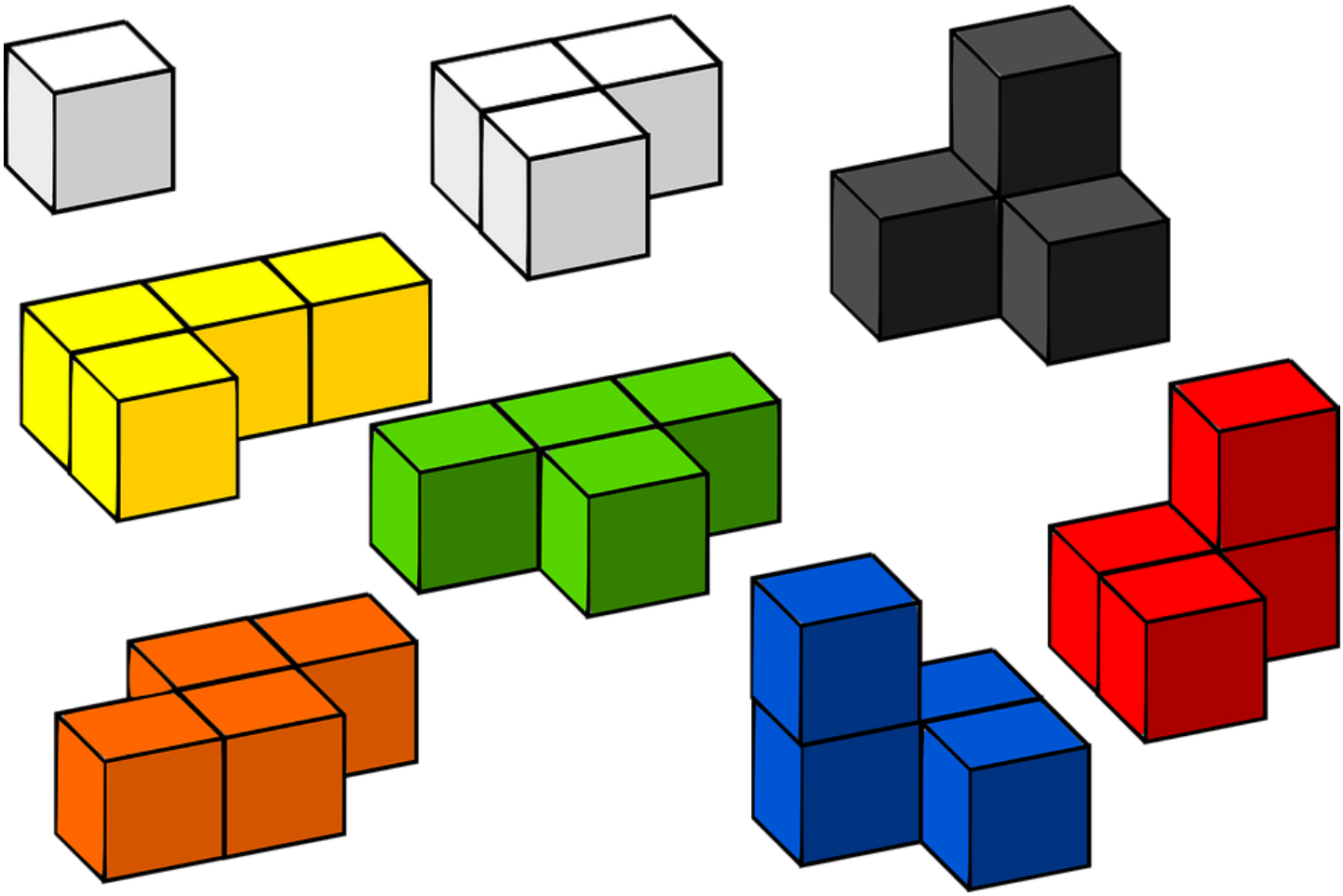
Meshileya Emmanuel Seun
Software Engineer
Oct 05 • 10 min read

Simplified Angular unit testing

In this article, we will learn how to write (simple) unit tests for your Angular modeling blocks (service, component, async task, etc).



Introduction

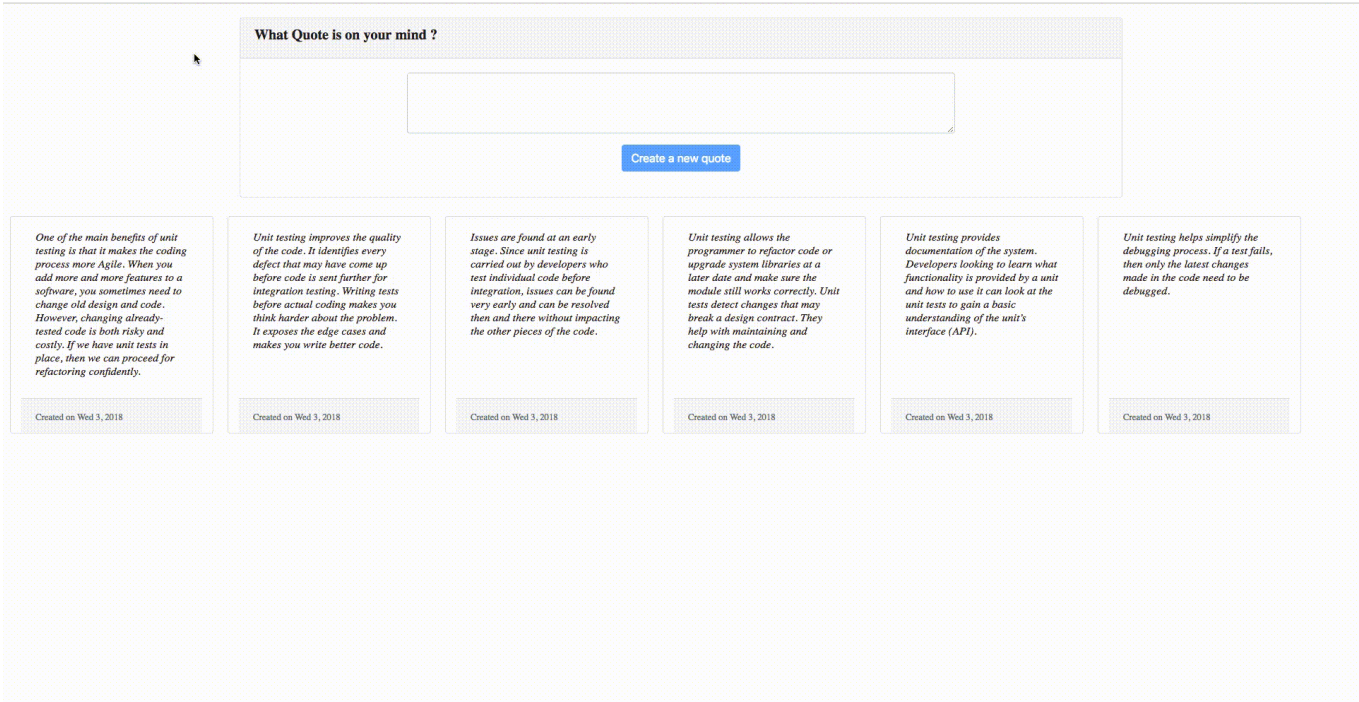
In this article, we will learn how to write (simple) unit tests for your Angular modeling blocks (service, component, async task, etc). We will be using a simple `Quotes` application to demonstrate how you can write a unit test for your project. I assume you already have an understanding of how to

use Angular 2+. If you are a beginner or you have interest in Angular, you can find more here.

Why do you need to test your application?

Have you been looking for a way to test what you have built based on user behavior? I really don't expect you to test each behavior one at a time, as this method does not only waste your time but is also ineffective. Writing tests for different coupling blocks in your application will help demonstrate how these blocks behave. The `Quotes` application we will be looking at has a service, a component and an async task to simulate data being fetched from the server.

One of the easiest ways to test the strengths of these blocks is writing a test for each of them. You don't necessarily need to wait until your users complain how the input field behaves when the button is clicked. Writing a test for your blocks (components, services etc) can easily detect when there is a break.



A simple Quote application

How do you set up an Angular test ?

When you create a new project with the cli (`ng new appName`), a default component and test file are added. Also, for those that always like a shortcut method like me, a test script is always created alongside any component module (service, component) you create using angular `cli` (Command Line Interface).

This test script which ends with `.spec.ts` is always added. Let's take a look at the initial test script file which is the `app.component.spec.ts` :

```
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';
describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));
  it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  }));
  it(`should have as title 'angular-unit-test'`, async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('angular-unit-test');
  }));
  it('should render title in a h1 tag', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;

expect(compiled.querySelector('h1').textContent).toContain('Welcome
 to angular-unit-test!');
  }));
});
```

Let's run our first test to make sure nothing has broken yet:

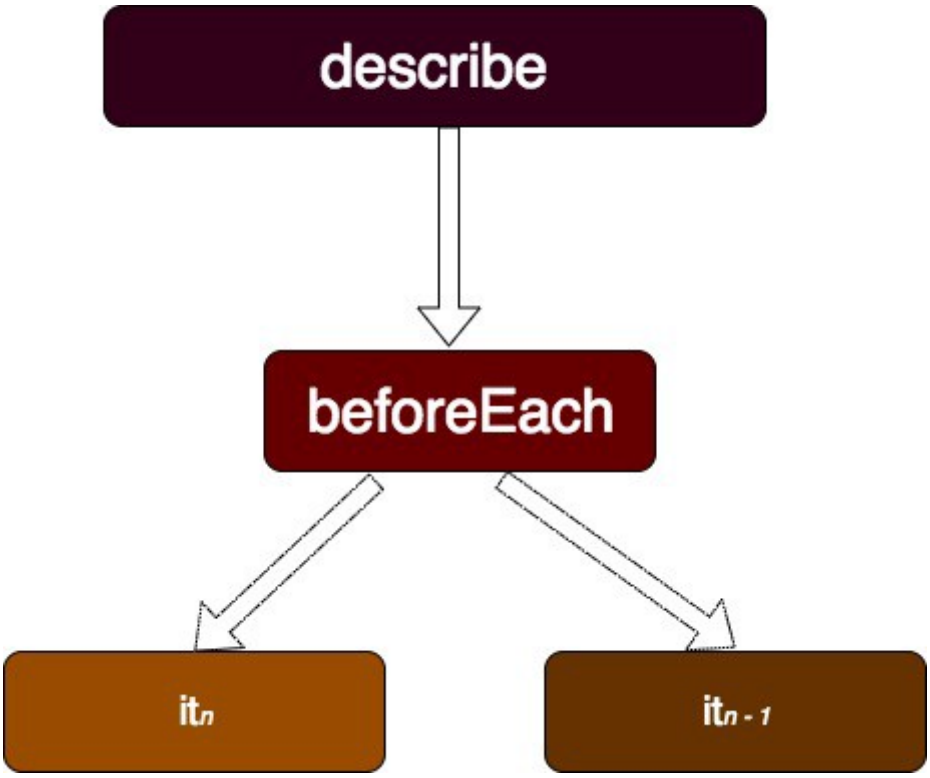
```
ng test
```

You might be wondering, how can we simulate a user behavior by simply writing a test, even though the project is being rendered in a browser? As we proceed, I will demonstrate how we can simulate the app running on a browser.

By default, Angular runs on `Karma` which is a test runner that runs the unit tests snippet like the above `app.component.spec.ts` file. Karma also ensures the result of the test is printed out either in the console or in file log. Other test runners are `mocha`, `jasmine` etc.

How does the test run?

The testing package has some utilities (`TestBed`, `async`). `TestBed` is the main Angular utility package.



flow of running test

The `describe` container contains different blocks (`it`, `beforeEach`,

`xit` etc). The `beforeEach` runs before any other block while others do not depend on each other to run.

From the `app.component.spec.ts` file, the first block is the `beforeEach` inside the container (`describe`). This is the only block that runs before any other block (`it`). The declaration of the app module in `app.module.ts` file is simulated (declared) in the `beforeEach` block. The component (`AppComponent`) declared in the `beforeEach` block is the main component we want to have in this testing environment. The same logic applies to other test declaration.

The `compileComponents` object is called to compile your component's resources like the template, styles etc. You might not necessarily compile your component if you are using webpack:

```
beforeEach(async(() => {
   TestBed.configureTestingModule({
      declarations: [
         AppComponent
      ],
   }).compileComponents();
}));
```

Now that the component has been declared in the `beforeEach` block, let's check if the component is created.

The `fixture.debugElement.componentInstance` creates an instance of the class (`AppComponent`) . We will test to see if the instance of the class is truly created or not using `toBeTruthy` :

```
it('should create the app', async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
}));
```

The third block demonstrates how you can have access to the properties of the created component(`AppComponent`). The only property added by default is the title. You can easily check if the title you set has changed or not from the instance of the component (`AppComponent`) created:

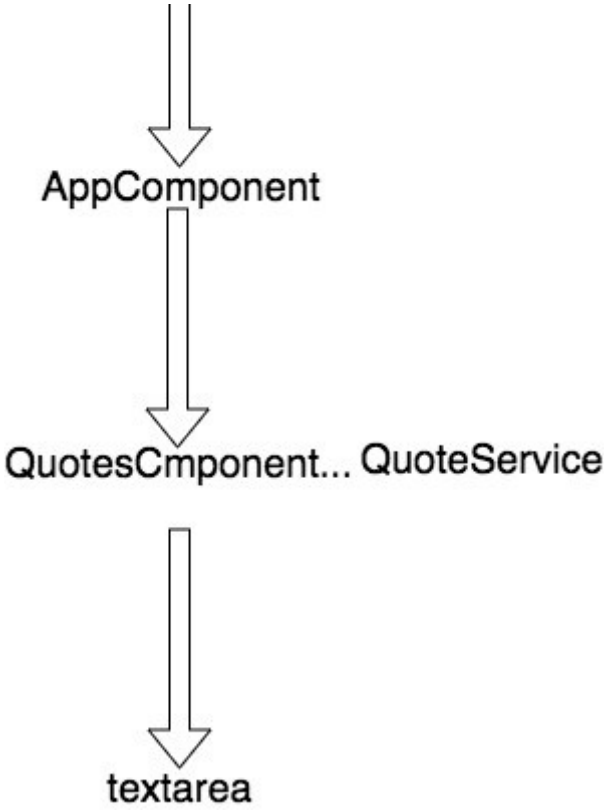
```
it(`should have as title 'angular-unit-test'`, async(() => {
     const fixture = TestBed.createComponent(AppComponent);
     const app = fixture.debugElement.componentInstance;
     expect(app.title).toEqual('angular-unit-test');
}));
```

The fourth block demonstrates how the test behaves in the browser environment. After creating the component, an instance of the created component (`detectChanges`) to simulate running on the browser environment is called. Now that the component has been rendered, you can have access to its child element by accessing the `nativeElelment` object of the rendered component (`fixture.debugElement.nativeElement`):

```
it('should render title in a h1 tag', async(() => {
   const fixture = TestBed.createComponent(AppComponent);
   fixture.detectChanges();
   const compiled = fixture.debugElement.nativeElement;

expect(compiled.querySelector('h1').textContent).toContain('Welcome
 to angular-unit-test!');
}));
```

Now that you have familiarized yourself with the basics of testing a component. Let's test the `Quote` application.



Overview of Quote application

How to test a service(QuoteService)

Services often depend on other services that Angular injects into the constructor. In many cases, it easy to create and *inject* these dependencies by adding `providedIn: root` to the injectable object which makes it accessible by any component or service:

```
import { Injectable } from "@angular/core";
import { QuoteModel } from "../model/QuoteModel";

@Injectable({
  providedIn: "root"
})
export class QuoteService {
  public quoteList: QuoteModel[] = [];

  private daysOfTheWeeks = ["Sun", "Mon", "Tue", "Wed", "Thurs",
"Fri", "Sat"];


  constructor() {}

  addNewQuote(quote: String) {
    const date = new Date();
```

```
    const dayOfTheWeek = this.daysOfTheWeeks[date.getDate()];
    const day = date.getDay();
    const year = date.getFullYear();
    this.quoteList.push(
      new QuoteModel(quote, `${dayOfTheWeek} ${day}, ${year}`)
    );
  }

  getQuote() {
    return this.quoteList;
  }

  removeQuote(index) {
    this.quoteList.splice(index, 1);
  }
}
```

Here are a few ways to test the `QuoteService` class:

```
/* tslint:disable:no-unused-variable */
import { QuoteService } from "./Quote.service";

describe("QuoteService", () => {
  let service: QuoteService;

  beforeEach(() => {
    service = new QuoteService();
  });

  it("should create a post in an array", () => {
    const qouteText = "This is my first post";
    service.addNewQuote(qouteText);
    expect(service.quoteList.length).toBeGreaterThanOrEqual(1);
  });

  it("should remove a created post from the array of posts", ()
=> {
    service.addNewQuote("This is my first post");
    service.removeQuote(0);
    expect(service.quoteList.length).toBeLessThan(1);
  });
});
```

In the first block, `beforeEach`, an instance of `QuoteService` is created to ensure its only created once and to avoid repetition in other blocks except for some exceptional cases:

```
it("should create a post in an array", () => {
    const qouteText = "This is my first post";
    service.addNewQuote(qouteText);
    expect(service.quoteList.length).toBeGreaterThanOrEqual(1);
  });
```

The first block tests if the post model `QuoteModel(text, date)` is created into an array by checking the length of the array. The length of the `quoteList` is expected to be 1:

```
it("should remove a created post from the array of posts", () =>
{
    service.addNewQuote("This is my first post");
    service.removeQuote(0);
    expect(service.quoteList.length).toBeLessThan(1);
  });
```

The second block creates a post in an array and removes it immediately by calling `removeQuote` in the service object. The length of the `quoteList` is expected to be 0.

How to test a component (QuotesComponent)

The `service` is injected into the `QuoteComponent` in order to have access to its properties which will be needed by the view:

```
import { Component, OnInit } from '@angular/core';
import { QuoteService } from '../service/Quote.service';
```

```
import { QuoteModel } from '../model/QuoteModel';

@Component({
  selector: 'app-Quotes',
  templateUrl: './Quotes.component.html',
  styleUrls: ['./Quotes.component.css']
})
export class QuotesComponent implements OnInit {

  public quoteList: QuoteModel[];
  public quoteText: String = null;

  constructor(private service: QuoteService) { }

  ngOnInit() {
    this.quoteList = this.service.getQuote();
  }

  createNewQuote() {
    this.service.addNewQuote(this.quoteText);
    this.quoteText = null;
  }

  removeQuote(index) {
    this.service.removeQuote(index);
  }
}
```

```
<div class="container-fluid">
  <div class="row">
    <div class="col-8 col-sm-8 mb-3 offset-2">
      <div class="card">
        <div class="card-header">
          <h5>What Quote is on your mind ?</h5>
        </div>
        <div class="card-body">
          <div role="form">
            <div class="form-group col-8 offset-2">
              <textarea #quote class="form-control" rows="3"
cols="8" [(ngModel)]="quoteText" name="quoteText"></textarea>
            </div>
            <div class="form-group text-center">
              <button class="btn btn-primary"
(click)="createNewQuote()" [disabled]="quoteText == null">Create
```

```
a new
                quote</button>
            </div>
          </div>
        </div>
      </div>
    </div>

    <div class="row">
      <div class="card mb-3 col-5 list-card" id="quote-cards"
style="max-width: 18rem;" *ngFor="let quote of quoteList; let i =
index"
        (click)="removeQuote(i)">
        <div class="card-body">
          <h6>{{ quote.text }}</h6>
        </div>
        <div class="card-footer text-muted">
          <small>Created on {{ quote.timeCreated }}</small>
        </div>
      </div>
    </div>
</div>
```

The first two blocks in the `describe` container run consecutively. In the first block, the `FormsModule` is imported into the configure test. This ensures the forms related directives like `ngModel` can be used.

Also, the `QuotesComponent` is declared in the `configTestMod` similarly to how the components are declared in `ngModule` residing in the appModule file. The second block `creates` a `QuoteComponent` and its `instance` which would be used by the other blocks:

```
let component: QuotesComponent;
  let fixture: ComponentFixture<QuotesComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [FormsModule],
      declarations: [QuotesComponent]
    });
  });
```

```
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(QuotesComponent);
    component = fixture.debugElement.componentInstance;
  });
```

This block tests if the instance of the component that is created is defined:

```
  it("should create Quote component", () => {
      expect(component).toBeTruthy();
    });
```

The injected service handles the manipulation of all operations (add, remove, fetch). The `quoteService` variable holds the injected service (`QuoteService`). At this point, the component is yet to be rendered until the `detectChanges` method is called:

```
  it("should use the quoteList from the service", () => {
      const quoteService =
fixture.debugElement.injector.get(QuoteService);
      fixture.detectChanges();
      expect(quoteService.getQuote()).toEqual(component.quoteList);
    });
```

Now let's test if we can successfully create a post. The properties of the component can be accessed upon instantiation, so the component rendered detects the new changes when a value is passed into the `quoteText` model. The `nativeElement` object gives access to the HTML element rendered which makes it easier to check if the `quote` added is part of the texts rendered:

```
  it("should create a new post", () => {
      component.quoteText = "I love this test";
      fixture.detectChanges();
```

```
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.innerHTML).toContain("I love this test");
  });
```

Apart from having access to the HTML contents, you can also get an element by its CSS property. When the `quoteText` model is empty or null, the button is expected to be disabled:

```
it("should disable the button when textArea is empty", () => {
    fixture.detectChanges();
    const button = fixture.debugElement.query(By.css("button"));
    expect(button.nativeElement.disabled).toBeTruthy();
  });
```

```
it("should enable button when textArea is not empty", () => {
    component.quoteText = "I love this test";
    fixture.detectChanges();
    const button = fixture.debugElement.query(By.css("button"));
    expect(button.nativeElement.disabled).toBeFalsy();
  });
```

Just like the way we access an element with its CSS property, we can also access an element by its class name. Multiple classes can be accessed at the same time using `By e.g By.css('.className.className')` .

The button clicks are simulated by calling the `triggerEventHandler` . The `event` type must be specified which ,in this case, is click. A quote displayed is expected to be deleted from the `quoteList` when clicked on:

```
it("should remove post upon card click", () => {
    component.quoteText = "This is a fresh post";
    fixture.detectChanges();
```

```
    fixture.debugElement
      .query(By.css(".row"))
      .query(By.css(".card"))
      .triggerEventHandler("click", null);
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.innerHTML).toContain("This is a fresh post");
  });
```

How do you test an asynchronous operation?

You can't escape a time you will need to fetch data remotely. This operation is best treated as an asynchronous task.

`fetchQoutesFromServer` represents an async task which returns an array of quotes after two seconds:

```
fetchQuotesFromServer() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve([new QuoteModel("I love unit testing", "Mon 4,
2018")]);
      }, 2000);
    });
  }
```

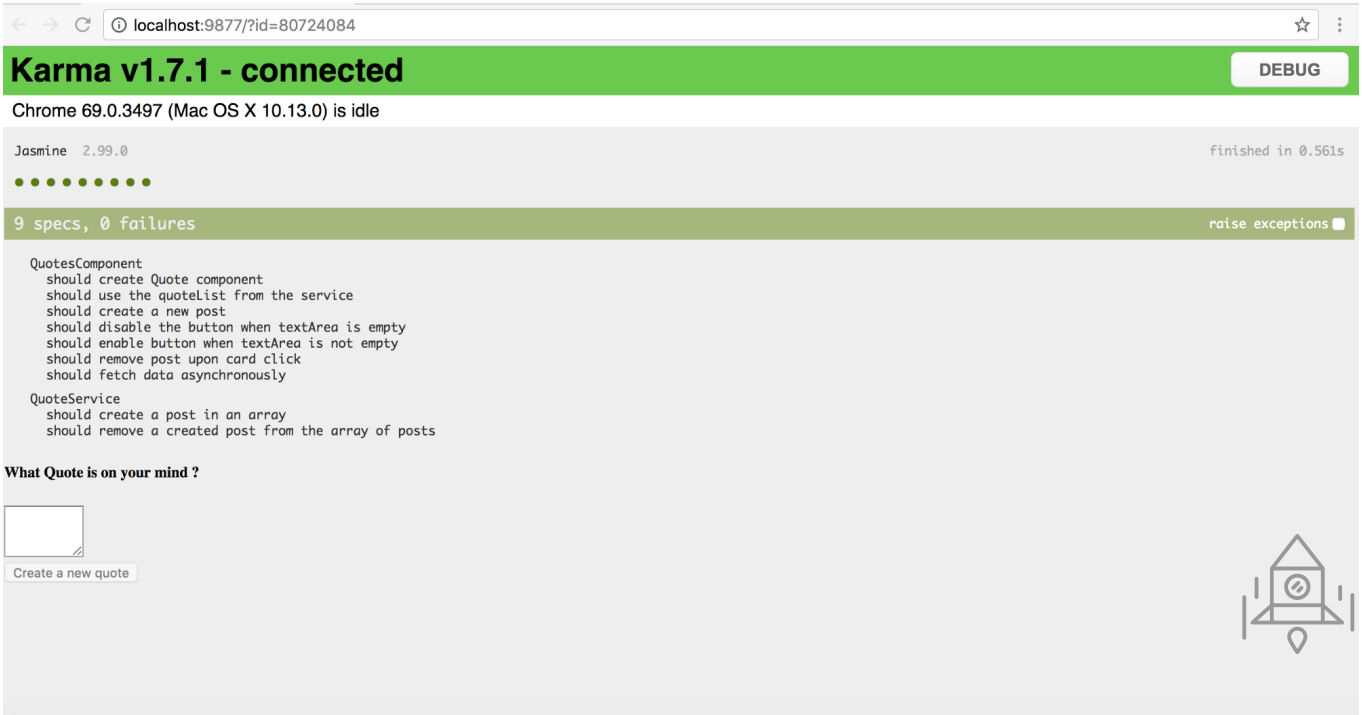
`spyOn` objects simulate how `fetchQuotesFromServer` method works. It accepts two argument `quoteService` which is injected into the component and the method `fetchQuotesFromServer` . `fetchQuotesFromServer` is expected to return a promise. `spyOn` chains the method using `and` with a fake promise call which is returned using `returnValue` . Since we want to fake how the `fetchQuotesFromServer` works, we need to pass a `promise` that will resolve with a list of quotes.

Just as we have done before, the `detectChanges` method is called to get the updated changes. `whenStable` allows access to results of all `async` tasks when they are done:

```
  it("should fetch data asynchronously", async () => {
      const fakedFetchedList = [
        new QuoteModel("I love unit testing", "Mon 4, 2018")
      ];
      const quoteService =
  fixture.debugElement.injector.get(QuoteService);
      let spy = spyOn(quoteService,
  "fetchQuotesFromServer").and.returnValue(
        Promise.resolve(fakedFetchedList)
      );
      fixture.detectChanges();
      fixture.whenStable().then(() => {
        expect(component.fetchedList).toBe(fakedFetchedList);
      });
    });
```

Conclusion

Angular also ensures test results are viewed in your browser. This will give a better visualization of the test results.

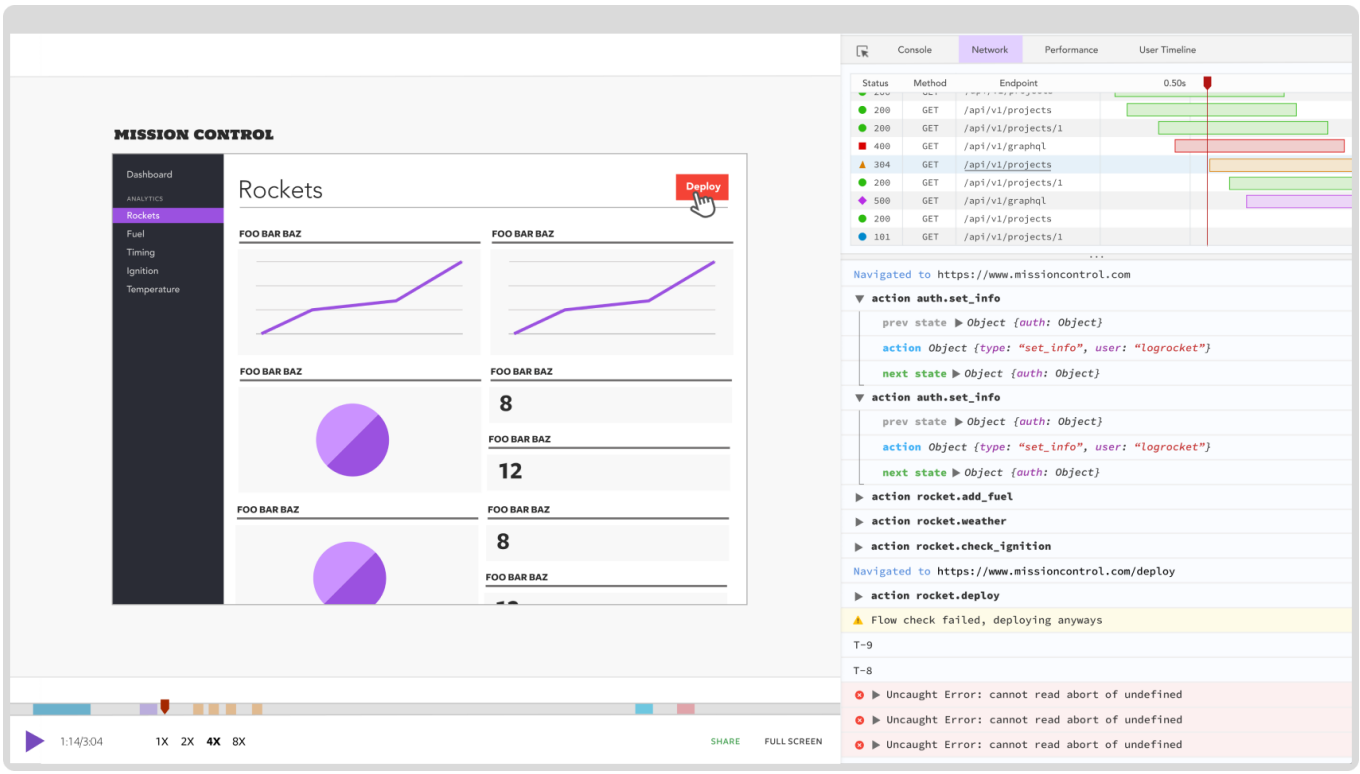


Test Result

The source code to the project can be found here.

. . .

Plug: LogRocket, a DVR for web apps



LogRocket is a frontend logging tool that lets you replay problems as if they happened in your own browser. Instead of guessing why errors happen, or asking users for screenshots and log dumps, LogRocket lets you replay the session to quickly understand what went wrong. It works perfectly with any app, regardless of framework, and has plugins to log additional context from Redux, Vuex, and @ngrx/store.

In addition to logging Redux actions and state, LogRocket records console logs, JavaScript errors, stacktraces, network requests/responses with headers + bodies, browser metadata, and custom logs. It also instruments the DOM to record the HTML and CSS on the page, recreating pixel-perfect videos of even the most complex single page apps.

Try it for free.