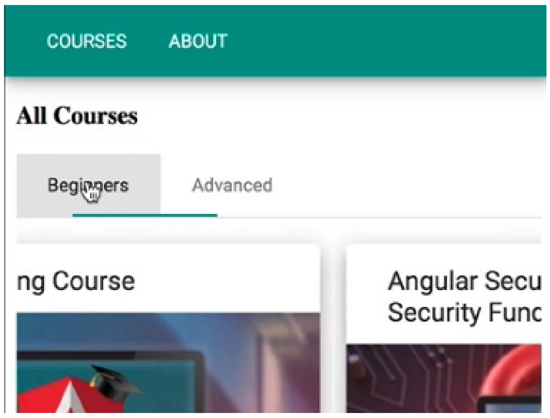
SECTION 04 - ASYNCHRONOUS TESTING IN DEPTH

Index

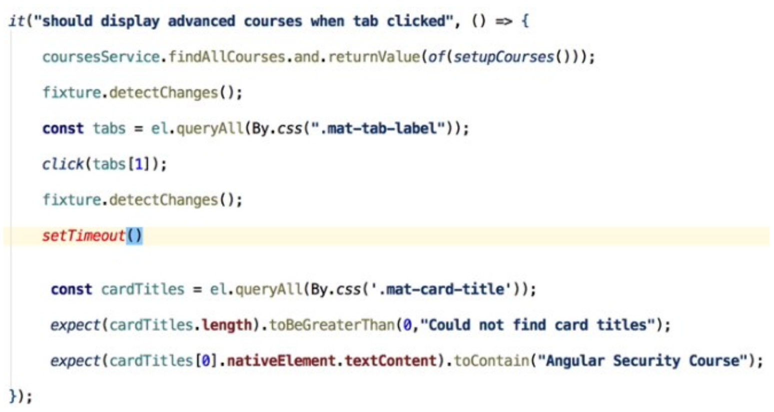
- Introduction to Asynchronous Testing - Understanding Jasmine done()

- Now we're gonna fix the *"should display advanced courses when tab clicked"* spec that we left in the previous section and it's still not working.

- The tab container component is performing an asynchronous operation while switching tabs. If you notice, there is a small animation whenever we switch tabs. This animation is produced by using the browser API *request animation frame*. This is one of the many browser asynchronous APIs.



- Another example of an asynchronous browser operation is *setTimeout()*. Imagine that the "Advanced" component was calling *setTimeout()* before switching tabs. That would make the component asynchronous as well.

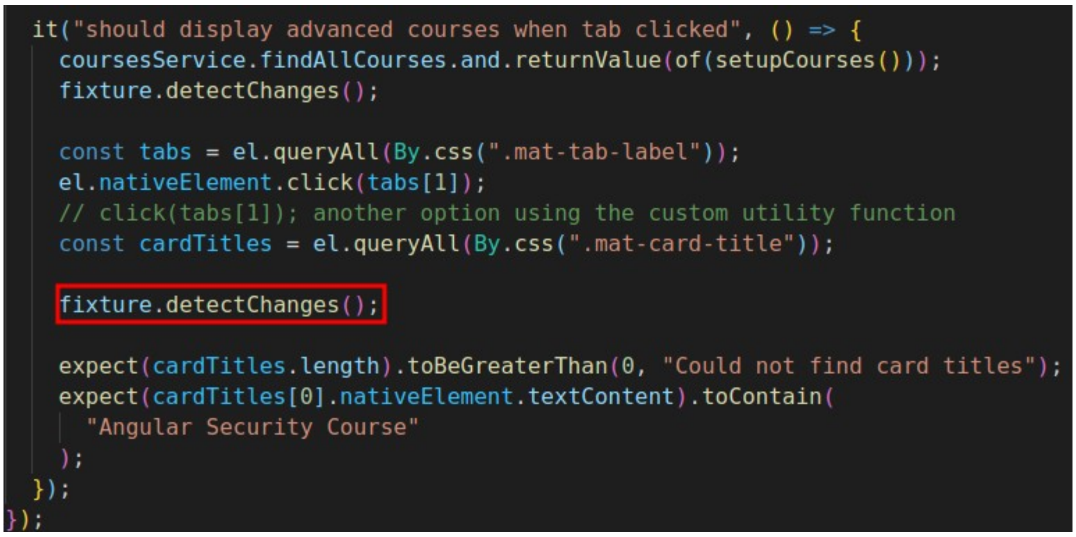


```
it("should display advanced courses when tab clicked", () => {

    coursesService.findAllCourses.and.returnValue(of(setupCourses()));

    fixture.detectChanges();

    const tabs = el.queryAll(By.css(".mat-tab-label"));

    click(tabs[1]);

    fixture.detectChanges();

    setTimeout()

     const cardTitles = el.queryAll(By.css('.mat-card-title'));

     expect(cardTitles.length).toBeGreaterThan(0,"Could not find card titles");

     expect(cardTitles[0].nativeElement.textContent).toContain("Angular Security Course");

});
```

- Another example of an asynchronous API would be to call *setInterval()* in order to execute a callback at regular intervals. Another example of a browser asynchronous API would be, for example, to do an HTTP request. We could do that, for example, using the browser built-In Fetch API. These API, which is a promise based API, is also asynchronous. The HTTP request might take several seconds to execute on the backend before retrieving the results back in the browser, which will trigger the execution of a callback.

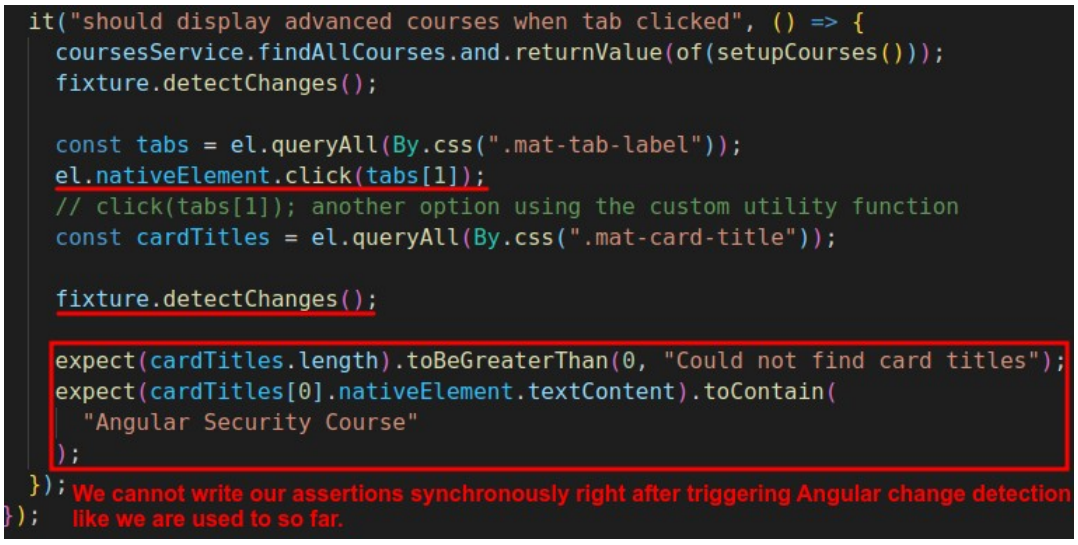
- Going back to the test, the problem is that after triggering the click, the tab container component that we are using to implement the HomeComponent, which is the <mat-tab-group>, is internally using a call to *request animation frame*.

- The problem is that the changes to the DOM are not going to be applied immediately after calling detectChanges().



```
it("should display advanced courses when tab clicked", () => {
  coursesService.findAllCourses.and.returnValue(of(setupCourses()));
  fixture.detectChanges();

  const tabs = el.queryAll(By.css(".mat-tab-label"));
  el.nativeElement.click(tabs[1]);
  // click(tabs[1]); another option using the custom utility function
  const cardTitles = el.queryAll(By.css(".mat-card-title"));

  fixture.detectChanges();

  expect(cardTitles.length).toBeGreaterThan(0, "Could not find card titles");
  expect(cardTitles[0].nativeElement.textContent).toContain(
    "Angular Security Course"
  );
 });
});
```

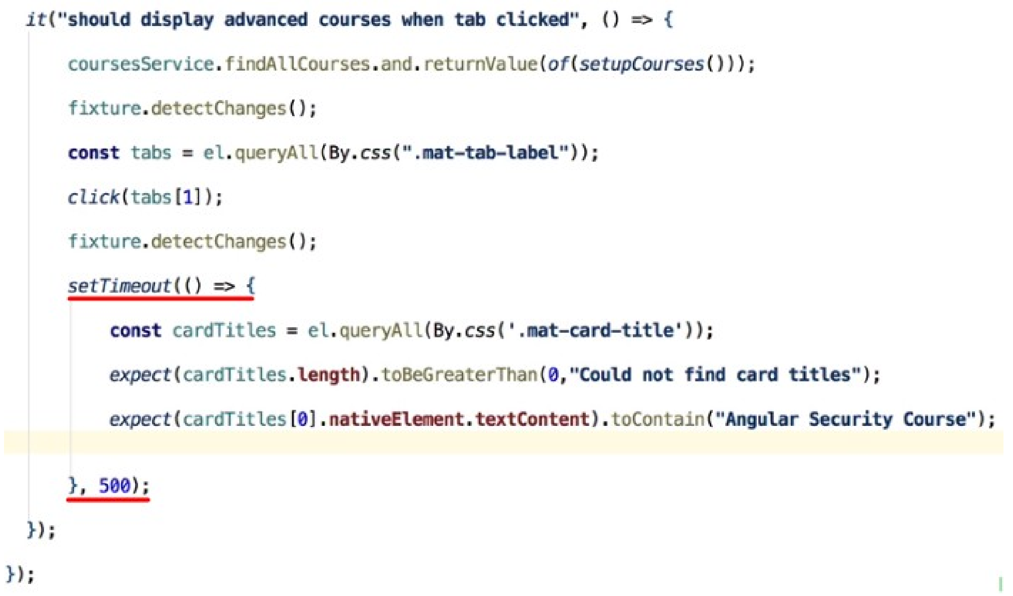
- Even though we have clicked the tab and even though we have asked Angular to apply the latest changes to the DOM (by calling fixture.detectChanges()), the component will not apply those changes immediately synchronously to the DOM. The container is going to first wait for the callback pass to request animation frame to complete before applying the changes to the DOM. So this means that we cannot write our assertions synchronously right after triggering angular change detection like we are used to so far.



```
it("should display advanced courses when tab clicked", () => {
  coursesService.findAllCourses.and.returnValue(of(setupCourses()));
  fixture.detectChanges();

  const tabs = el.queryAll(By.css(".mat-tab-label"));
  el.nativeElement.click(tabs[1]);
  // click(tabs[1]); another option using the custom utility function
  const cardTitles = el.queryAll(By.css(".mat-card-title"));

  fixture.detectChanges();

  expect(cardTitles.length).toBeGreaterThan(0, "Could not find card titles");
  expect(cardTitles[0].nativeElement.textContent).toContain(
    "Angular Security Course"
  );
 });
});
```

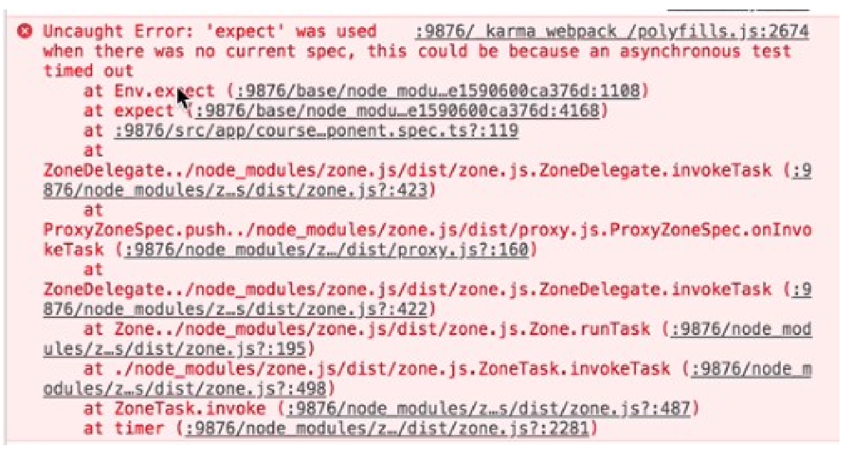
We cannot write our assertions synchronously right after triggering Angular change detection like we are used to so far.

- Because we know that the changes are going to be eventually applied to the DOM asynchronously, one **deceptive** way would be to simply call setTimeout().



```
    it("should display advanced courses when tab clicked", () => {

        coursesService.findAllCourses.and.returnValue(of(setupCourses()));

        fixture.detectChanges();

        const tabs = el.queryAll(By.css(".mat-tab-label"));

        click(tabs[1]);

        fixture.detectChanges();

        setTimeout(() => {

            const cardTitles = el.queryAll(By.css('.mat-card-title'));

            expect(cardTitles.length).toBeGreaterThan(0,"Could not find card titles");

            expect(cardTitles[0].nativeElement.textContent).toContain("Angular Security Course");

        }, 500);

    });

});
```

- The bad thing about this deceptive fix is that the test render exited the block and because it did not find any assertions during the execution, it considered the test as being passed.



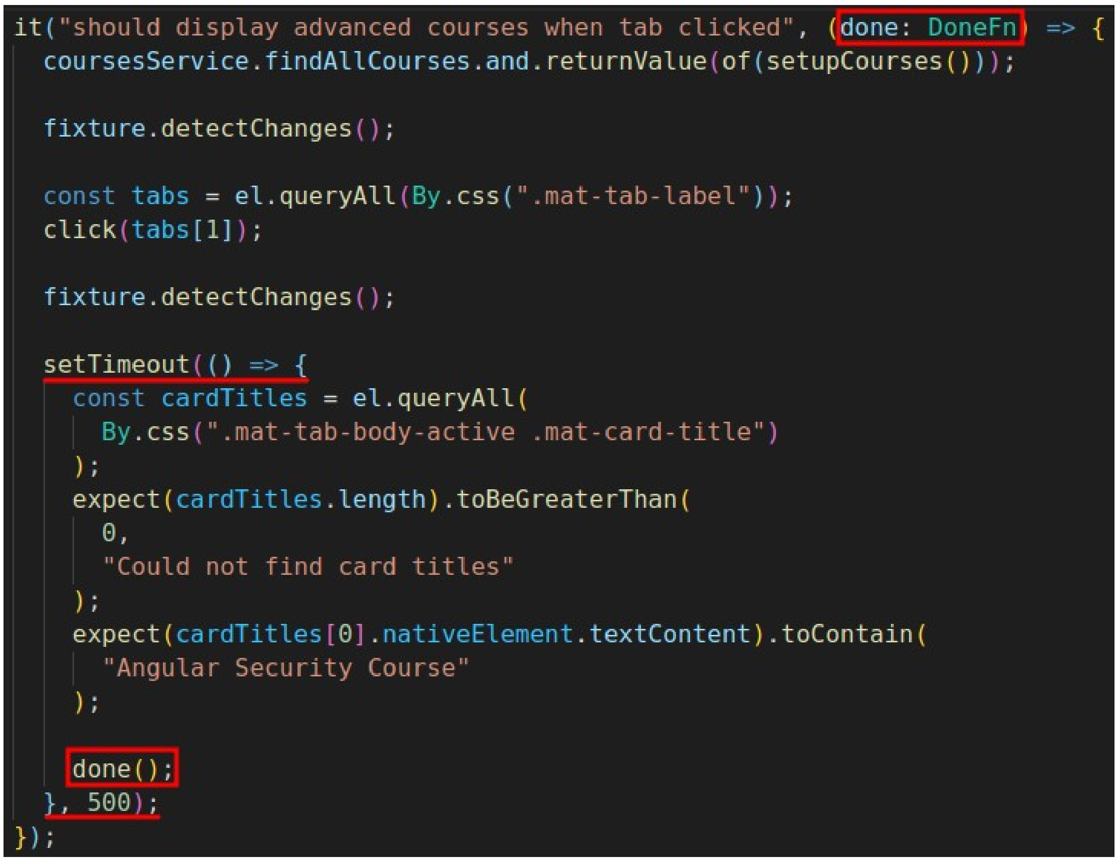
```
Uncaught Error: 'expect' was used    :9876/ karma webpack /polyfills.js:2674
when there was no current spec, this could be because an asynchronous test
timed out
    at Env.expect (:9876/base/node modu…e1590600ca376d:1108)
    at expect (:9876/base/node modu…e1590600ca376d:4168)
    at :9876/src/app/course…ponent.spec.ts?:119
    at
ZoneDelegate../node_modules/zone.js/dist/zone.js.ZoneDelegate.invokeTask (:9
876/node modules/z…s/dist/zone.js?:423)
    at
ProxyZoneSpec.push../node_modules/zone.js/dist/proxy.js.ProxyZoneSpec.onInvo
keTask (:9876/node modules/z…/dist/proxy.js?:160)
    at
ZoneDelegate../node_modules/zone.js/dist/zone.js.ZoneDelegate.invokeTask (:9
876/node modules/z…s/dist/zone.js?:422)
    at Zone../node_modules/zone.js/dist/zone.js.Zone.runTask (:9876/node mod
ules/z…s/dist/zone.js?:195)
    at ./node_modules/zone.js/dist/zone.js.ZoneTask.invokeTask (:9876/node m
odules/z…s/dist/zone.js?:498)
    at ZoneTask.invoke (:9876/node modules/z…s/dist/zone.js?:487)
    at timer (:9876/node modules/z…/dist/zone.js?:2281)
```

- We would like the test runner to somehow wait for the assertions to be called before considering the test completed and moving on to the other test.

- The real fix comes with Jasmine giving us support for asynchronous tests via a parameter called "***DoneFn***". This is a Jasmine specific callback and Jasmine knows that this is an synchronous test because of the use of the DoneFn callback.

- The ***DoneFn*** function is supposed to be called whenever our synchronous test is completed.

- Whenever Jasmine executes test, will not immediately consider the test finished when the block exits. Instead, Jasmine will only consider the test completed and move on to the other tests of the test suite only when "*done()*" callback is called.

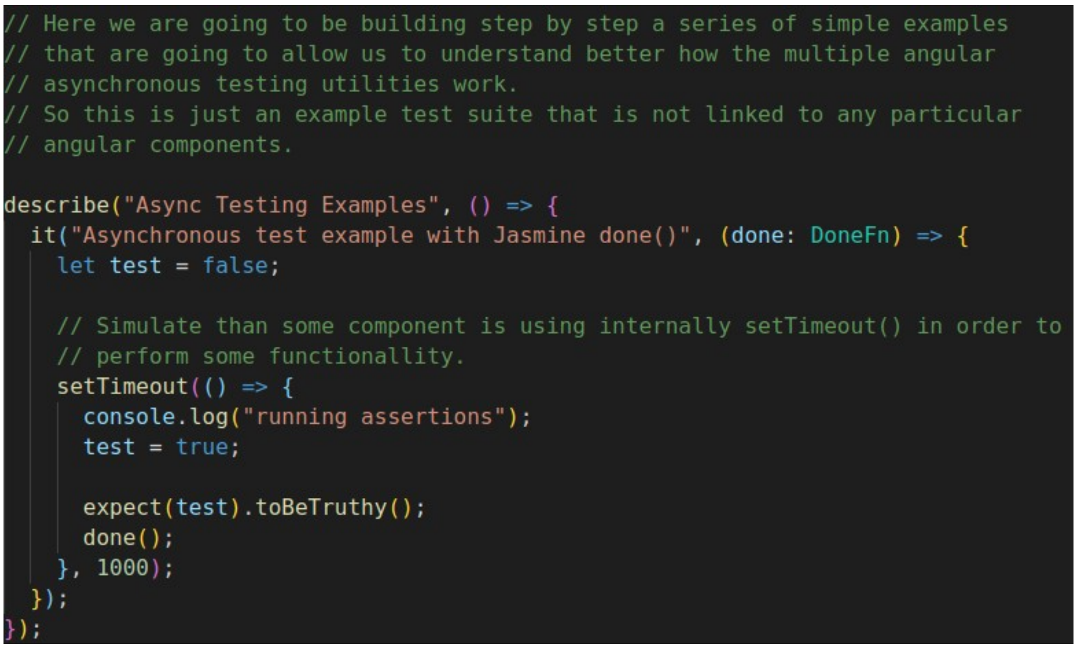


```
it("should display advanced courses when tab clicked", (done: DoneFn) => {
  coursesService.findAllCourses.and.returnValue(of(setupCourses()));

  fixture.detectChanges();

  const tabs = el.queryAll(By.css(".mat-tab-label"));
  click(tabs[1]);

  fixture.detectChanges();

  setTimeout(() => {
    const cardTitles = el.queryAll(
      By.css(".mat-tab-body-active .mat-card-title")
    );
    expect(cardTitles.length).toBeGreaterThan(
      0,
      "Could not find card titles"
    );
    expect(cardTitles[0].nativeElement.textContent).toContain(
      "Angular Security Course"
    );

    done();
  }, 500);
});
```

These are the keys in order to fix the test

- Understanding Asynchronous Testing - A Simple Example

- We are going to cover several of the Angular testing activities available for writing Angular asynchronous tests. So, tests that attempt to validate the functionality of both services and components that use browser asynchronous operations, such as, for example, *setTimeout*, *setInterval* and many other asynchronous browser APIs that are available.



```
// Here we are going to be building step by step a series of simple examples
// that are going to allow us to understand better how the multiple angular
// asynchronous testing utilities work.
// So this is just an example test suite that is not linked to any particular
// angular components.

describe("Async Testing Examples", () => {
  it("Asynchronous test example with Jasmine done()", (done: DoneFn) => {
    let test = false;

    // Simulate than some component is using internally setTimeout() in order to
    // perform some functionallity.
    setTimeout(() => {
      console.log("running assertions");
      test = true;

      expect(test).toBeTruthy();
      done();
    }, 1000);
  });
});
```

- The first conclusion that we can take from this very simple example is that even if our components, directives or services are using internally Browser asynchronous operations such as *setTimeout*, request, animation, frame, AJAX requests, *setInterval* and many other asynchronous browser APIs, we can still write our tests using the "*done*" Jasmine callback. However, this approach is still not very convenient, and it's to be avoided, especially in more complex components that are using multiple of these operations. Imagine a component that is doing several multiple timeouts, it's calling an interval, doing an Ajax call, etc. It would be hard to know for how long to wait. And most of all, our tests might become hard to read with nested set timeout blocks, etc.

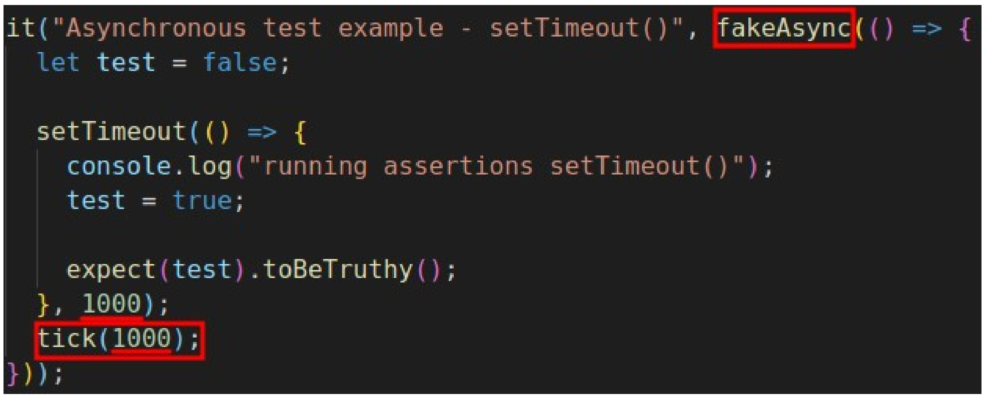
- Understanding the Angular fakeAsync Testing Zone

- We are now going to look into the first recommended alternative to the use Jasmine *done*, which is going to be the Angular Fake Async test utility. The Fake Async test utility allows us to write our tests in asynchronous looking way. It's even going to allow us to simulate the passing of time. So there are a lot of benefits for using that approach instead of the Jasmine "*done*" callback.

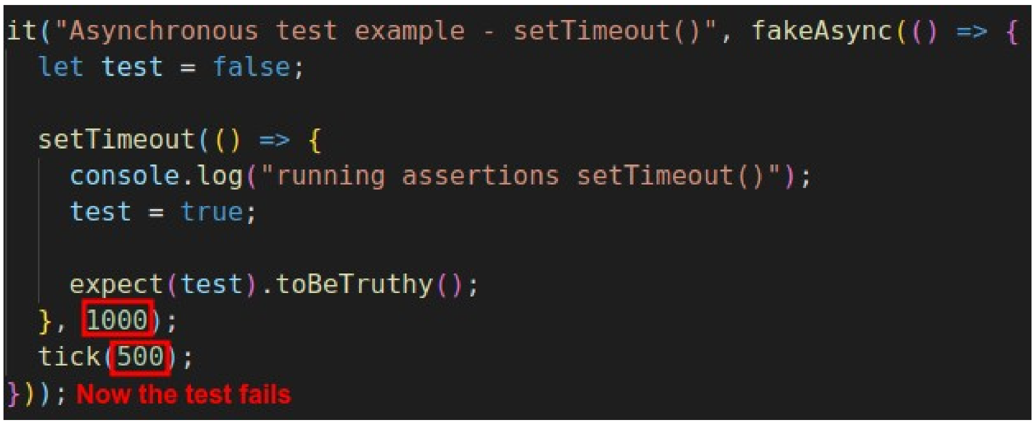
- Angular Fake Async test utility is a mechanism that executes the test block and detects the presence of asynchronous operations such as *setTimeout*, *setInterval*, *request animation frame* and many other asynchronous browser APIs. This mechanism detects the execution of these multiple asynchronous calls, and it would wait for all of those asynchronous operations to complete before considering the test completed.

- The Zone is going to detect all the browser asynchronous operations, such as *setTimeouts* that are triggered by the test block. Zone is going to wait for all those asynchronous operations to complete before considering that the spec is completed. So in a way, it's something very similar to Jasmine API, but it's a lot more flexible. This will allow us to test for components that launch multiple asynchronous operations, including components that launch some of those asynchronous operations nested inside each other, such as, for example, a setTimeout that triggers a setInterval, that triggers an API request, etc.

- The ***fakeAsync zone*** is going to replace the browser default implementation of the setTimeout function with its own custom function that simulates the passage of time.

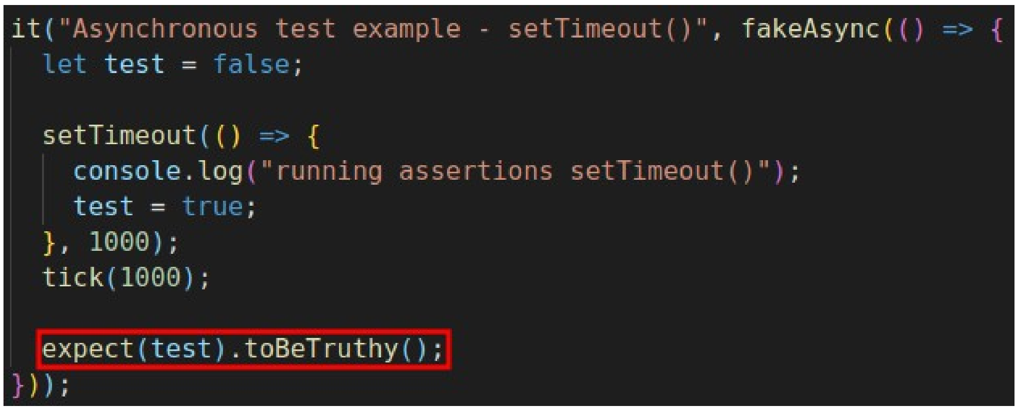


```
it("Asynchronous test example - setTimeout()", fakeAsync(() => {
  let test = false;

  setTimeout(() => {
    console.log("running assertions setTimeout()");
    test = true;

    expect(test).toBeTruthy();
  }, 1000);
  tick(1000);
}));
```



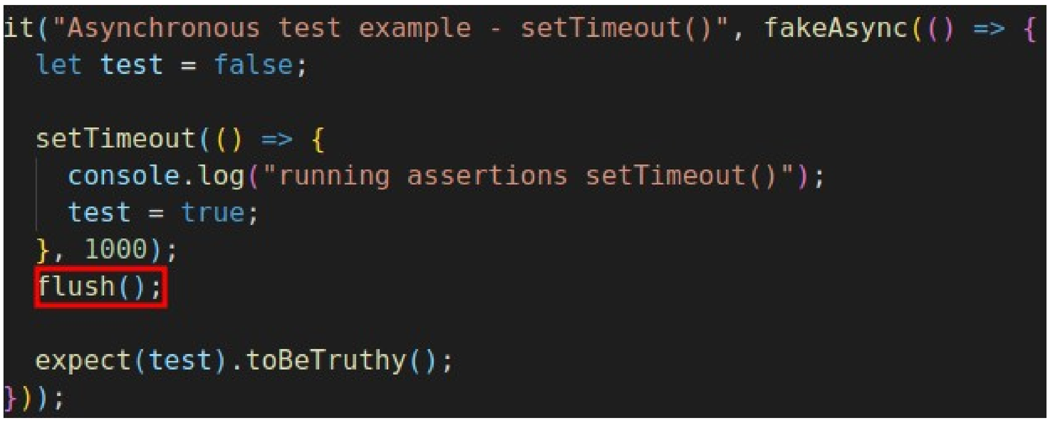
```
it("Asynchronous test example - setTimeout()", fakeAsync(() => {
  let test = false;

  setTimeout(() => {
    console.log("running assertions setTimeout()");
    test = true;

    expect(test).toBeTruthy();
  }, 1000);
  tick(500);
})); Now the test fails
```

→ The test fails because we have moved the clock only 500 milliseconds and that did not give enough time for the block to get executed with the assertions.



```
it("Asynchronous test example - setTimeout()", fakeAsync(() => {
  let test = false;

  setTimeout(() => {
    console.log("running assertions setTimeout()");
    test = true;
  }, 1000);
  tick(1000);

  expect(test).toBeTruthy();
}));
```

→ We can also move the assertion below *tick*. This is one of the main advantages that we have by using fakeAsync when compared to using the Jasmine Done callback because we no longer have to write assertions in nested code blocks.



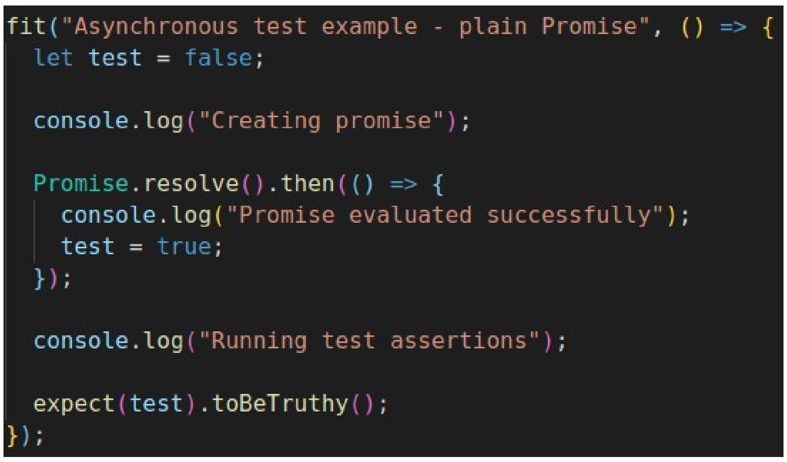
```
it("Asynchronous test example - setTimeout()", fakeAsync(() => {
  let test = false;

  setTimeout(() => {
    console.log("running assertions setTimeout()");
    test = true;
  }, 1000);
  flush();

  expect(test).toBeTruthy();
}));
```

→ flush() is a mechanism for making sure that all the synchronous operations are completed before the test can be completed.

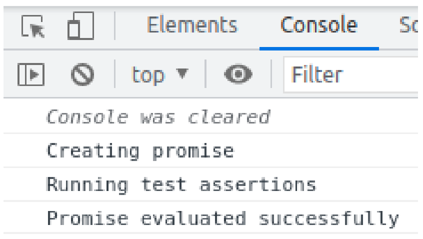
- Testing Promised-based code - Introduction to Microtasks

- With promises things are a bit different. One example is the order on which promises are executed by the JavaScript runtime when compared, for example, to other asynchronous operations such as setTimeouts.

→ Given this code:

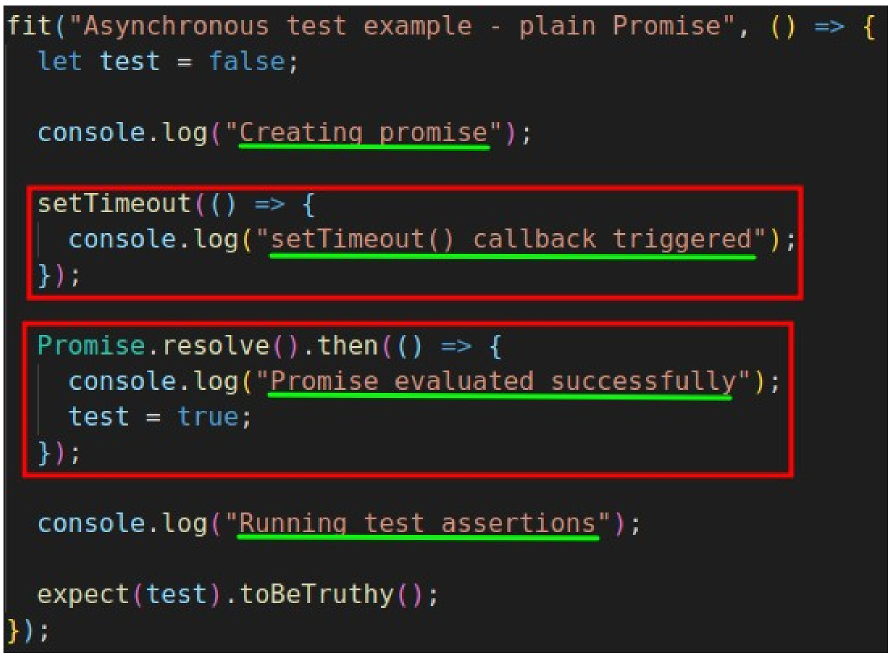


```
fit("Asynchronous test example - plain Promise", () => {
  let test = false;

  console.log("Creating promise");

  Promise.resolve().then(() => {
    console.log("Promise evaluated successfully");
    test = true;
  });

  console.log("Running test assertions");

  expect(test).toBeTruthy();
});
```

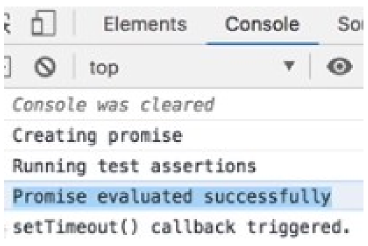


```
Console was cleared
Creating promise
Running test assertions
Promise evaluated successfully
```

→ Even though the Promise is getting immediately resolved, the *then block* is not getting evaluated synchronously immediately after creating the promise.



```
fit("Asynchronous test example - plain Promise", () => {
  let test = false;

  console.log("Creating promise");

  setTimeout(() => {
    console.log("setTimeout() callback triggered");
  });

  Promise.resolve().then(() => {
    console.log("Promise evaluated successfully");
    test = true;
  });

  console.log("Running test assertions");

  expect(test).toBeTruthy();
});
```



```
Console was cleared
Creating promise
Running test assertions
Promise evaluated successfully
setTimeout() callback triggered.
```

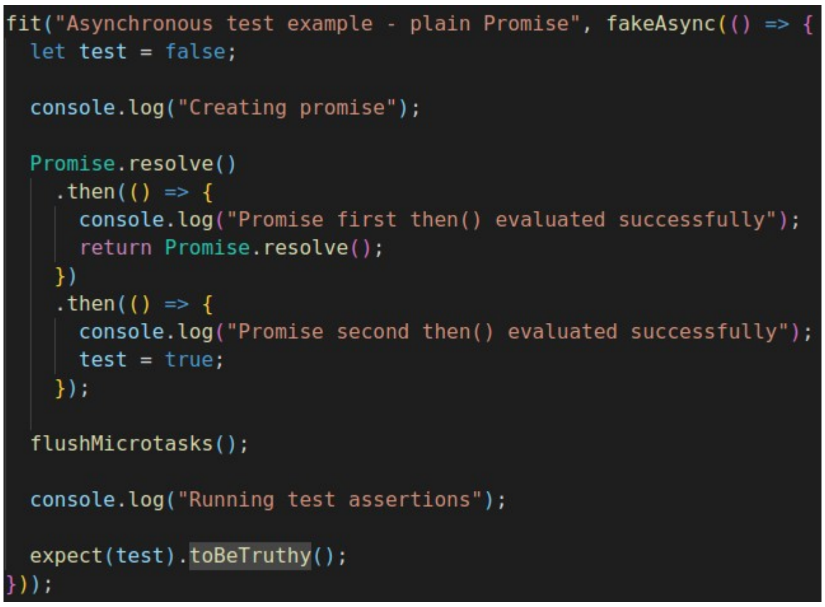
→ Even though the Promise was defined after the *setTimeout* and even though the *setTimeout* call has no delay associated to it, the Promise will still get evaluated first. So this seems like Promises somehow have some sort of priority over an operation such as set timeout.

- A Promise is considered a **micro-task**. This is in contrast with *setTimeout*, which is considered either a **macro-task** or simply a task. So the difference between both is that macro-tasks (*setTimeout*, *setInterval*, Ajax calls, mouse clicks and several other browser operations) are added to the ***event loop***. And between each of these macro-tasks, the browser rendering engine is going to get a chance to update the screen. And this is unlike the case of micro-tasks, such as, for example, Promises. So Promises are added to their own separate queue, which is different than the queue where tasks such as, for example, setTimeout or setInterval are added.

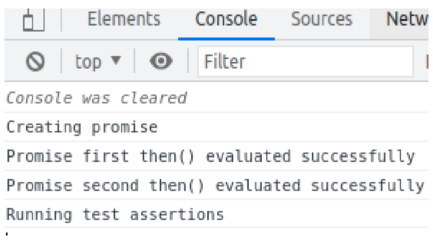
- Between the execution of two micro-tasks the browser will not get the chance to update the view.

- The browser is going to execute any synchronous code. Then the browser is going to empty the micro-task queue, which in general means to resolve all the Promises scheduled by our code. And only after the micro-task queue is empty will the browser then process another major tasks, such as for example, a setTimeout.

- The key thing to remember is that the browser runtime has two different types of queues for two different types of asynchronous operations. So some asynchronous operations are considered more lightweight, such as Promises, and they go into the micro-task. Other asynchronous operations such as, for example setTimeout go into the task queue.



```
fit("Asynchronous test example - plain Promise", fakeAsync(() => {
  let test = false;

  console.log("Creating promise");

  Promise.resolve()
    .then(() => {
      console.log("Promise first then() evaluated successfully");
      return Promise.resolve();
    })
    .then(() => {
      console.log("Promise second then() evaluated successfully");
      test = true;
    });

  flushMicrotasks();

  console.log("Running test assertions");

  expect(test).toBeTruthy();
}));
```



```
Console was cleared
Creating promise
Promise first then() evaluated successfully
Promise second then() evaluated successfully
Running test assertions
```

- Between two macro-tasks, such as for example, between two *setTimeouts*, the browser has the opportunity of updating the DOM. So that is unlike a micro-task, such as a Promise, where the browser does not have the opportunity of updating the DOM.

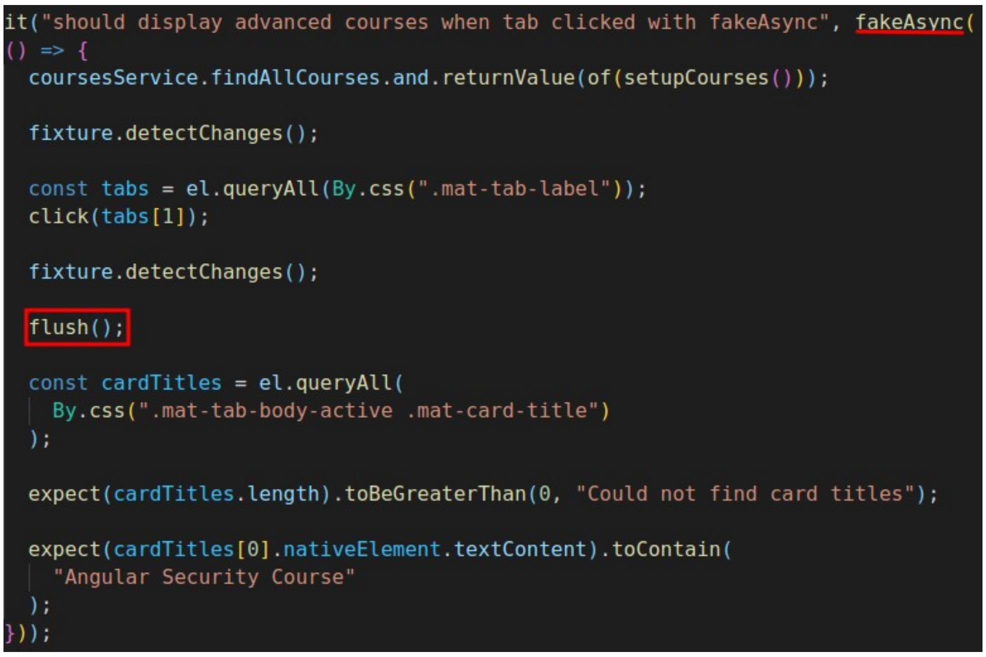
→ Example mixing micro-task and macro-task:

```
it("Asynchronous test example - Promises + setTimeout()", fakeAsync(() => {
 let counter = 0;

 Promise.resolve().then(() => {
   counter += 10;

   setTimeout(() => {
     counter += 1;
   }, 1000);
 });

 expect(counter).toBe(0);
 flushMicrotasks();
 expect(counter).toBe(10);
 tick(500);
 expect(counter).toBe(10);
 tick(500);
 expect(counter).toBe(11);
}));
```

- Testing Observables

- Testing Observables is very similar to testing any other code that also uses setTimeout, promises, etc.
- There are multiple types of Observables. Some Observables are synchronous and don't even need the use of the fakeAsync test utility. And some other Observables are asynchronous because they use internally in their implementation things such as setTimeout, Promises, etc.

- Replacing done: "DoneFn" with "fakeAsync"



```
it("should display advanced courses when tab clicked with fakeAsync", fakeAsync(
() => {
  coursesService.findAllCourses.and.returnValue(of(setupCourses()));

  fixture.detectChanges();

  const tabs = el.queryAll(By.css(".mat-tab-label"));
  click(tabs[1]);

  fixture.detectChanges();

  flush();

  const cardTitles = el.queryAll(
    By.css(".mat-tab-body-active .mat-card-title")
  );

  expect(cardTitles.length).toBeGreaterThan(0, "Could not find card titles");

  expect(cardTitles[0].nativeElement.textContent).toContain(
    "Angular Security Course"
  );
}));
```

- Understanding the Angular waitForAsync() Test Zone

- *waitForAsync* works in a very different way than *fakeAsync*.

- In the case of *waitForAsync* we cannot call *flush()*, so we don't have full control over the emptying of the tasks. We also can't call *tick()* and control the passage of time inside the test.

- The *waitForAsync* is going to keep track of any asynchronous operations created by our code, such as *setTimeout*, *setInterval*, *Promises*. And the *waitForAsync* is then going to give us a callback that is going to notify us when all those asynchronous operations are completed. Using that callback, we will then be able to run our test assertions and confirm if the test is passing or not.

- In order to get access to the callback we use *fixture.whenStable()*.

- It's clearly not as convenient as *fakeAsync*. For one, we don't have control over the passage of time. We don't have control over the emptying of the micro-tasks and the task queue. And most of all, we don't have the possibility of writing our assertions in a synchronous looking way directly in the body of the test, instead that we need to use a promise and some nested code.

- With the fine grained control of the passage of time, we can write certain tests to test intermediate states of our components at specific points in time. That is just not possible with calling fixture.whenStable() callback. It's going to get call whenever all the operations get resolved.

- Differences between waitForAsync and fakeAsync

- One major feature of the *waitForAsync* that *fakeAsync* does not have is that *waitForAsync* supports HTTP requests. So if by some reason you need to write a test that is actually not a unit test, but, for example, an integration test that is doing HTTP calls to a backend, that would not be possible with *fakeAsync*.

- The goal of *fakeAsync* is to write our code in a fully synchronous way.

- If we have to test a component that is doing an actual HTTP backend call, then in that case we need to use the *waitForAsync* and not *fakeAsync*.

- I would recommend that you use fakeAsync as much as possible due to the more fine grained control over the test and also by the nicer looking way that the test is organized with the assertions nicely at the bottom. Leave the use of waitForAsync only in the rare cases where you needed although this should almost never be necessary. In general, waitForAsync is really only used in the beforeEach block of your test suites.

- It's important to keep in mind that both fakeAsync and waitForAsync are test utilities aimed at assisting in testing asynchronous functionality. So we should only use them if necessary.

- The majority of the functionality in our components and services should be testable using synchronous testing techniques. Only in rare cases we really need to write asynchronous tests. And in those cases, the fakeAsync test utility with the waitForAsync is usually the best choice.