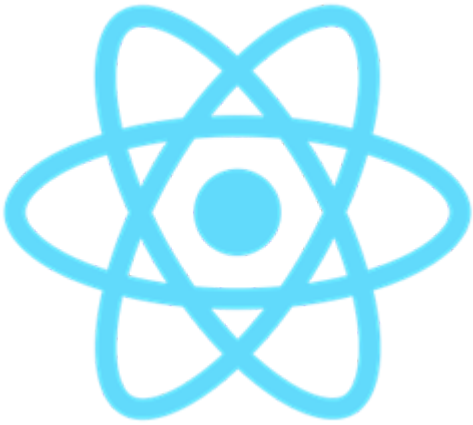
```
{ dirty: false, value: 'outline' }
```

Once the rating is updated, Angular runs change detection and processes the instructions. The first function takes the result of the evaluated expression and uses it to compare to the previous value remembered by the binding. This is where the name "dirty checking" comes from. If the value has changed, it updates the current value and marks this binding as dirty.

```
{ dirty: true, value: 'solid' }
```

The second instruction checks if the binding is dirty and if so, uses the new value to update the DOM. In our case it will update the `className` property of the list item.

Processing bindings that perform dirty checks and update the relevant parts of the DOM are the core operations of change detection in Angular.



React

As it turns out, React uses a completely different approach. To be honest, I got so used to bindings in Angular that it took me quite a while to crack the exact algorithm in React. As you can guess, React doesn't use bindings. **The core part of the change detection mechanism in React is Virtual DOM comparisons.** So how does it work?

> The explanation below provides a high-level overview in comparison to Angular. I was lucky to have Andrew Clark explain me the fundamentals behind the new Fiber implementation. I'm working on the new series of in-depth articles that explore the implementation details of Fiber change detection algorithm in React. **Stay tuned and follow me on Twitter and on Medium, I'll tweet as soon as it's ready.**

All React components implement the `render` method that returns a JSX template:

```
export class RatingComponent extends ReactComponent {
    ...
    render() {
        return (
            <ul className="rating" onClick={handleClic
                <li className={'star ' + (rating > 0 ?
                ...
            </ul>
```

In React, a template is compiled into a bunch of `React.createElement` function calls. In the code below I'm using the `el` variable as an alias for the the function:

```
const el = React.createElement;

export class RatingComponent extends ReactComponent {
    ...
    render() {
        return el('ul', { className: 'ratings', onclic
                 el('li', { className: 'star ' + (rati
                    ...
```

Each call of the `React.createElement` function creates a data structure known as a Virtual DOM node. Nothing fancy, it's just a plain JavaScript object that describes an HTML element, its properties and children. And when you have multiple calls to the function, collectively, they create a Virtual DOM tree. So, eventually the `render` method returns a Virtual DOM tree:

```
export class RatingComponent extends ReactComponent {
    ...
    render() {
        return {
            tagName: 'UL',
            properties: {className: 'ratings'},
            children: [
                {tagName: 'LI', properties: {className
                ...
```

Expressions with a component property are evaluated at the time of the `render` function call. The Virtual DOM node properties contain

the results of the evaluated expressions. Let's imagine that in our case the value of the rating property is `0`. This means that the following expression:

```
{ className: rating > 0 ? 'solid' : 'outline' }
```

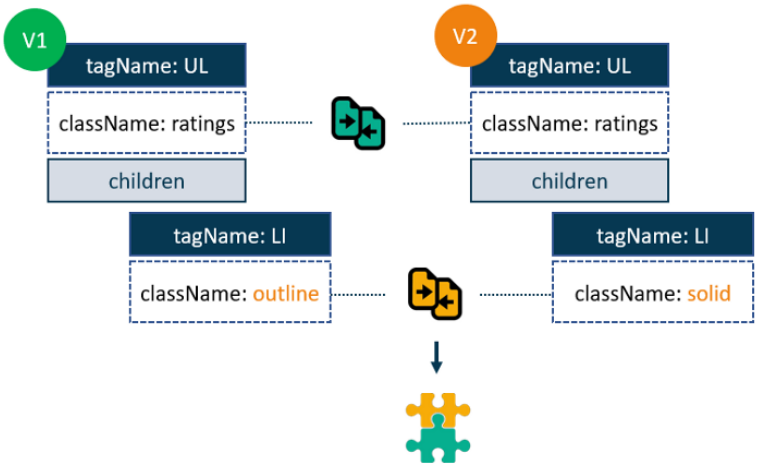
evaluates to the value `outline` and that's the value used for the `className` property in the Virtual DOM. Based on this Virtual DOM tree React will create the list item element with the CSS class `outline`.

Now suppose the `rating` has been updated to `1` and the expression

```
{ className: rating > 0 ? 'solid' : 'outline' }
```

has now produced the result `solid`. React runs change detection, executes the `render` function that returns a new version of Virtual DOM tree. The value of the `className` property in this Virtual DOM version is `solid`. **It's very important to understand that the `render` function is called during each change detection cycle.** This means that every time the function is called, it can return a completely different Virtual DOM tree.

So React now has two Virtual DOM data structures at hand:



It then runs a diffing algorithm on the two Virtual DOMs to get the set of changes between them. In our case this is going to be the difference in the `className` property on the list item. **Once the difference is found, the algorithm produces a patch to update the corresponding DOM nodes.** In our case the patch will update the `className` property with the value `solid` from the new Virtual DOM. And then this updated version of Virtual DOM will be used for the comparisons during the next change detection cycle.

Obtaining a new Virtual DOM tree from a component, comparing it to the previous version of the tree, generating a patch to update the relevant parts of the DOM and performing updates are the **core operations of change detection in React.**

When does change detection run?

This is the one question that we haven't looked at. To have a comprehensive understanding of change detection, we need to know when React calls the `render` function or Angular executes instructions that process bindings. I think that the mechanism that initiates change detection must be reviewed separately from the mechanism that detects changes and performs rendering. Let's take a look at it now.

If you think about it, there can be two ways to initiate change detection.
The first one is **to explicitly tell the framework** that something has changed or there's a possibility of a change so it should run change detection. Basically, in this way we initiate change detection manually. The second way is **to rely on the framework** to know when there's a possibility of a change and run change detection automatically. And this is where the frameworks also differ.

React

In React we always initialize the process of change detection manually. And we do it by calling the `setState` function:

```
1  export class RatingComponent extends React.Component {
2      ...
3      handleClick(event) {
4          this.setState({rating: Number(event.target.data
5      };
```

There's no way to trigger change detection automatically in React. Every change detection cycle starts with the call to the `setState` function.

Angular

But in Angular we have both options. We can use the Change Detector service to run change detection manually:

```
class RatingWidget {
    constructor(changeDetector) {
        this.cd = changeDetector;
    }

    handleClick(event) {
        this.rating = Number(event.target.dataset.valu
```

However, we can also rely on the framework to trigger change detection automatically. In this way, we simply update the property on a component:

```
class RatingWidget {
    handleClick(event) {
        this.rating = Number(event.target.dataset.value
    };
```

But how does Angular know when it should run change detection?

Well, since we bind to UI events in a template using mechanisms provided by Angular it knows about all UI event listeners. This means that it can intercept an event listener and schedule a change detection run after the application code has finished executing. This is a clever idea, but this mechanism can't be used to intercept all asynchronous events.

Since we don't use an Angular mechanism to bind to timing events like `setTimeout` or network events like `XHR`, change detection can't be triggered automatically. To solve this Angular uses a library called zone.js. It patches all asynchronous events in a browser and can then notify Angular when a certain event occurs. Similarly to UI events, Angular can then wait until the application code has finished executing and initiate change detection automatically.

We've learned a lot. Where do you go from here?

Change detection in web frameworks

The repository on GitHub that demonstrates under-the-hood details of change detection mechanism in web frameworks. Go through the code, see concrete details and use the knowledge to continue exploring the internals on your own.

Level Up Your Reverse Engineering Skills

A write up on reverse-engineering that summarizes my experience and outlines some guidelines and principles that will help you start with your exploration journey.

Practical application of reverse-engineering guidelines and principles

Insights into the thought process while reverse-engineering React. It shows a practical application of these principles through the actual process of reverse-engineering a small part of React. It also demonstrates a few interesting debugging techniques to accelerate your reverse-engineering efforts.

These 5 articles will make you an Angular Change Detection expert

This series is a must-read if you want to have a solid grasp of the change detection mechanism in Angular. Each article builds upon the information explained in the preceding one and goes from high-level overview down to implementation details with references to the sources.

In-depth overview of the new reconciliation algorithm in React

This series will teach you internal architecture of React. This article provides an in-depth overview of important concepts and data structures relevant to the algorithm. It gradually builds up background necessary to understand the general algorithm and main operations.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story.

React Grid — the fastest and most feature-rich grid component from ag-Grid