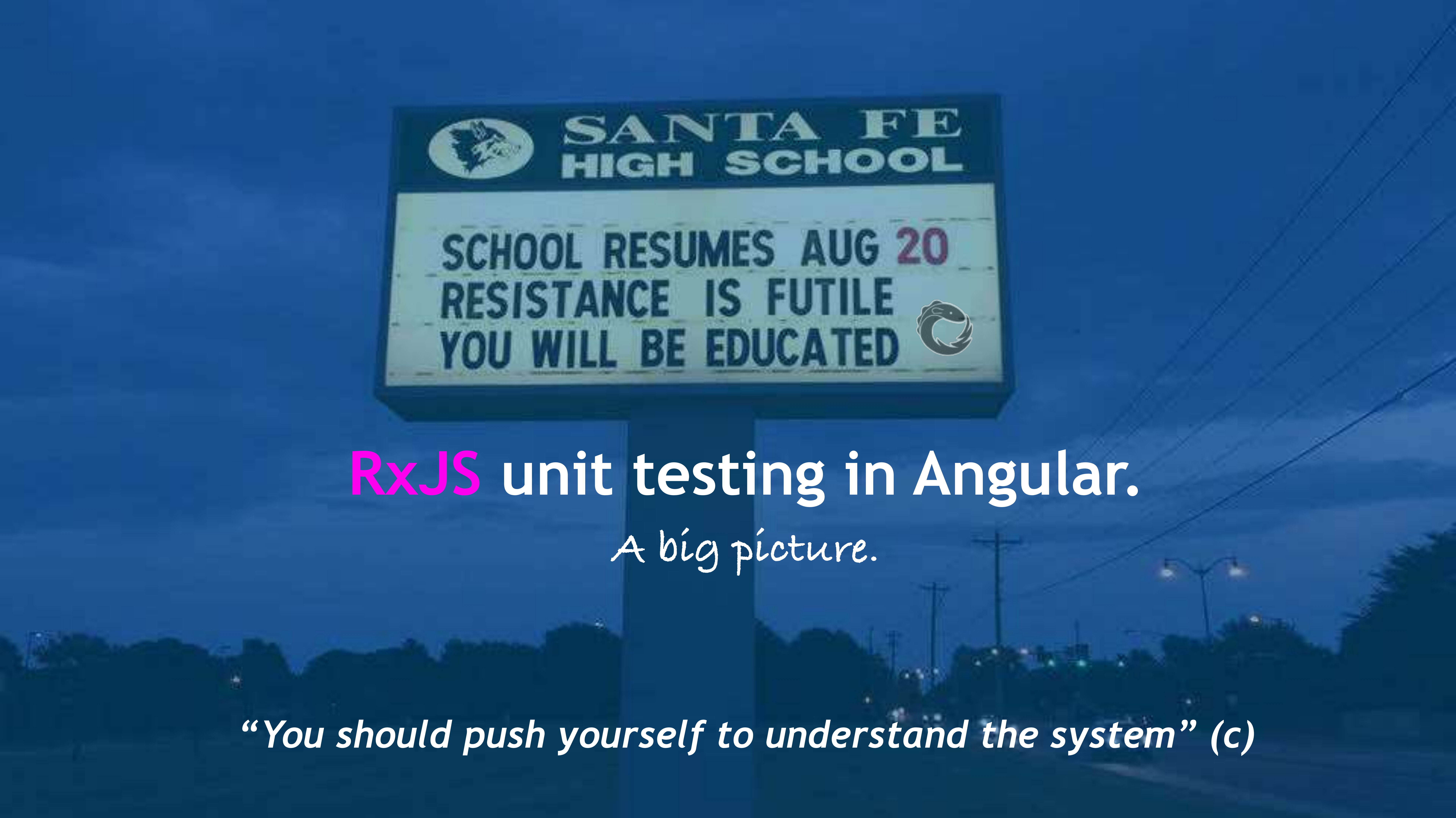


RxJS unit testing in Angular.

A big picture.

"You should push yourself to understand the system" (c)

About myself

- Senior Front-end developer
- Working in commercial projects using Angular and RxJS for last 4 years.
- Angular and RxJS mentor
- Writer for “Angular in Depth” blog.
- Video-course's author on Udemy
  “Hands-on RxJS for Web development” (paid)
  “RxJS unit testing in Angular” (free)
- “Angular can waste you time” Youtube series author
- Married, father of two playful sons:)

Oleksandr Poshtaruk

Twitter: @El_extremal

Codementor.io: https://bit.ly/2Qkl9Hm

Skype/E-mail: kievsash@ukr.net

RxJS Schedulers

Jasmine ‘done' callback

VirtualTimeScheduler

TestScheduler

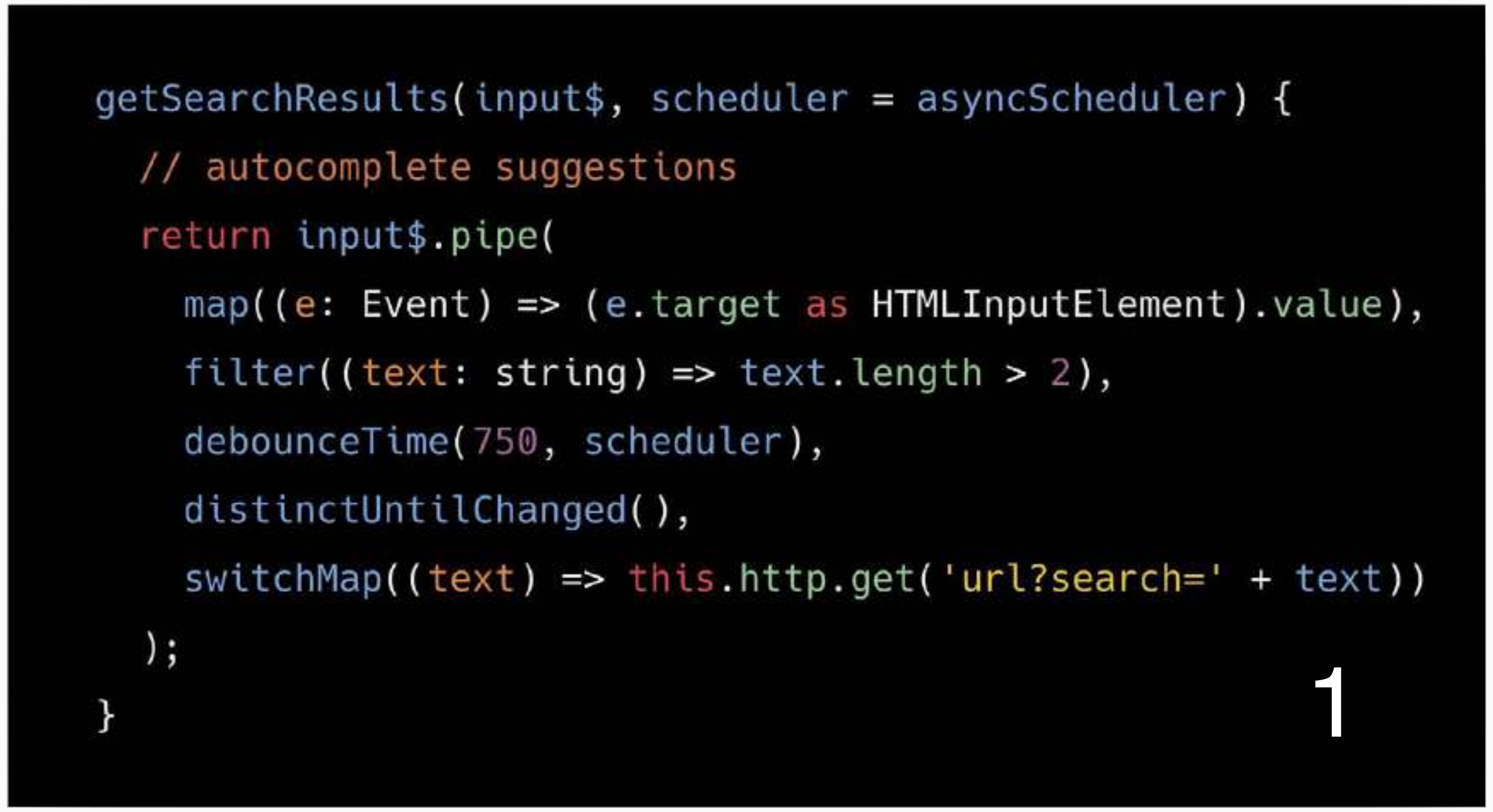
Jasmine-marbles

TestScheduler.run

rxjs-marbles

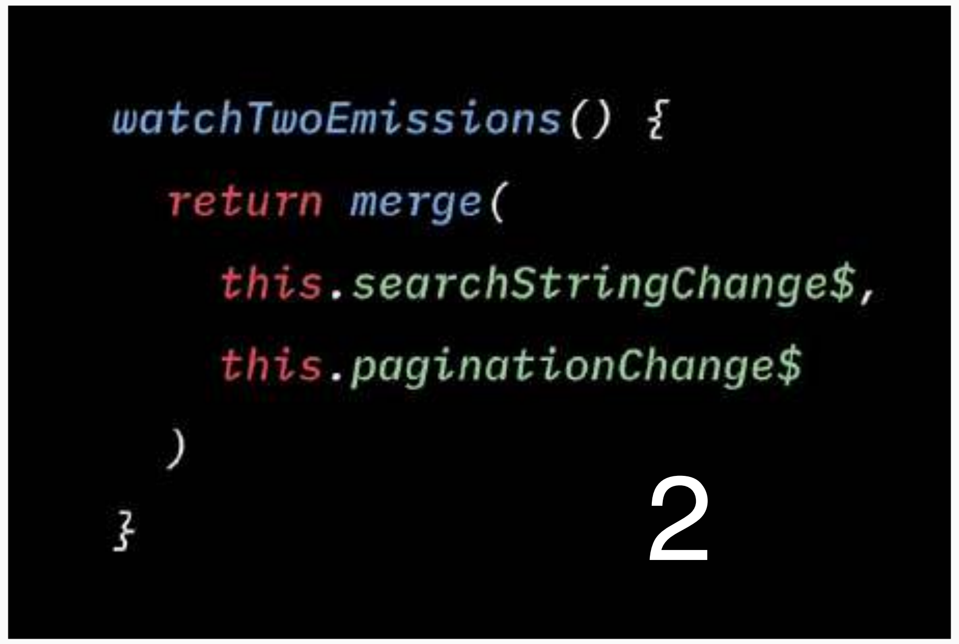
Angular fakeAsync

Test-cases



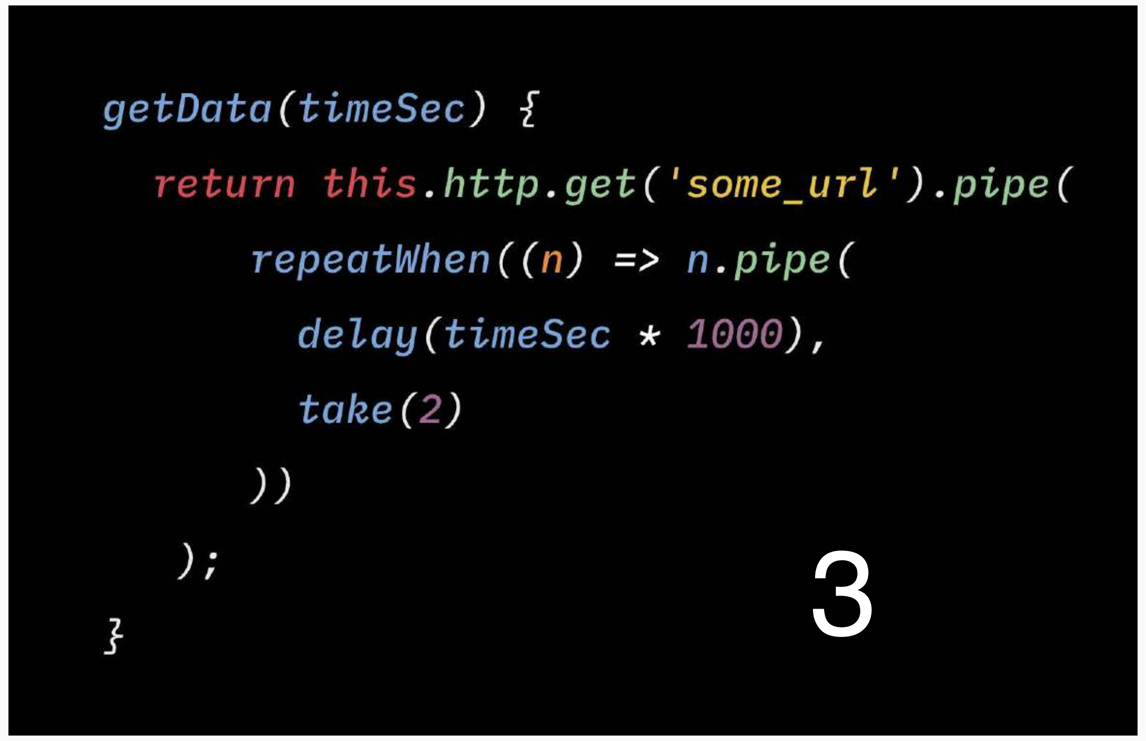
```
getSearchResults(input$, scheduler = asyncScheduler) {
  // autocomplete suggestions
  return input$.pipe(
    map((e: Event) => (e.target as HTMLInputElement).value),
    filter((text: string) => text.length > 2),
    debounceTime(750, scheduler),
    distinctUntilChanged(),
    switchMap((text) => this.http.get('url?search=' + text))
  );
}
```

1



```
watchTwoEmissions() {
  return merge(
    this.searchStringChange$,
    this.paginationChange$
  )
}
```

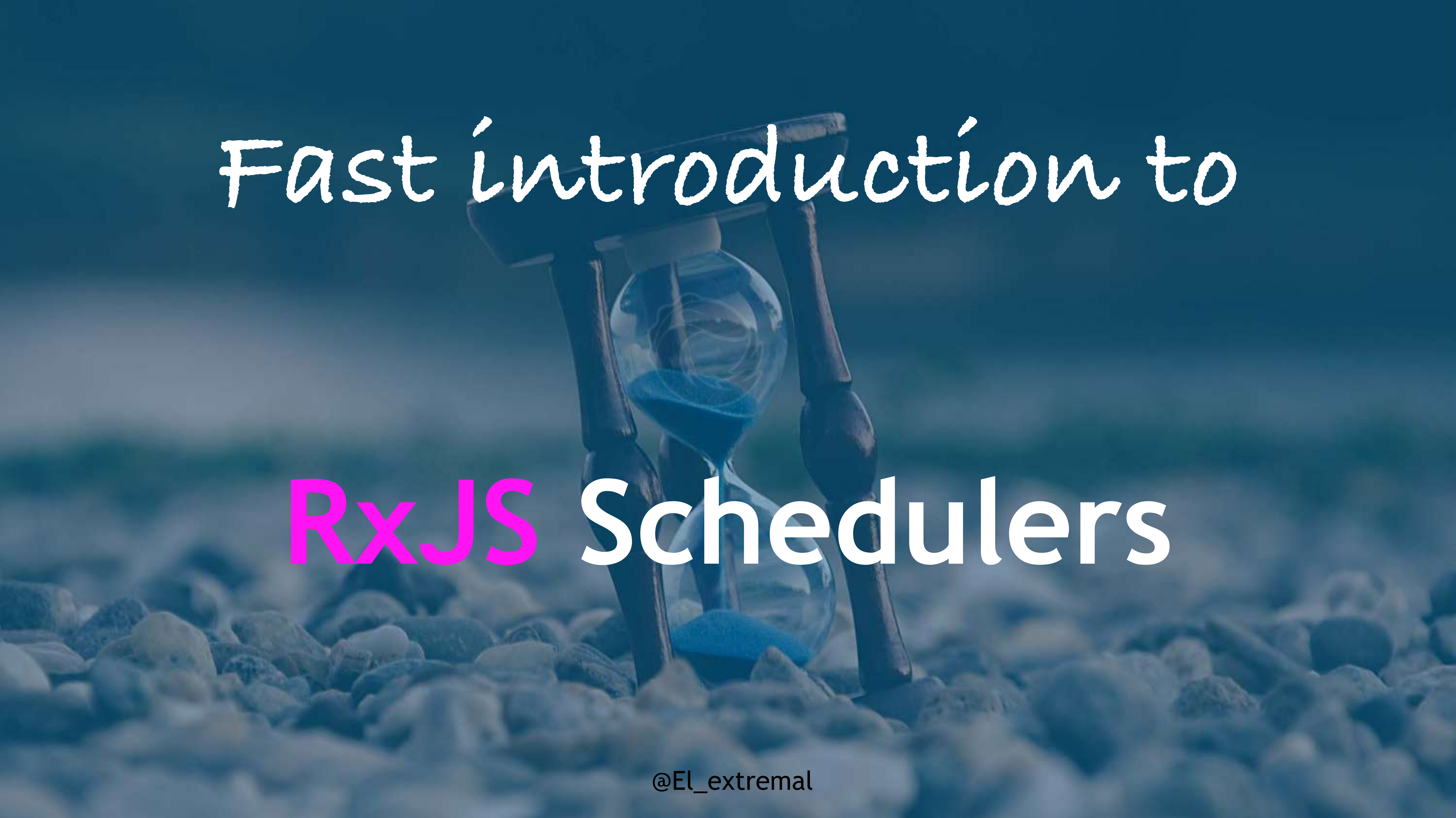
2



```
getData(timeSec) {
  return this.http.get('some_url').pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000),
        take(2)
      ))
    );
}
```

3

@El_extremal



Fast introduction to

RxJS Schedulers

@El_extremal

Event Loop

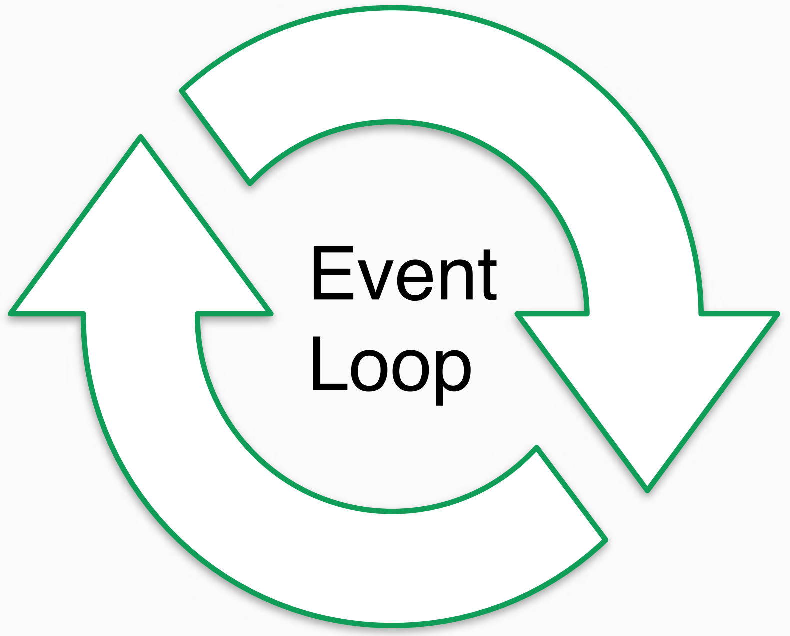
Current macrotask

Executed code:

```
console.log('1');
setTimeout(() => console.log('2'))
Promise.resolve().then(
  () => console.log('3')
)
```

3

Microtasks



2

Eventloop queue for macrotasks

Web API

setInterval
setTimeout
requestAnimationFrame

Console

1
3
2

@El_extremal

Event Loop

Web API

setInterval
setTimeout
requestAnimationFrame

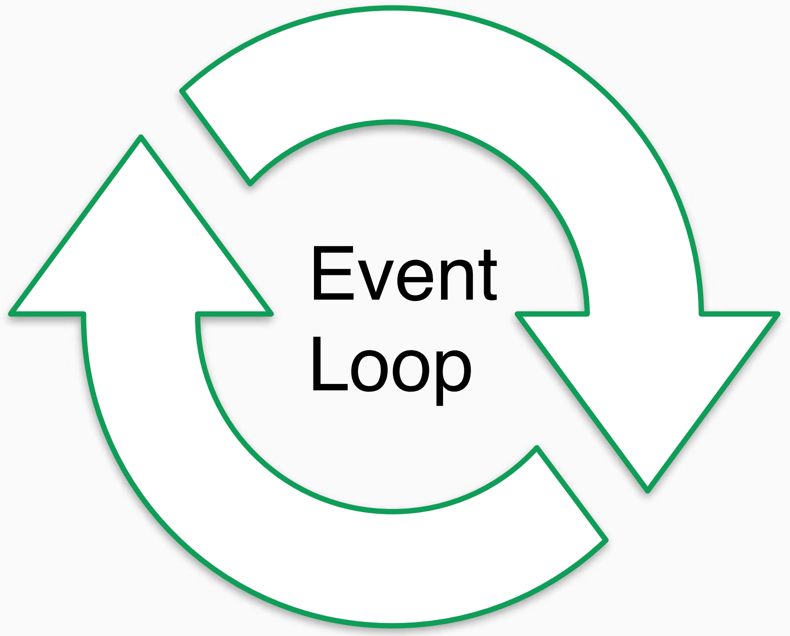
Current macrotask

Executed code:

```
merge(
    of(1),
    of(2, asyncScheduler),
    of(3, asapScheduler)
  ).subscribe(v => console.log(v))
```

3

Microtasks



Console

```
1
3
2
```

2

Eventloop queue for macrotasks

@El_extremal

What is Scheduler?

A scheduler is a data-structure that controls when emissions are delivered...

Scheduler name	When scheduled	How it works
<no scheduler>	**Current Macrotask**	Sync loop is used for some operators (**of, range**)
queueScheduler	**Current Macrotask**	Schedules values emission in a queue and then performa emissions (same macrotask).
asapScheduler	**Microtask**	Schedules on the micro task queue (like Promises do). Emission is being performed just after current macro task (as soon as possible).
asyncScheduler	**Another macro task**	Works like setInterval to delay value emissions in another macrotask.
animationFrameScheduler	**Another macro task**	Emission is aligned with browser re-draw event to create smooth animations

Observable unit testing challenges

1. Values are asynchronously emitted

2. More then one value may be emitted

3. Order and timings can matter

4. They can emit in current macrotask, subsequent microtask or emission can be scheduled in a future.

More sophisticated test-cases

1. ~~current macrotask (same tick)~~
2. microtask (Promise-based)
3. Future macrotask (setInterval, ...)
4. Streams combinations
   (combineLatest, merge, concat, etc)

Schedulers for unit testing

Scheduler	When scheduled	Description
VirtualTimeScheduler	**Current Macrotask**	Scheduler put all the emitted values in internal queue sorted according to specified delay. And with flush() method we can execute them instantly.
TestScheduler	**Current Macrotask**	Scheduler for unit testing. Inherits from VirtualTimeScheduler and have additional methods for convenient testing.

Imports for Schedulers

of(2, asyncScheduler)

Instance of AsyncScheduler

Import scheduler instance (used by operators):

```
import {asapScheduler, asyncScheduler} from 'rxjs';
```

Import scheduler class:

```
import {AsyncScheduler} from ‘rxjs/internal/scheduler/AsyncScheduler';
import {AsapScheduler} from ‘rxjs/internal/scheduler/AsapScheduler’;
import {VirtualTimeScheduler} from ‘rxjs’;
import {TestScheduler} from 'rxjs/testing';
```

Video’s

Philip Roberts - What the heck is the event loop anyway?
https://www.youtube.com/watch?v=8aGhZQkoFbQ

Michael Hladky - RxJS Schedulers In-Depth
https://www.youtube.com/watch?v=OQ1eiEw0kfs&t=384s

RxJS unit testing

Sync code unit testing

Code

```
getRange() {
  return of(0, 1, 2, 3).pipe(
      map(x => x + 1) // emits 1..2..3..4
  );
}
```

Test

```
describe('getRange', () => {
    it('should emit 4 values', () => {
        let result = [];
        let expectedResult = [1,2,3,4];

        getRange().subscribe((v) => result.push(v));

        expect(result).toEqual(expectedResult)
    })
})
```

SUCCESS

Nuances of async code unit testing

Code

```
getRange() {
  return of(0, 1, 2, 3, asyncScheduler).pipe(
      map(x => x + 1) // emits 1..2..3..4
  );
}
```

Test

```
describe('getRange', () => {
    it('should emit 4 values', () => {
        let result = [];
        let expectedResult = [1,2,3,4];

        getRange().subscribe((v) => result.push(v));

        expect(result).toEqual(expectedResult)
    })
})
```

FAILED

RxJS unit testing

#1. subscribe + done

Example: subscribe + done

Code

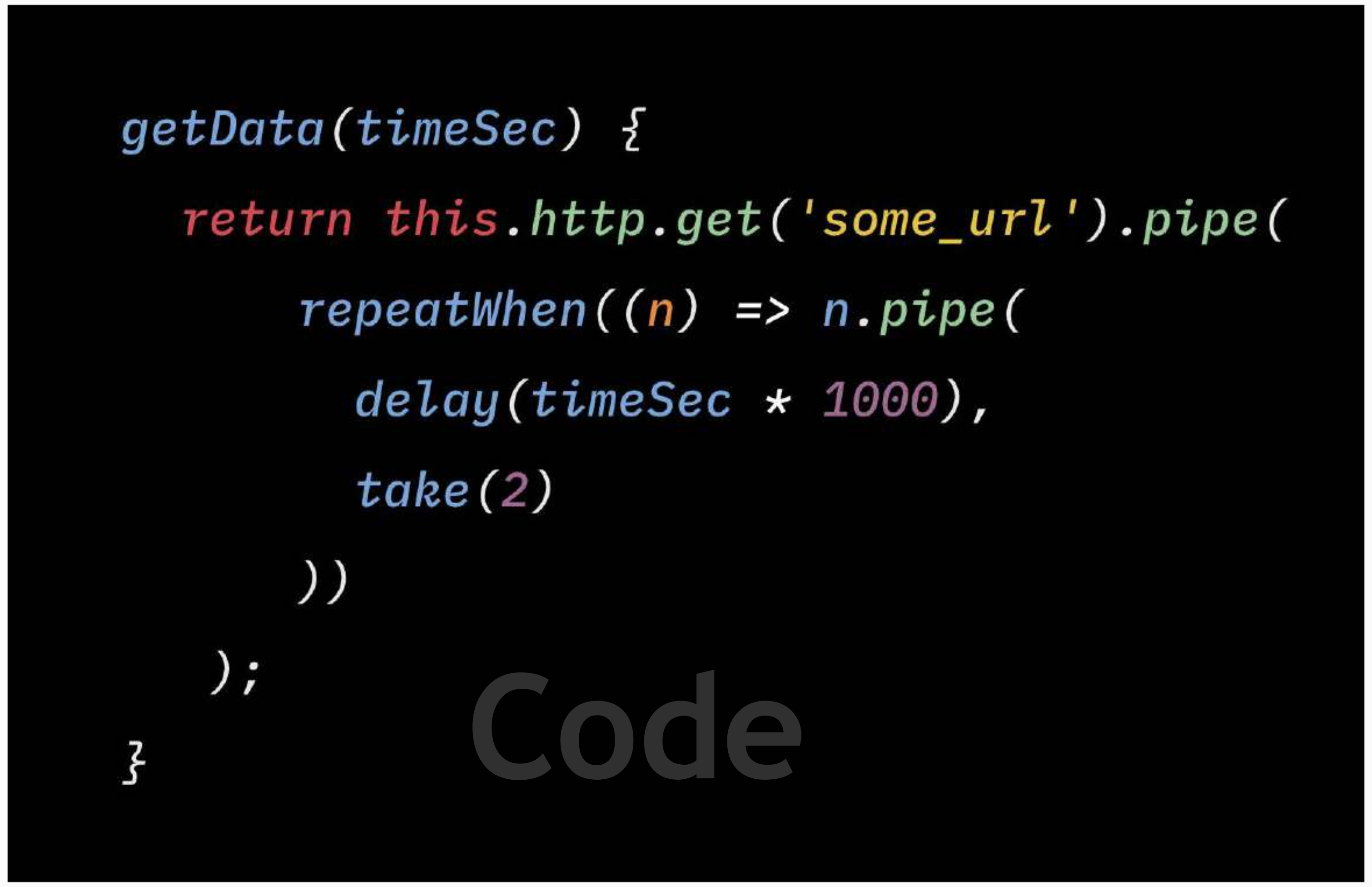
```
getRange() {
  return of(0, 1, 2, 3, asyncScheduler).pipe(
      map(x => x + 1) // emits 1..2..3..4
  );
}
```

Test

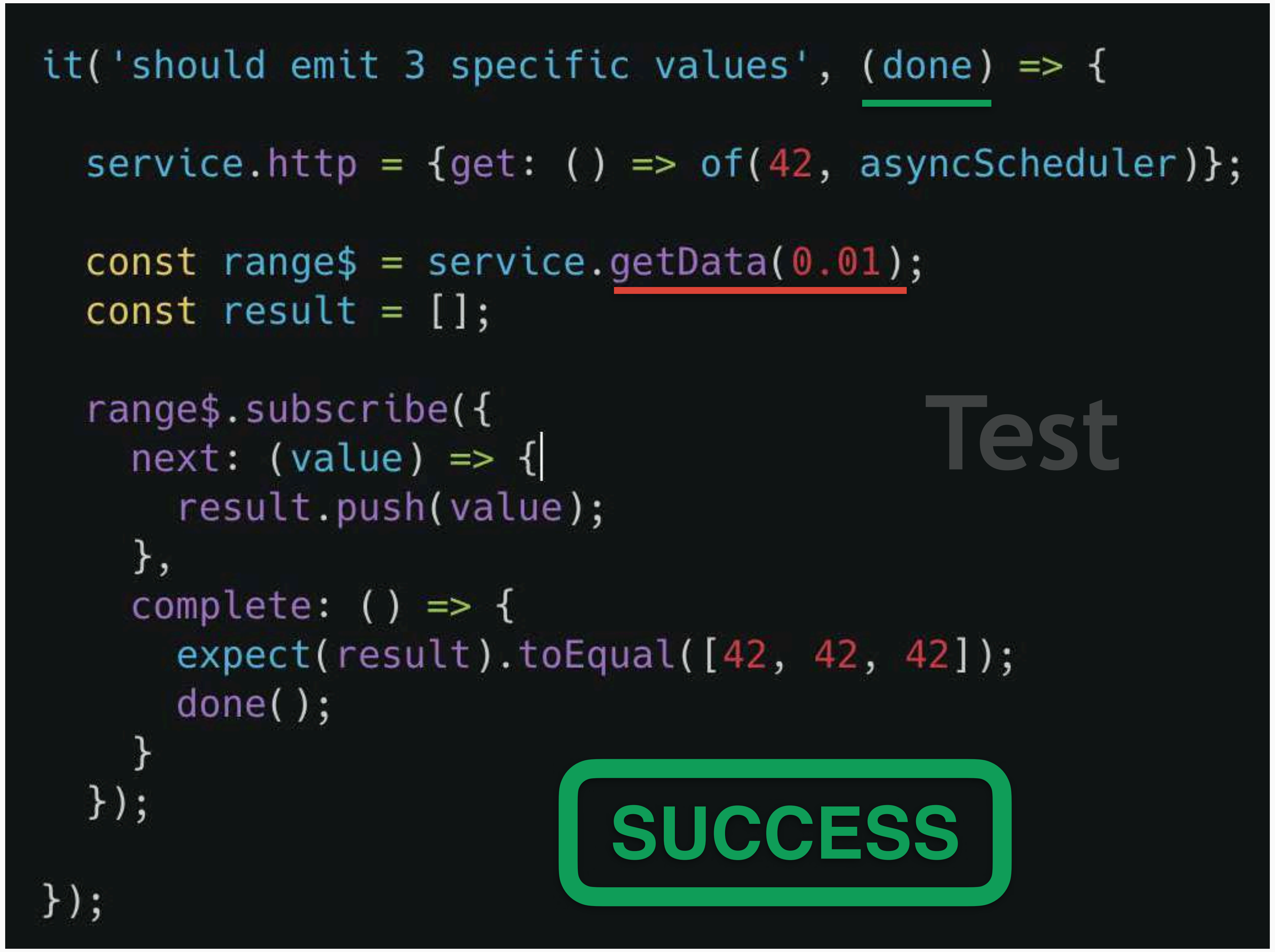
```
it('should emit 4 specific values', (done) => {
  const range$ = service.getRange();

  const result = [];
  range$.subscribe({
    next: (value) => {
      result.push(value);
    },
    complete: () => {
      expect(result).toEqual([0, 1, 2, 3]);
      done();
    }
  });
});
```

SUCCESS

Example 2: subscribe + done



```
getData(timeSec) {
  return this.http.get('some_url').pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000),
        take(2)
      ))
    );
}
```



```
it('should emit 3 specific values', (done) => {

  service.http = {get: () => of(42, asyncScheduler)};

  const range$ = service.getData(0.01);
  const result = [];

  range$.subscribe({
    next: (value) => {
      result.push(value);
    },
    complete: () => {
      expect(result).toEqual([42, 42, 42]);
      done();
    }
  });

});
```

Method #1 pros & cons

✓

1. Simple
2. Good for single value with no/very small delays

✗

1. Not visual- only final result is checked
2. Bad for distributed over time emissions with hardcoded timings

RxJS unit testing

#2. Using virtual time

a) VirtualTimeScheduler

Special Schedulers

Scheduler	When scheduled	Description
VirtualTimeScheduler	**Current Macrotask**	Scheduler put all the emitted values in internal queue sorted according to specified delay. And with flush() method we can execute them instantly.
TestScheduler	**Current Macrotask**	Scheduler for unit testing. Inherits from VirtualTimeScheduler and have additional methods for convenient testing.

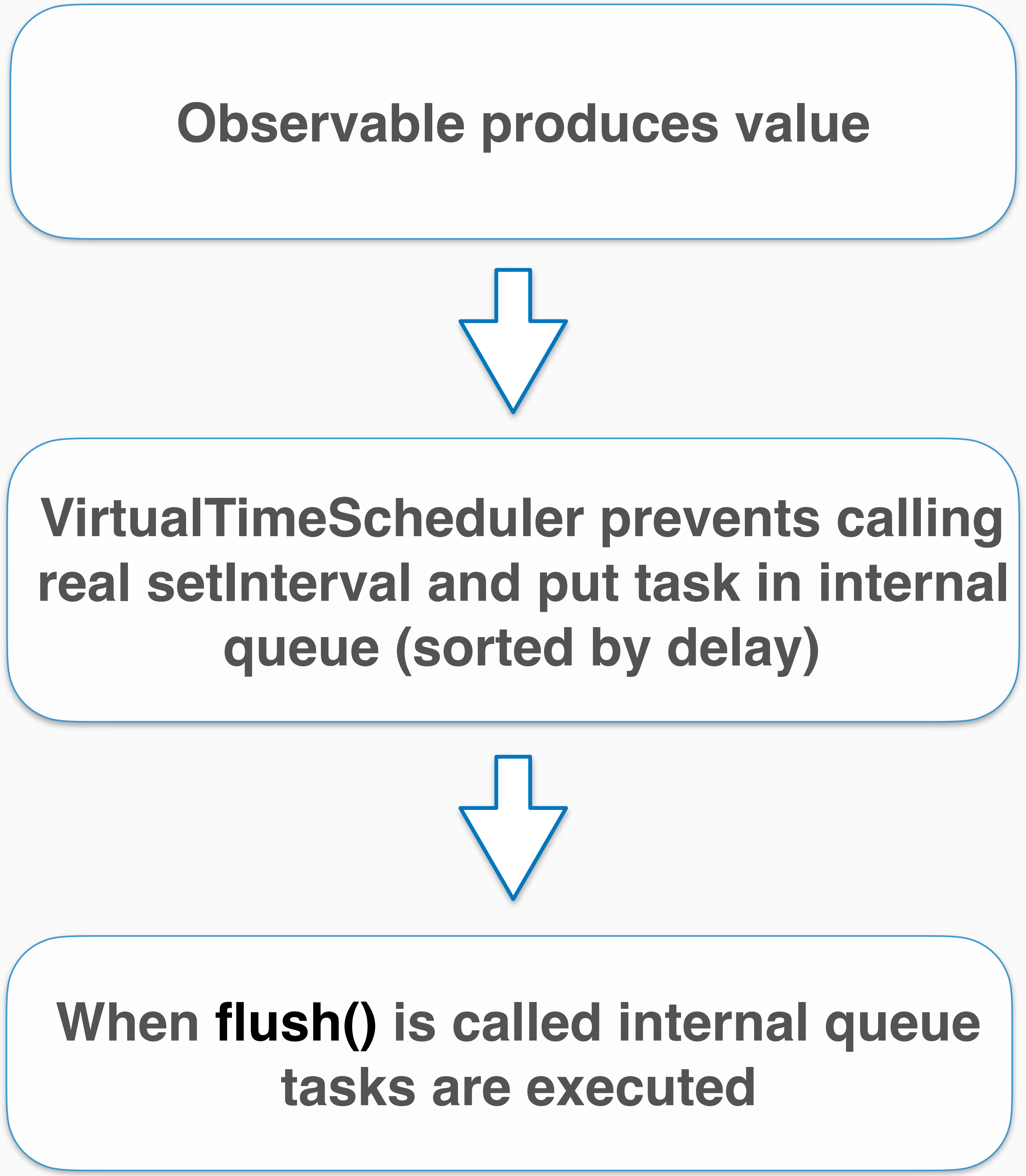
Inheritance diagram:

AsyncScheduler —> VirtualTimeScheduler —> TestScheduler

Method #2a: replace AsyncScheduler with VirtualTimeScheduler

VirtualTime
Scheduler

VirtualTime



Observable produces value

VirtualTimeScheduler prevents calling real setInterval and put task in internal queue (sorted by delay)

When **flush()** is called internal queue tasks are executed

Method #2 - VirtualTimeScheduler

1. Use VirtualTimeScheduler instead of AsyncScheduler

(we should add new scheduler param to production code methods)

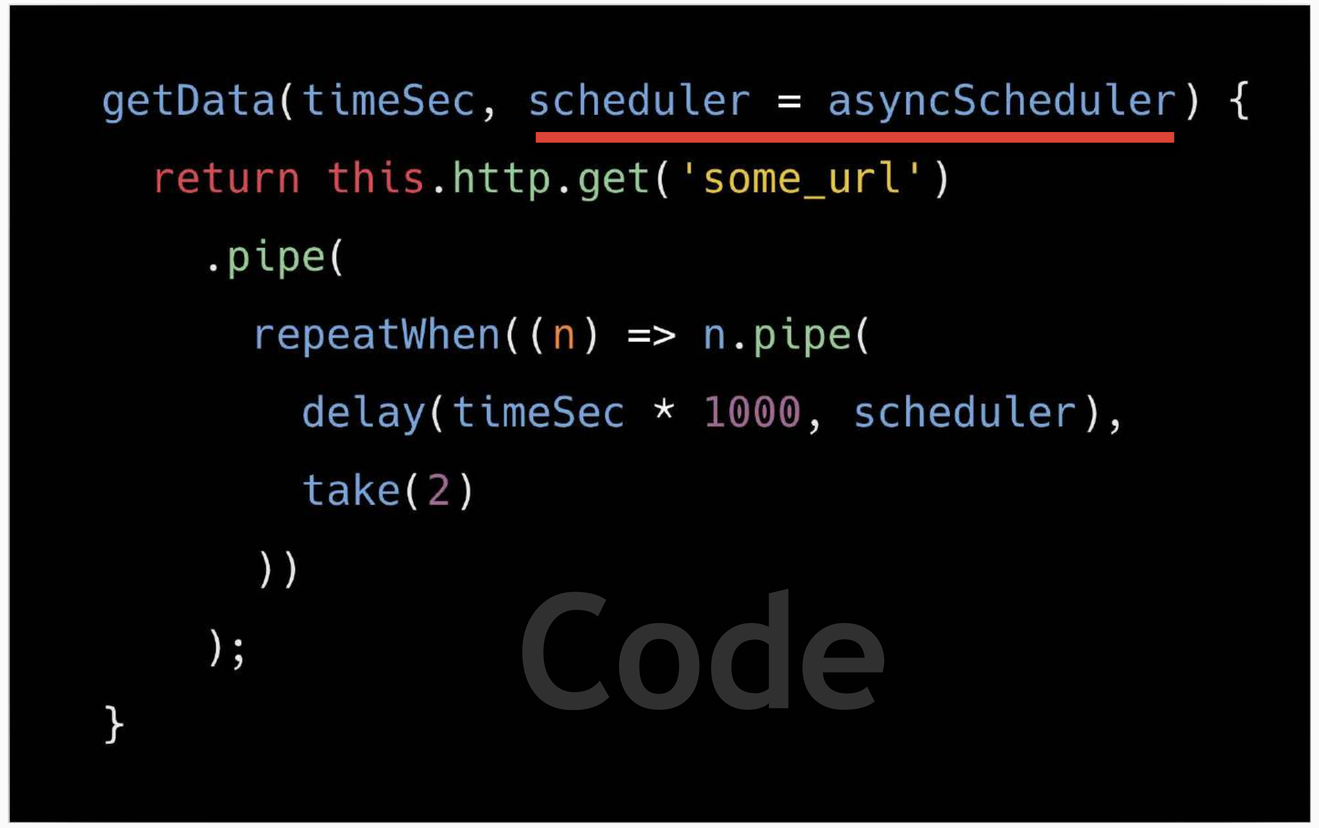
2. Run source$ observable with production delay values

3. Call VirtualTimeScheduler.flush()

3a. We can limit flush() timespan with VirtualTimeScheduler.*maxFrame* value

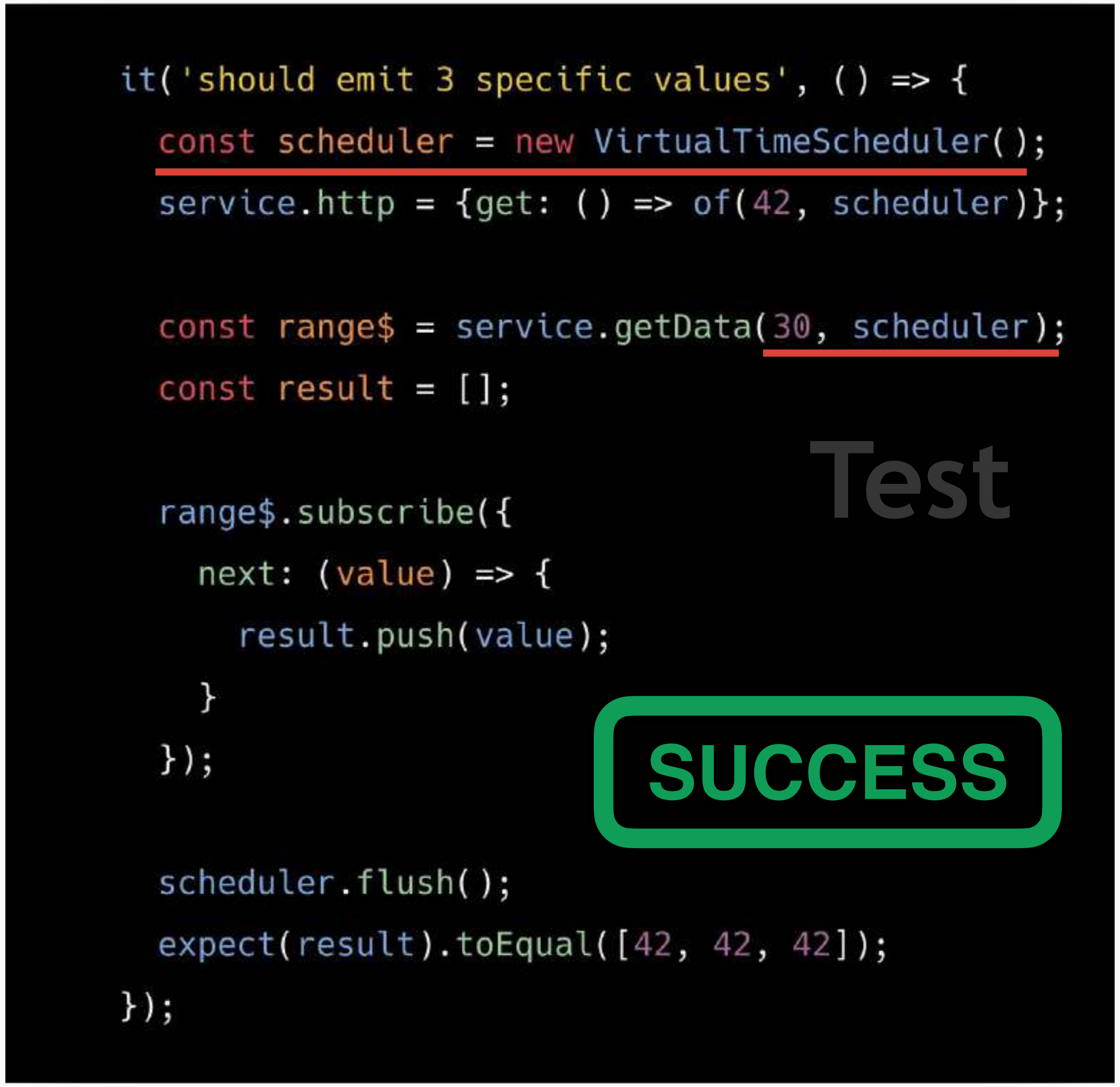
4. Check final result

Example #1 for VirtualTimeScheduler



```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

Code



```
it('should emit 3 specific values', () => {
  const scheduler = new VirtualTimeScheduler();
  service.http = {get: () => of(42, scheduler)};

  const range$ = service.getData(30, scheduler);
  const result = [];

  range$.subscribe({
    next: (value) => {
      result.push(value);
    }
  });

  scheduler.flush();
  expect(result).toEqual([42, 42, 42]);
});
```

Test

SUCCESS

Method #2 pros & cons

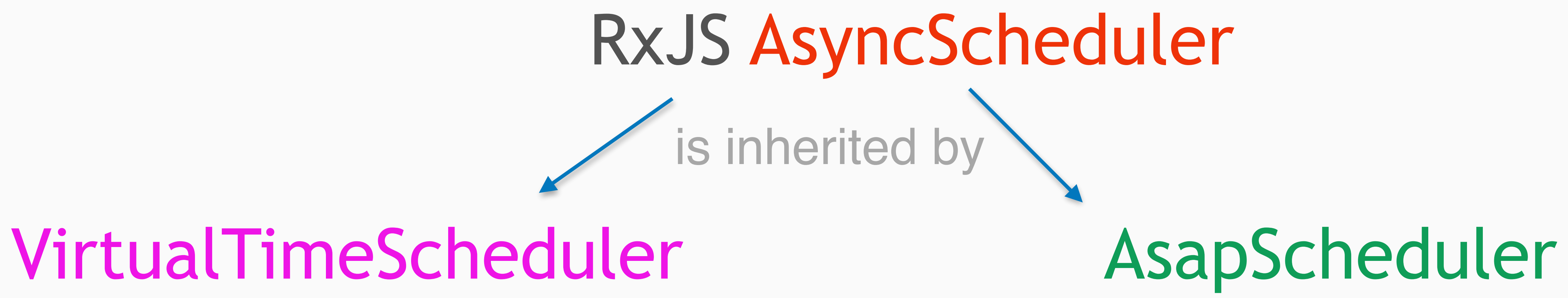
✓

1. Production delay values
2. We can test even with hardcoded values

✗

1. Not visual - only final result is checked
2. Additional method params is needed

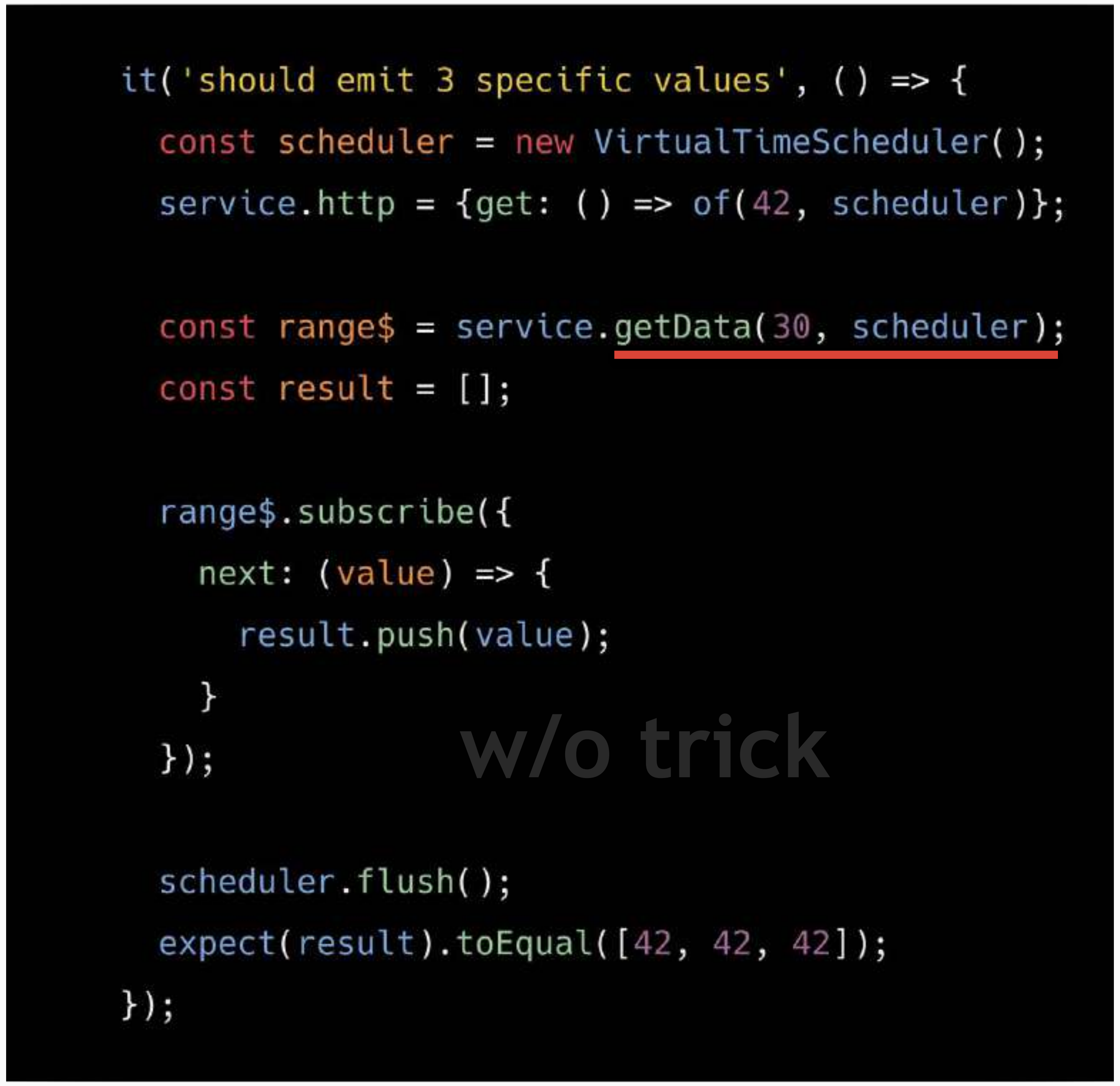
How to avoid *scheduler* argument in methods definitions?



Assign special property AsyncScheduler.delegate (and remove after test):

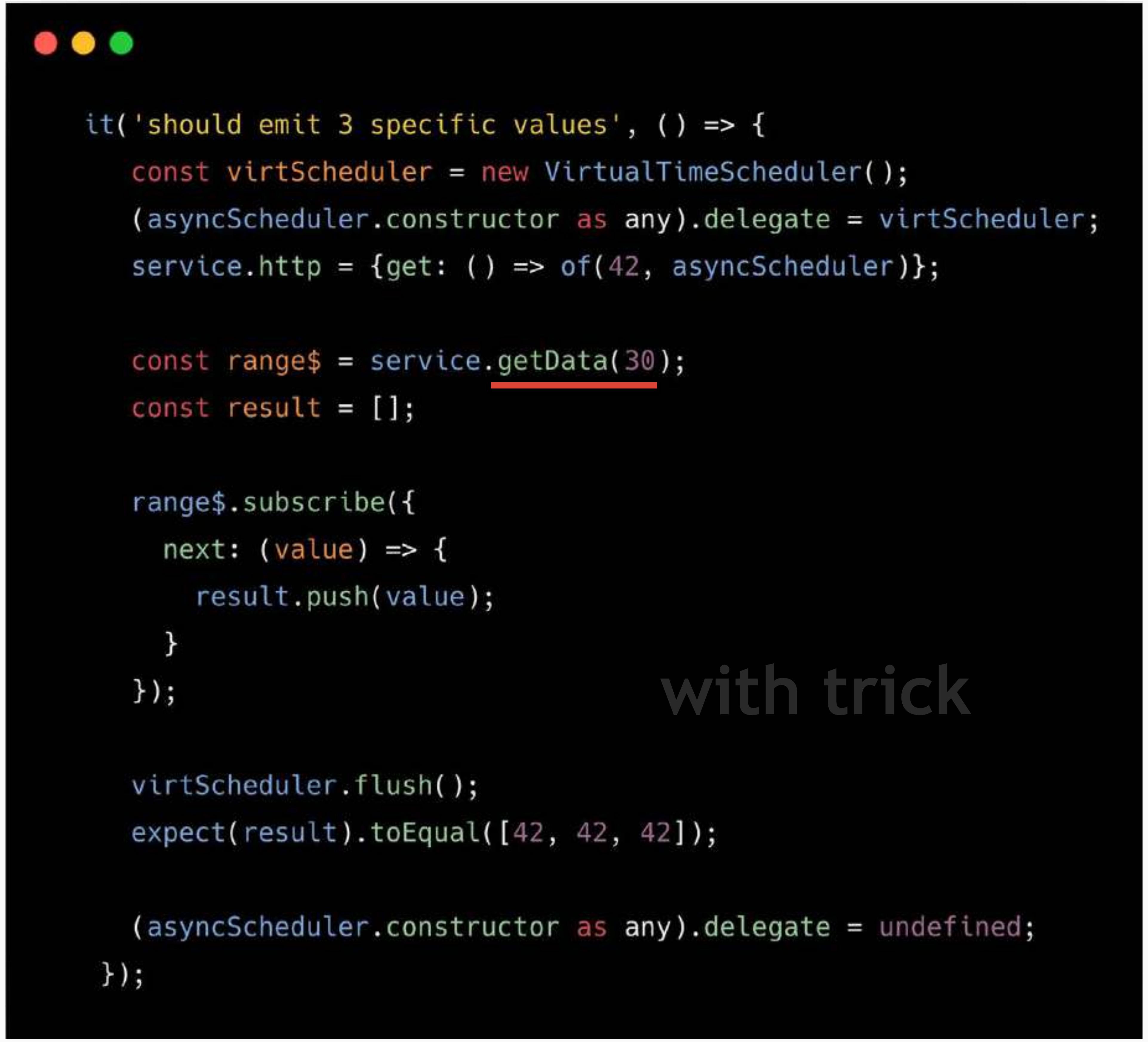
```
const virtScheduler = new VirtualTimeScheduler();
(asyncScheduler.constructor as any).delegate = virtScheduler;

  ... // test of code with asyncScheduler goes here

(asyncScheduler.constructor as any).delegate = undefined;
```

Trick - use AsyncScheduler.delegate method



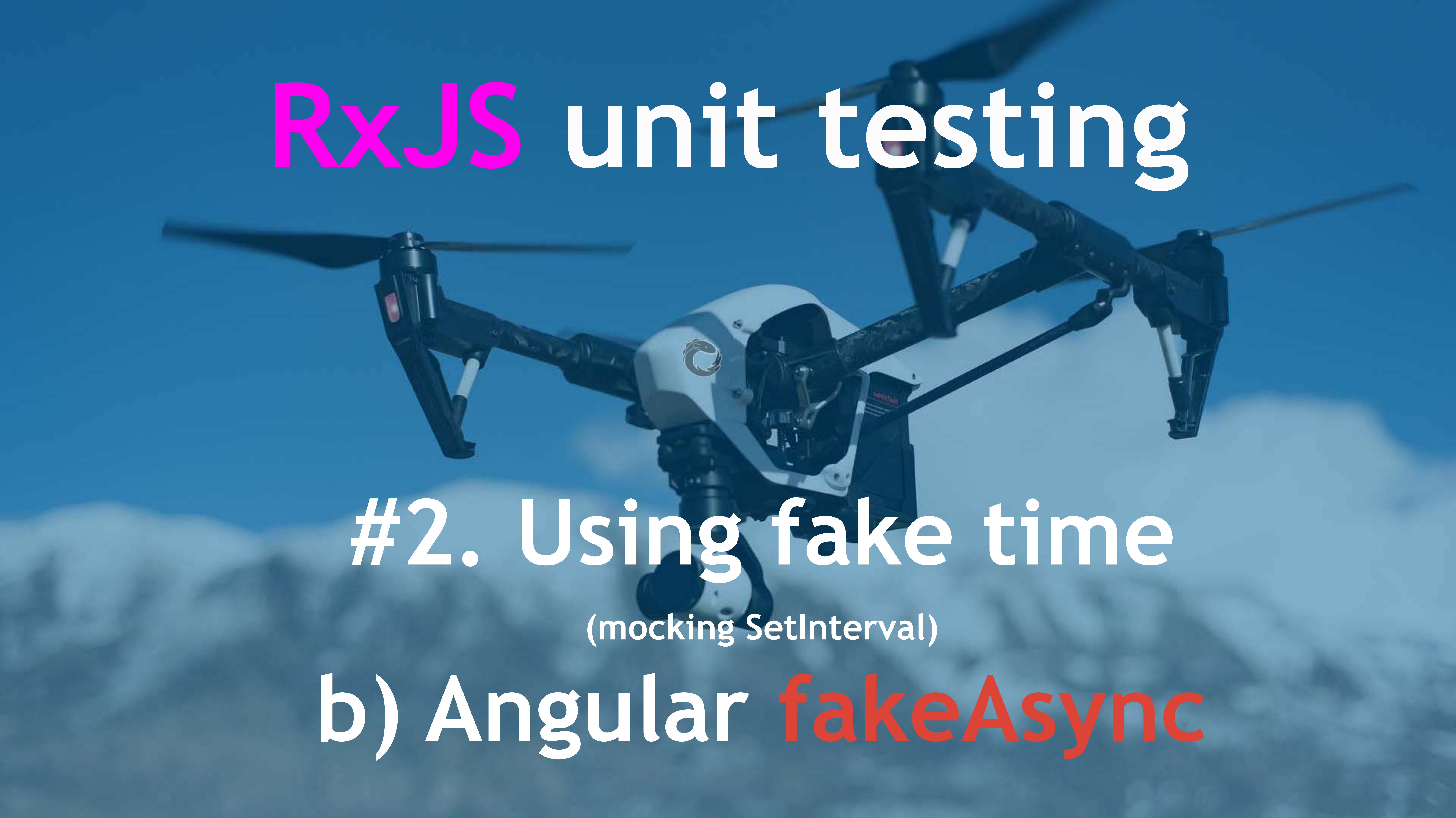
```
it('should emit 3 specific values', () => {
  const scheduler = new VirtualTimeScheduler();
  service.http = {get: () => of(42, scheduler)};

  const range$ = service.getData(30, scheduler);
  const result = [];

  range$.subscribe({
    next: (value) => {
      result.push(value);
    }
  });

  scheduler.flush();
  expect(result).toEqual([42, 42, 42]);
});
```

w/o trick



```
it('should emit 3 specific values', () => {
  const virtScheduler = new VirtualTimeScheduler();
  (asyncScheduler.constructor as any).delegate = virtScheduler;
  service.http = {get: () => of(42, asyncScheduler)};

  const range$ = service.getData(30);
  const result = [];

  range$.subscribe({
    next: (value) => {
      result.push(value);
    }
  });

  virtScheduler.flush();
  expect(result).toEqual([42, 42, 42]);

  (asyncScheduler.constructor as any).delegate = undefined;
});
```

with trick



RxJS unit testing

#2. Using fake time

(mocking SetInterval)

b) Angular fakeAsync

Method #4: fakeAsync instead of TestScheduler

VirtualTime Scheduler

Observable produces value

↓

VirtualTimeScheduler prevents calling real setInterval and put task in internal queue

↓

When flush() is called internal queue tasks are executed

VirtualTime

fakeAsync

Observable produces value

↓

asyncScheduler schedules emission by calling mocked **setInterval** (FakeAsyncTestZoneSpec)

↓

mocked setInterval execute emission tasks when tick is called

Faketime

How fakeAsync works?

1. It uses FakeAsyncTestZoneSpec instead of ngZone
2. Patched SetTimeout and Promise put tasks to special internal _schedulerQueue
3. tick(n) - flushes the internal _schedulerQueue by running all the tasks one by one with no delay.
4. flushMicrotasks() - flushes micro tasks queue.

Files:

angular/packages/zone.js/lib/testing/**fake-async.ts**

angular/packages/zone.js/lib/zone-spec/**fake-async-test.ts**

Example #1 with fakeAsync

Code

```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

Test

```
it('should emit 3 specific values', fakeAsync(() => {

  const range$ = service.getData(30);
  const result = [];

  range$.subscribe({
    next: (value) => {
      result.push(value);
    }
  });

  tick(60005);
  // 60000 = value + 30000ms + value + 30000ms + value + 5ms(to be sure)

  expect(result).toEqual([42, 42, 42]);
}));
```

SUCCESS

fakeAsync pros & cons

✓

1. Production delay values
2. We can test even with hardcoded values
3. A bit less code
4. You shouldn’t know how RxJS schedulers work

✗

1. Not visual - only final result is checked

RxJS unit testing

#2. Mocking SetInterval

c) VirtualTimeScheduler vs TestScheduler

VirtualTimeScheduler and TestScheduler

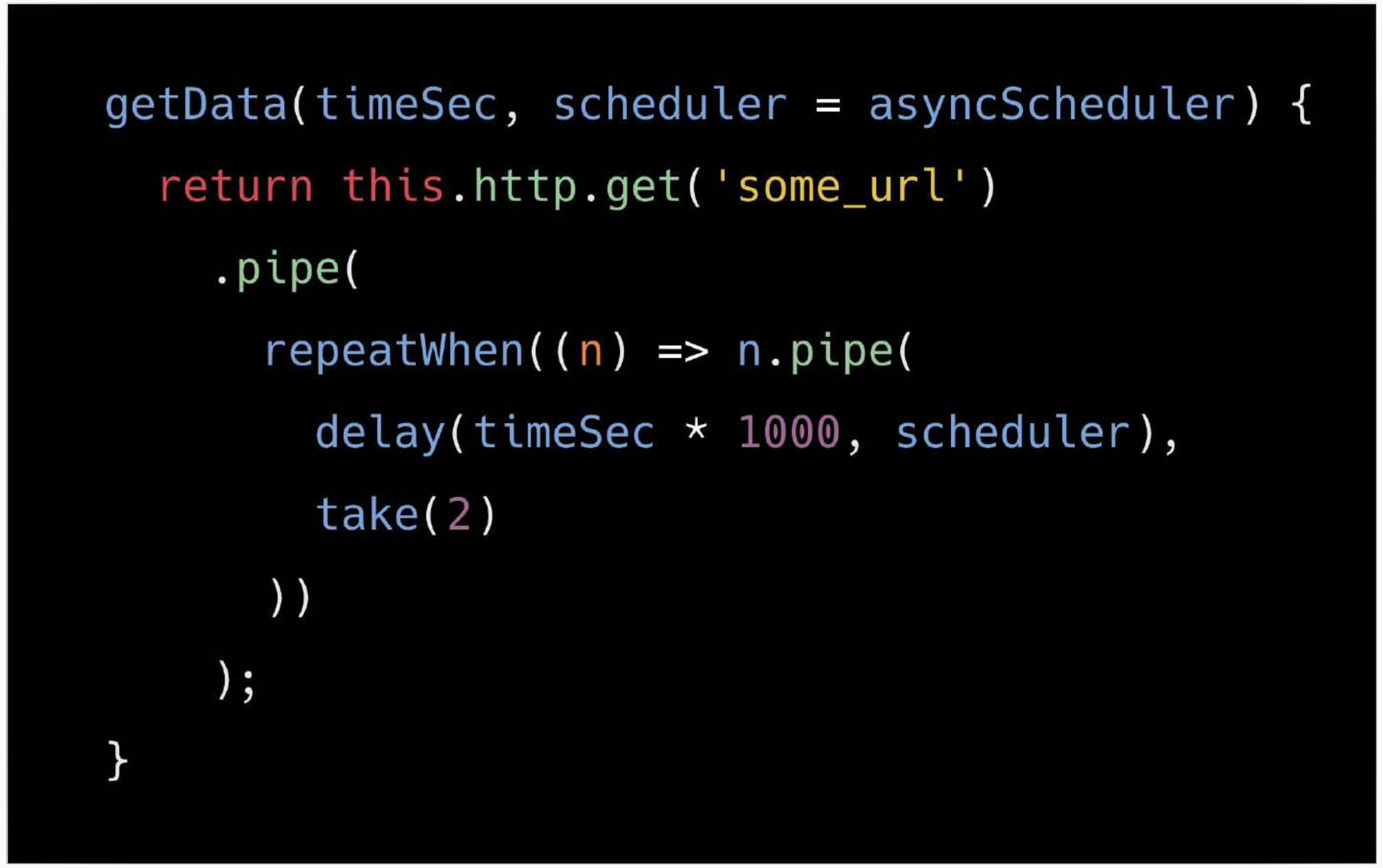
1. We can use **TestScheduler** the same way as VirtualTimeScheduler

2. TestScheduler.**maxFrame** = 750 while VirtualTimeScheduler.maxFrame === infinity

3. TestScheduler constructor demands **assertion equality** function while VirtualTimeScheduler does not

4. TestScheduler can do **marble testing**

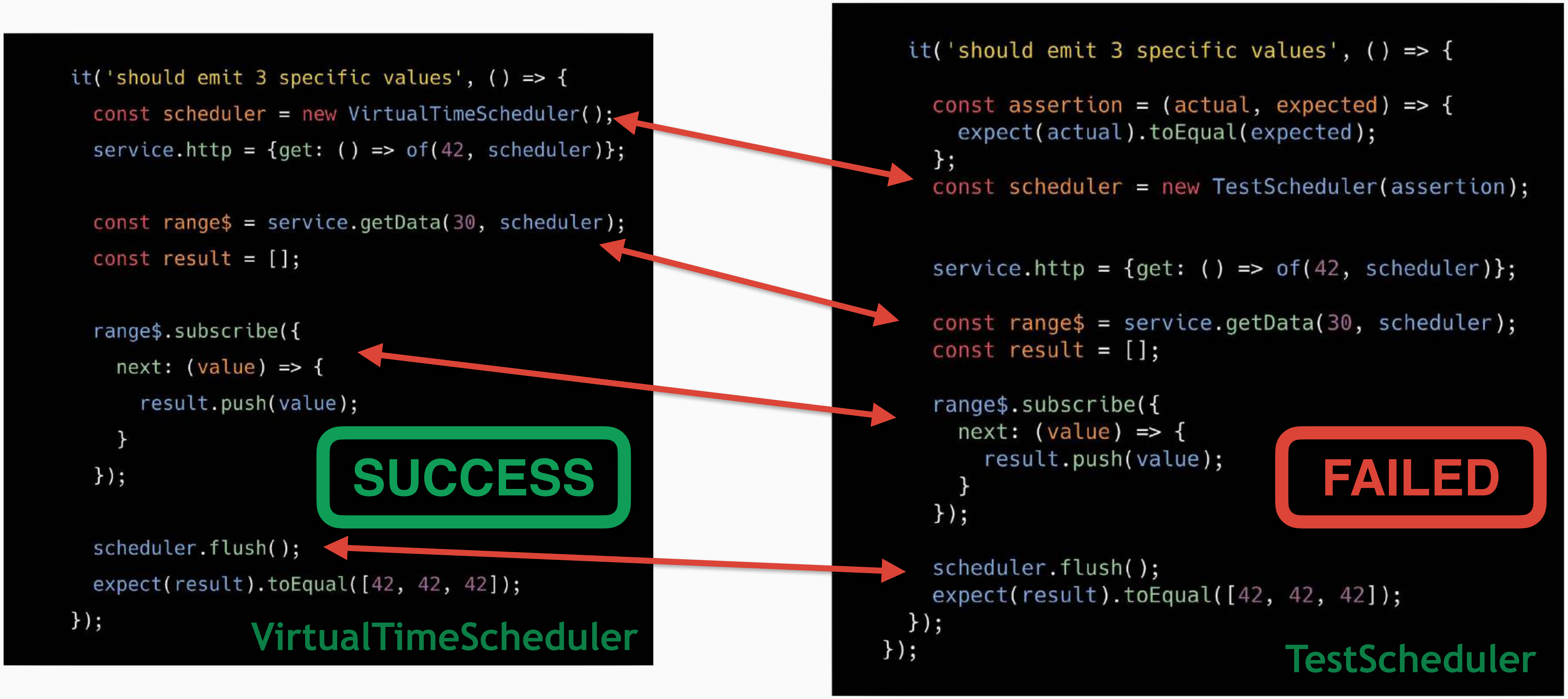
Example: VirtualTimeScheduler and TestScheduler



```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

getData(30) - will take 60 000ms to complete

Example: VirtualTimeScheduler and TestScheduler

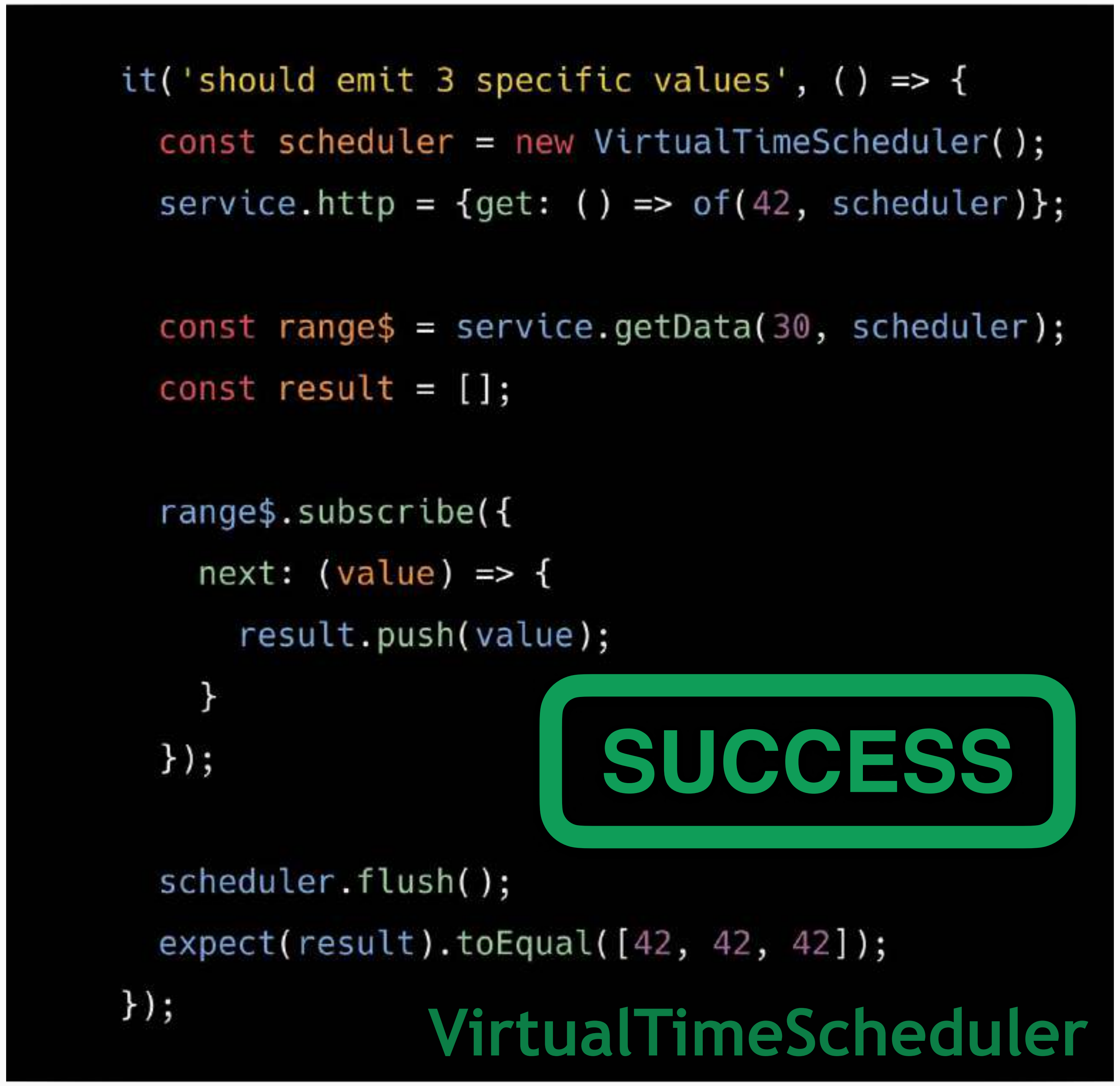


```
it('should emit 3 specific values', () => {
  const scheduler = new VirtualTimeScheduler();
  service.http = {get: () => of(42, scheduler)};

  const range$ = service.getData(30, scheduler);
  const result = [];

  range$.subscribe({
    next: (value) => {
      result.push(value);
    }
  });

  scheduler.flush();
  expect(result).toEqual([42, 42, 42]);
});
```

```
  it('should emit 3 specific values', () => {

    const assertion = (actual, expected) => {
      expect(actual).toEqual(expected);
    };
    const scheduler = new TestScheduler(assertion);


    service.http = {get: () => of(42, scheduler)};

    const range$ = service.getData(30, scheduler);
    const result = [];

    range$.subscribe({
      next: (value) => {
        result.push(value);
      }
    });

    scheduler.flush();
    expect(result).toEqual([42, 42, 42]);
  });
});
```

const defaultMaxFrame: number = 750;

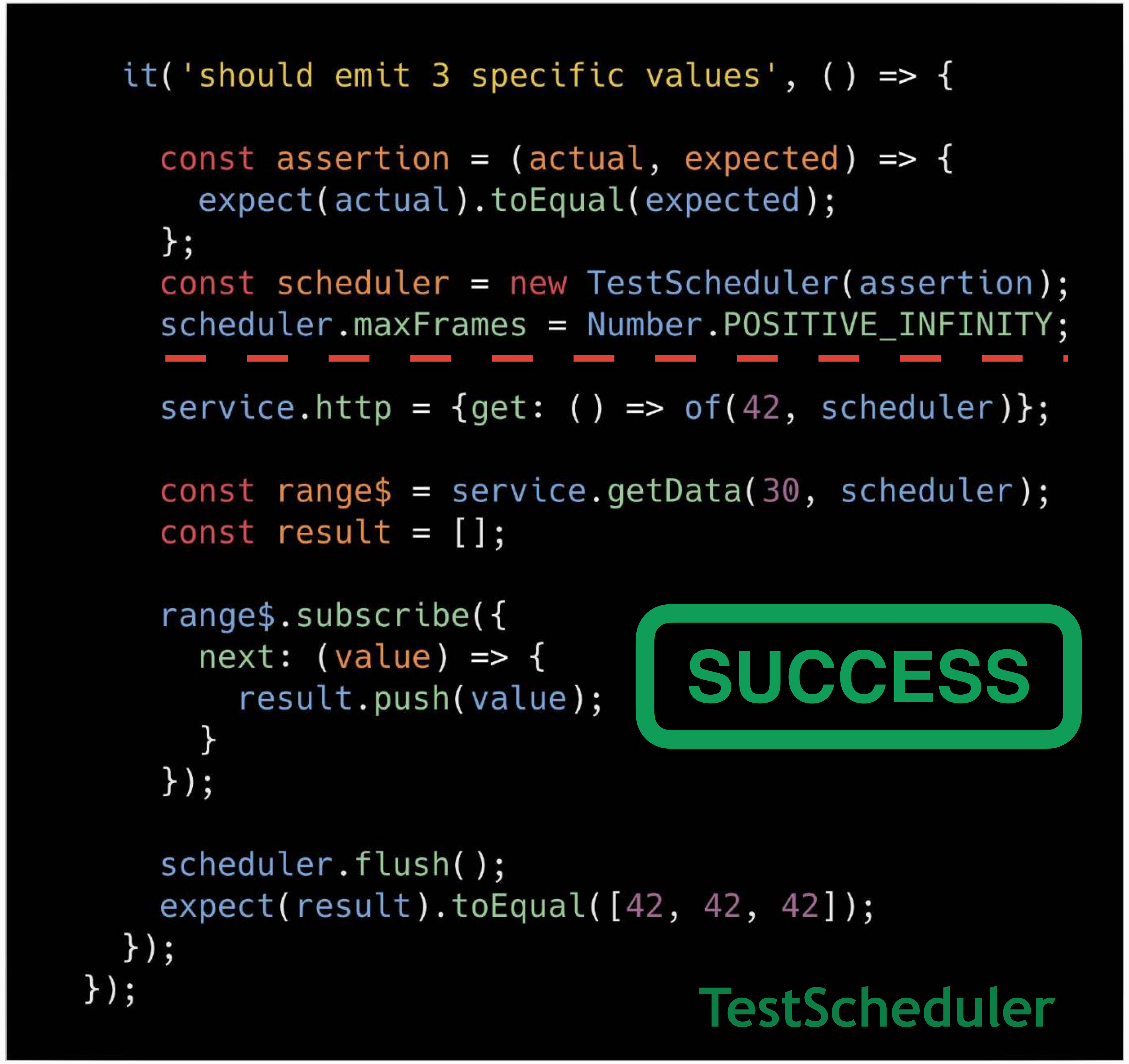
Example #2: VirtualTimeScheduler and TestScheduler



```
it('should emit 3 specific values', () => {
  const scheduler = new VirtualTimeScheduler();
  service.http = {get: () => of(42, scheduler)};

  const range$ = service.getData(30, scheduler);
  const result = [];

  range$.subscribe({
    next: (value) => {
      result.push(value);
    }
  });

  scheduler.flush();
  expect(result).toEqual([42, 42, 42]);
});
```

SUCCESS

VirtualTimeScheduler



```
  it('should emit 3 specific values', () => {

    const assertion = (actual, expected) => {
      expect(actual).toEqual(expected);
    };
    const scheduler = new TestScheduler(assertion);
    scheduler.maxFrames = Number.POSITIVE_INFINITY;

    service.http = {get: () => of(42, scheduler)};

    const range$ = service.getData(30, scheduler);
    const result = [];

    range$.subscribe({
      next: (value) => {
        result.push(value);
      }
    });

    scheduler.flush();
    expect(result).toEqual([42, 42, 42]);
  });
});
```

SUCCESS

TestScheduler

Method #2 drawbacks

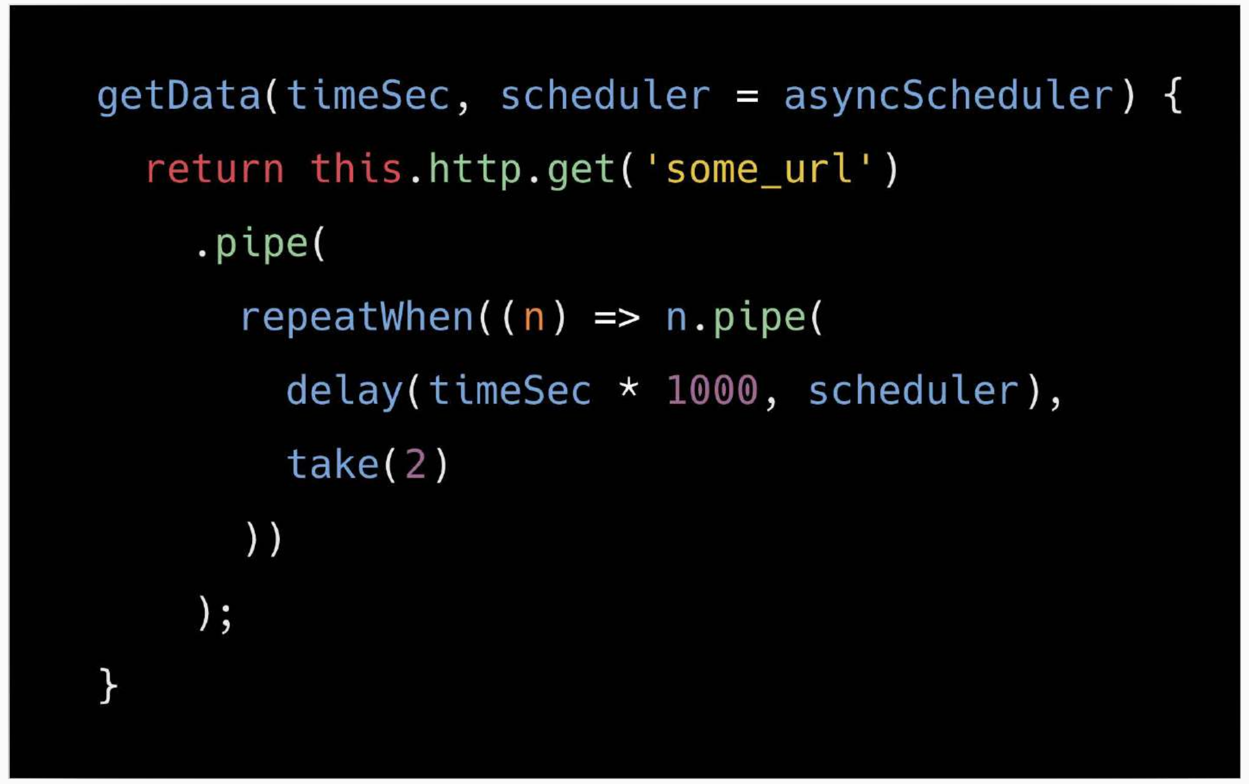
✗

1. Not visual
2. Only final result is checked

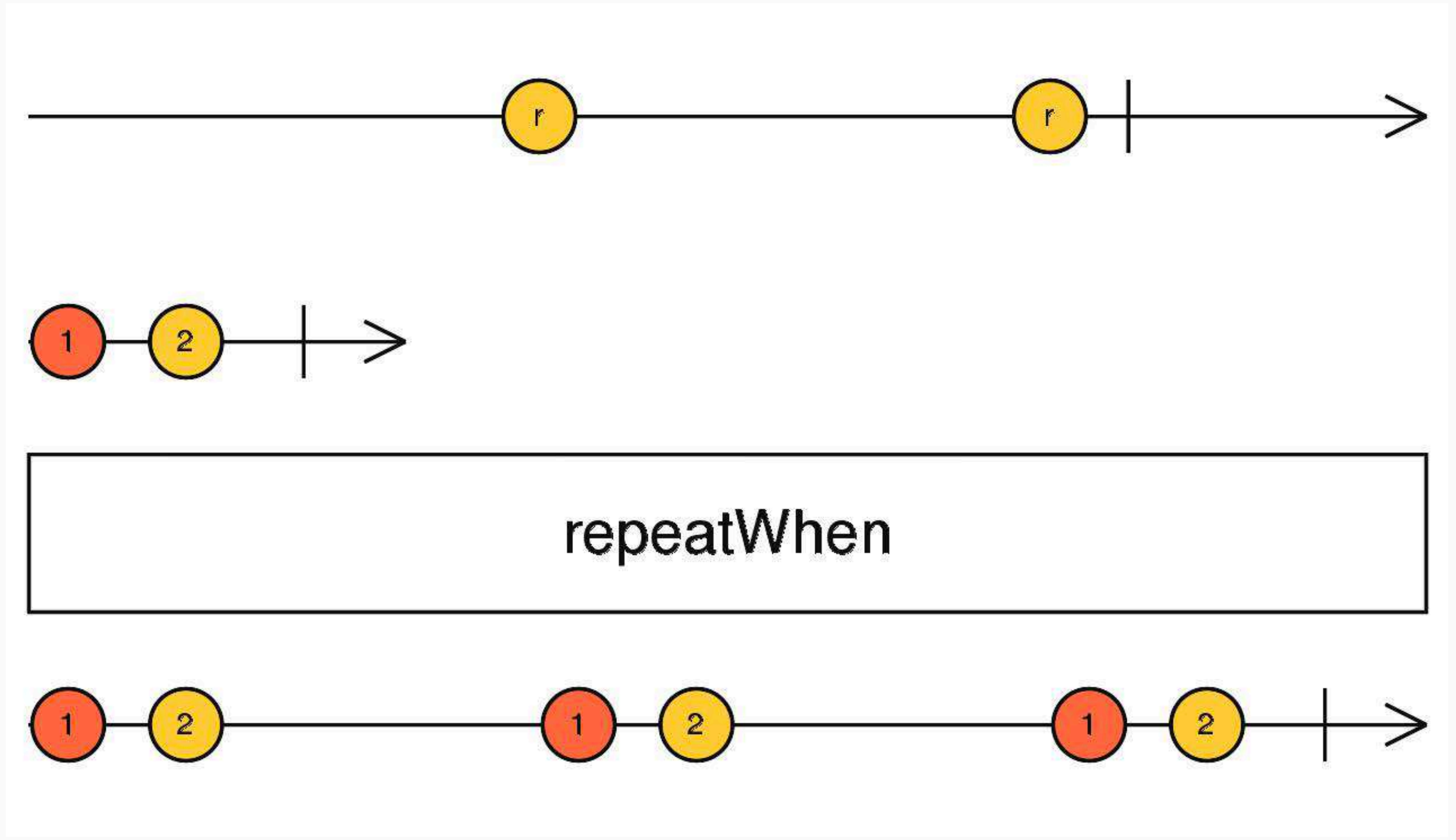
RxJS unit testing

#3. Marbles

Method #3 - Marbles



```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

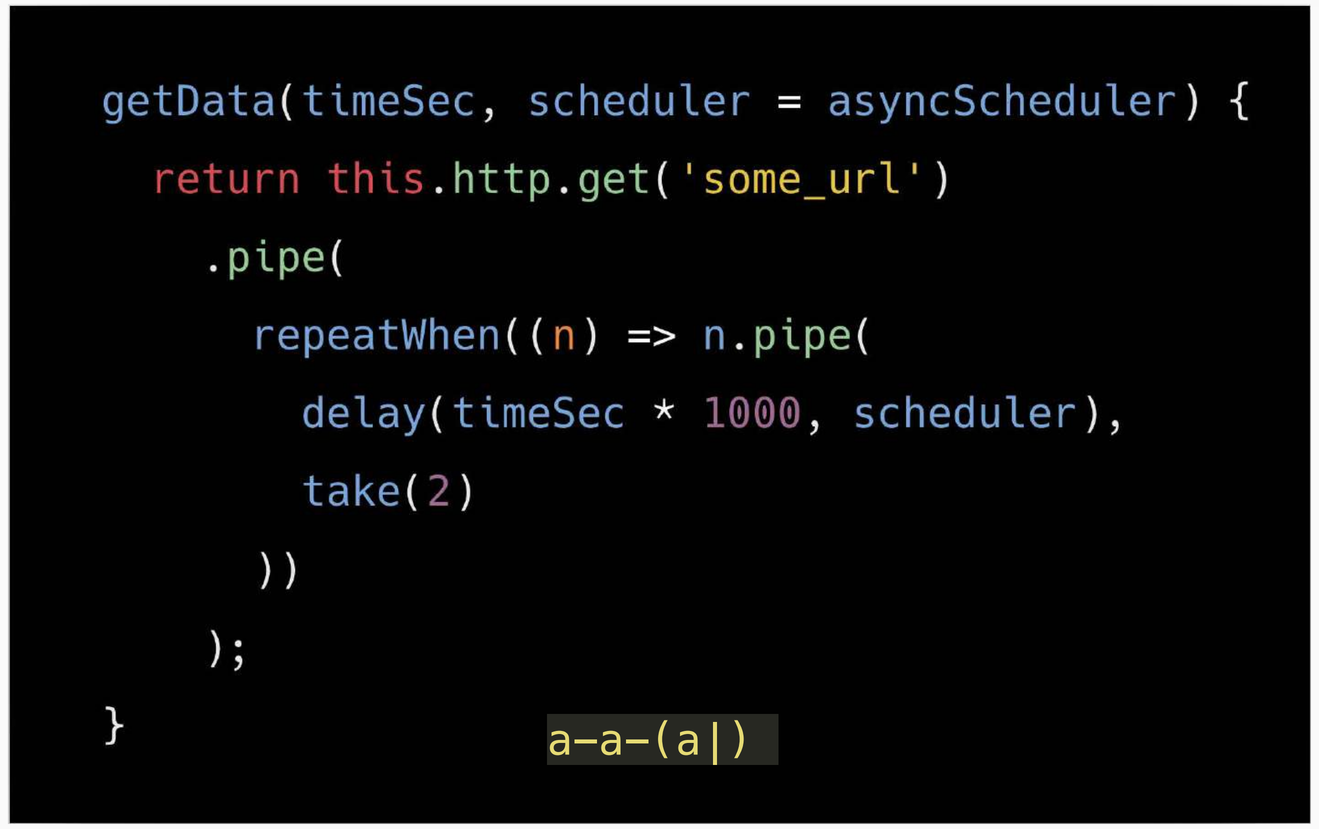


Marble Diagrams are visual representation for events emitted over the time.

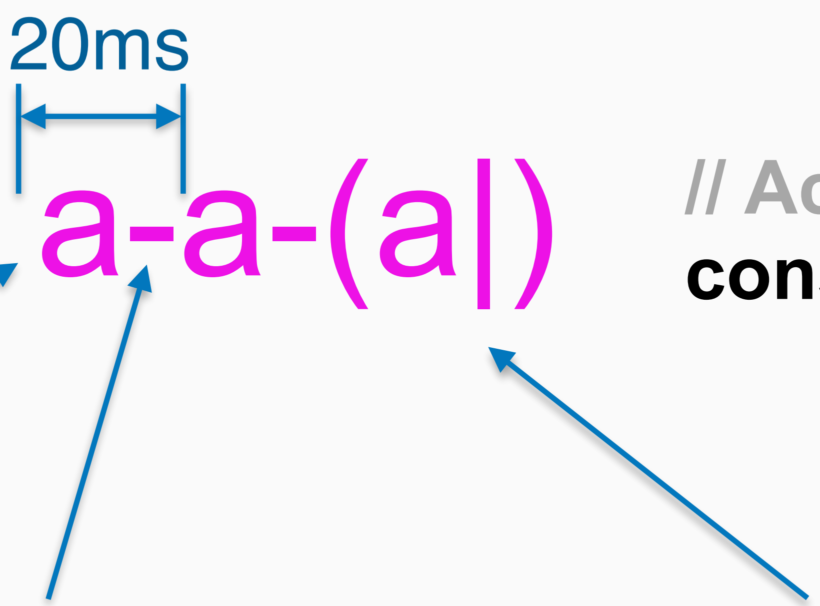
RxJS unit testing

#3. Marbles
a) TestScheduler

How marbles look like for example #2



```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}                     a-a-(a|)
```



// Actually 750ms - not 750 frames
const defaultMaxFrame: number = 750;

Value emission frame (10ms)

'-' Delay frame: 10ms

Value and completion in same frame

Example #1: TestScheduler in 4 simple steps

Code

```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

Test

```
it('should emit 3 values', () => {
  const assertion = (actual, expected) => {
    expect(actual).toEqual(expected);
  };                                                  1
  const scheduler = new TestScheduler(assertion);
  (asyncScheduler.constructor as any).delegate = scheduler;
                                                           2
  const marbleValues = {a: 42};
  service.http = {get: () => scheduler.createColdObservable('(a|)', marbleValues)};

  const expectedMarble = 'a-a-(a|)';             3

  scheduler.expectObservable(service.getData(0.01)).toBe(expectedMarble, marbleValues);

  scheduler.flush();         4
});
```

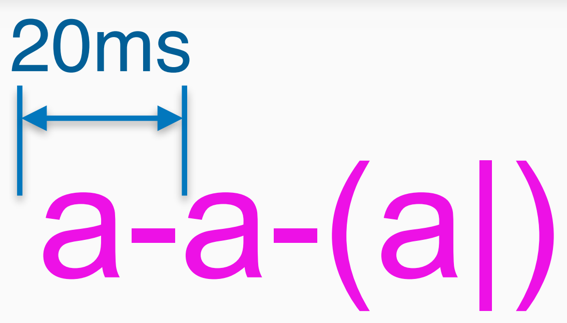
Marble explanation: mocking

```
const marbleValues = {a: 42};
service.http = {get: () =>
scheduler.createColdObservable('(a|)', marbleValues)};
```

Emitted values mapping object

Creates Observable from marble string

Marble explanation: assertion expression



Small delay value
since frame = 10ms only

```
const expectedMarble = 'a-a-(a|)';
scheduler.expectObservable(service.getData(0.02))
                              .toBe(expectedMarble, marbleValues);
```

How to create marbles (syntax)

-	**(dash)**: simulate the **passage of time**, one dash correspond to a frame
a-z	**(a to z)**: represent value **emission**, value provided with mapping object
|	**(pipe)**: emit a **completed** (end of the stream)
#	**(pound sign)**: indicate an **error** (end of the stream)
( )	**(parenthesis)**: **multiple values together in the same unit of time**
^	**(caret)**: indicate a **subscription point**
!	**(exclamation point)**: indicate the **end of a subscription point**

https://github.com/ReactiveX/rxjs/blob/master/doc/marble-testing.md

TestScheduler methods

createColdObservable: creates a "hot" observable (like a subject) that will behave as though it's already "running" when the test begins.

createHotObservable: creates a "cold" observable whose subscription starts when the test begins.

expectObservable: schedules an assertion for when the TestScheduler flushes.

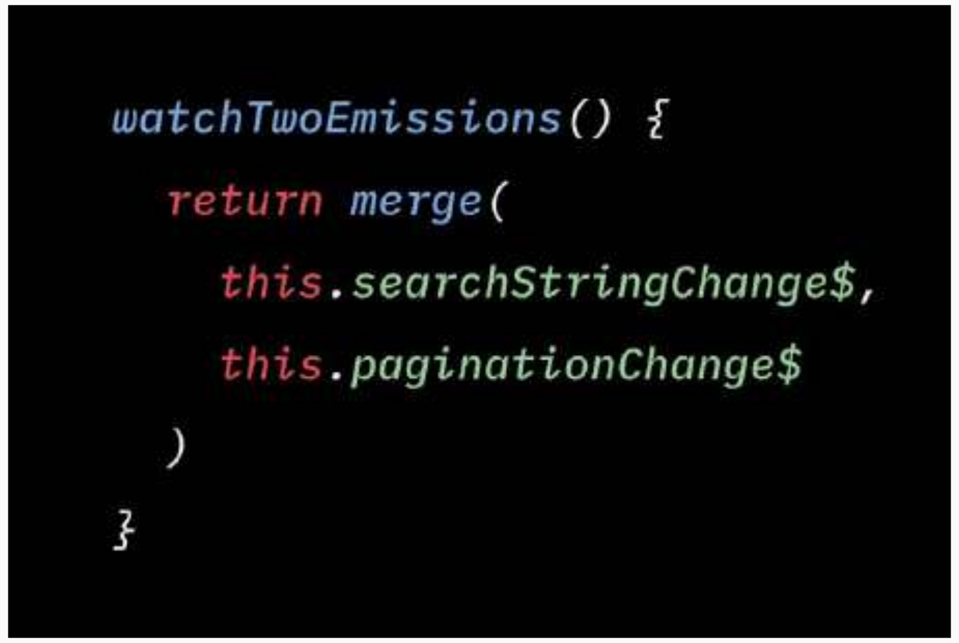
flush: immediately starts virtual time (flushing AsyncScheduler queue)

More to go: createTime, expectSubscriptions, ...

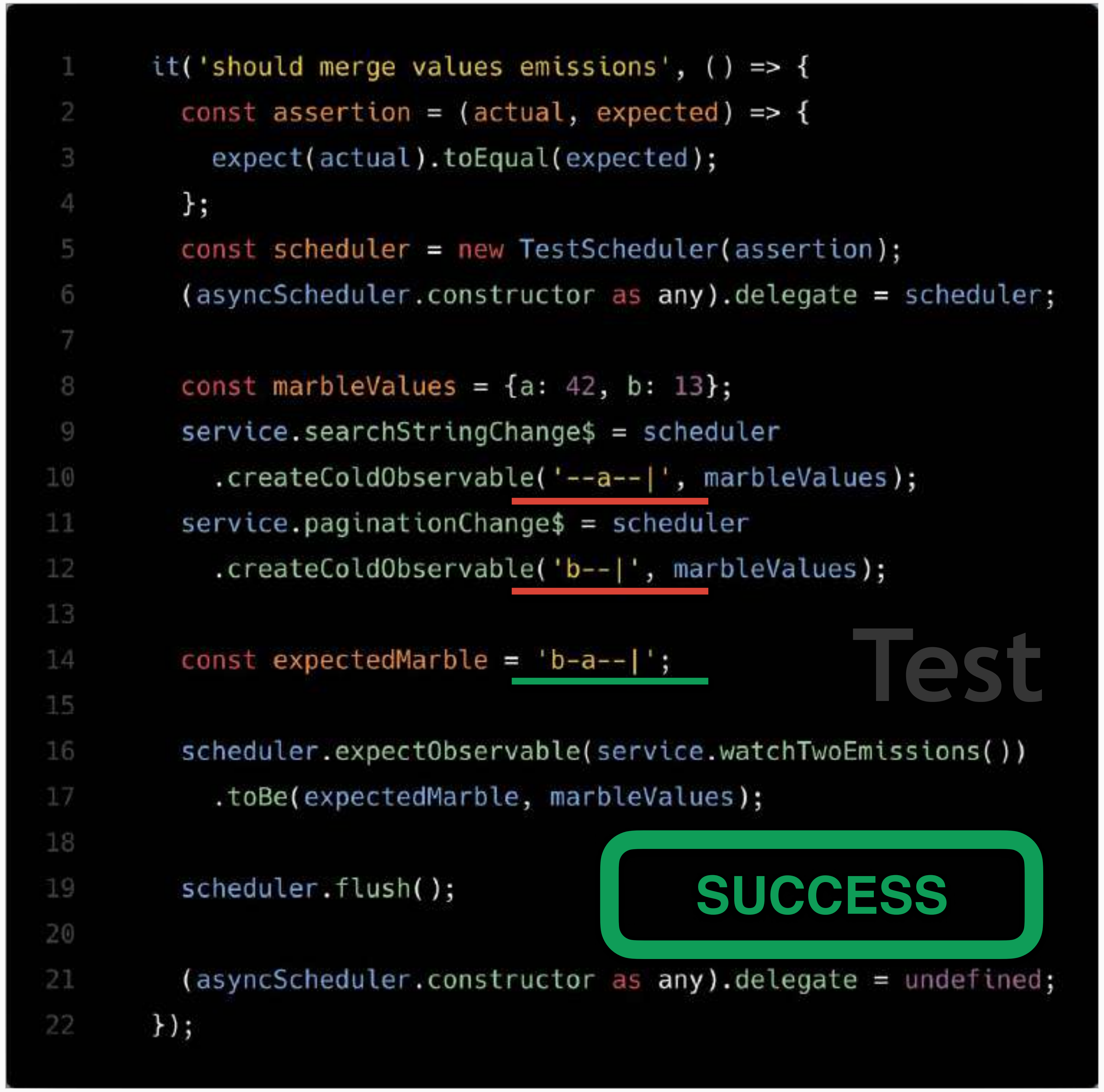
https://github.com/ReactiveX/rxjs/blob/master/doc/marble-testing.md

Example: TestScheduler

Code



```
watchTwoEmissions() {
  return merge(
    this.searchStringChange$,
    this.paginationChange$
  )
}
```



```
it('should merge values emissions', () => {
  const assertion = (actual, expected) => {
    expect(actual).toEqual(expected);
  };
  const scheduler = new TestScheduler(assertion);
  (asyncScheduler.constructor as any).delegate = scheduler;

  const marbleValues = {a: 42, b: 13};
  service.searchStringChange$ = scheduler
    .createColdObservable('--a--|', marbleValues);
  service.paginationChange$ = scheduler
    .createColdObservable('b--|', marbleValues);

  const expectedMarble = 'b-a--|';

  scheduler.expectObservable(service.watchTwoEmissions())
    .toBe(expectedMarble, marbleValues);

  scheduler.flush();

  (asyncScheduler.constructor as any).delegate = undefined;
});
```

TestScheduler pros & cons

✓

1. Visual - we test all emitted values
2. No need for additional method scheduler param

✗

1. Delay values are not prod one's
2. Demands some learning curve

RxJS unit testing

#3. Marbles

b) jasmine-marbles

- wrapper for TestScheduler

Example #1: TestScheduler in 4 simple steps

Code

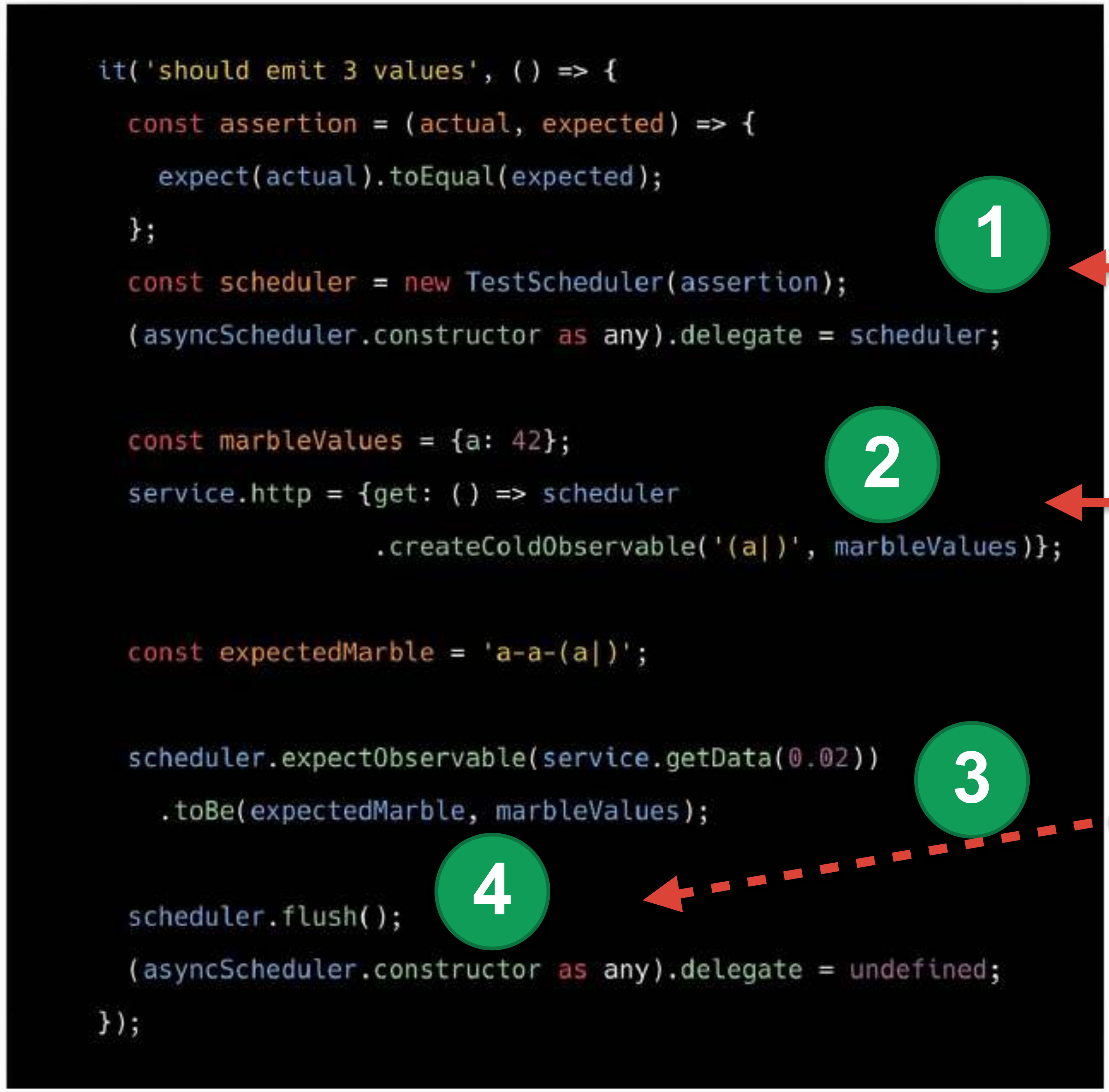
```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

Test

```
it('should emit 3 values', () => {
  const assertion = (actual, expected) => {
    expect(actual).toEqual(expected);
  };                                                              1
  const scheduler = new TestScheduler(assertion);
  (asyncScheduler.constructor as any).delegate = scheduler;
                                                                       2
  const marbleValues = {a: 42};
  service.http = {get: () => scheduler.createColdObservable('(a|)', marbleValues)};

  const expectedMarble = 'a-a-(a|)';                   3

  scheduler.expectObservable(service.getData(0.01)).toBe(expectedMarble, marbleValues);

  scheduler.flush();                         4
});
```

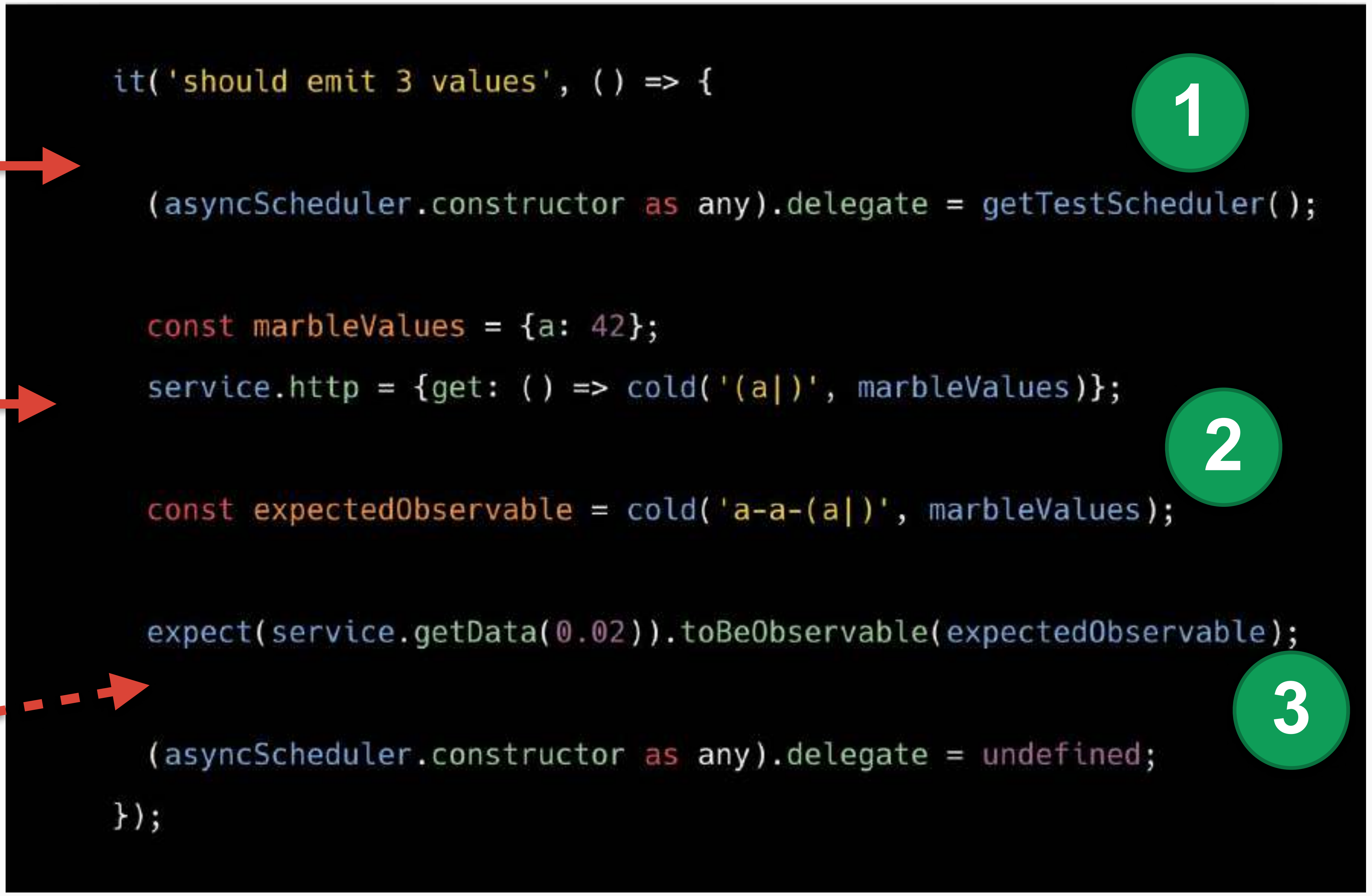
jasmine-marbles

TestScheduler



```
it('should emit 3 values', () => {
  const assertion = (actual, expected) => {
    expect(actual).toEqual(expected);
  };
  const scheduler = new TestScheduler(assertion);
  (asyncScheduler.constructor as any).delegate = scheduler;

  const marbleValues = {a: 42};
  service.http = {get: () => scheduler
                  .createColdObservable('(a|)', marbleValues)};

  const expectedMarble = 'a-a-(a|)';

  scheduler.expectObservable(service.getData(0.02))
    .toBe(expectedMarble, marbleValues);

  scheduler.flush();
  (asyncScheduler.constructor as any).delegate = undefined;
});
```

jasmine-marbles



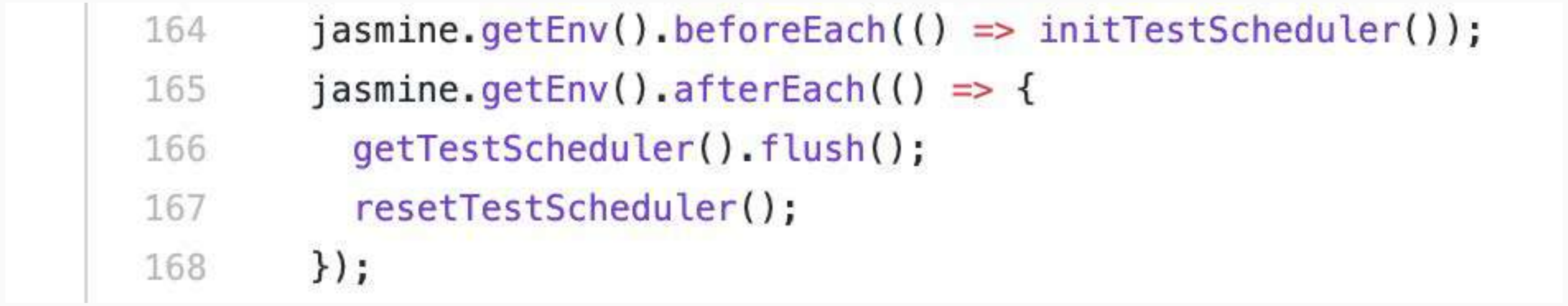
```
it('should emit 3 values', () => {

  (asyncScheduler.constructor as any).delegate = getTestScheduler();

  const marbleValues = {a: 42};
  service.http = {get: () => cold('(a|)', marbleValues)};

  const expectedObservable = cold('a-a-(a|)', marbleValues);

  expect(service.getData(0.02)).toBeObservable(expectedObservable);

  (asyncScheduler.constructor as any).delegate = undefined;
});
```

TestScheduler methods vs jasmine-marbles methods

TestScheduler	jasmine-marbles
createColdObservable	cold
createHotObservable	hot
expectObservable(…).toBe(…)	expect(..).toBeObservable(…)
flush()	<run implicitly>

jasmine-marbles

https://github.com/synapse-wireless-labs/jasmine-marbles/blob/.../index.ts



```
jasmine.getEnv().beforeEach(() => initTestScheduler());
jasmine.getEnv().afterEach(() => {
  getTestScheduler().flush();
  resetTestScheduler();
});
```

Method #3 TestScheduler pros & cons

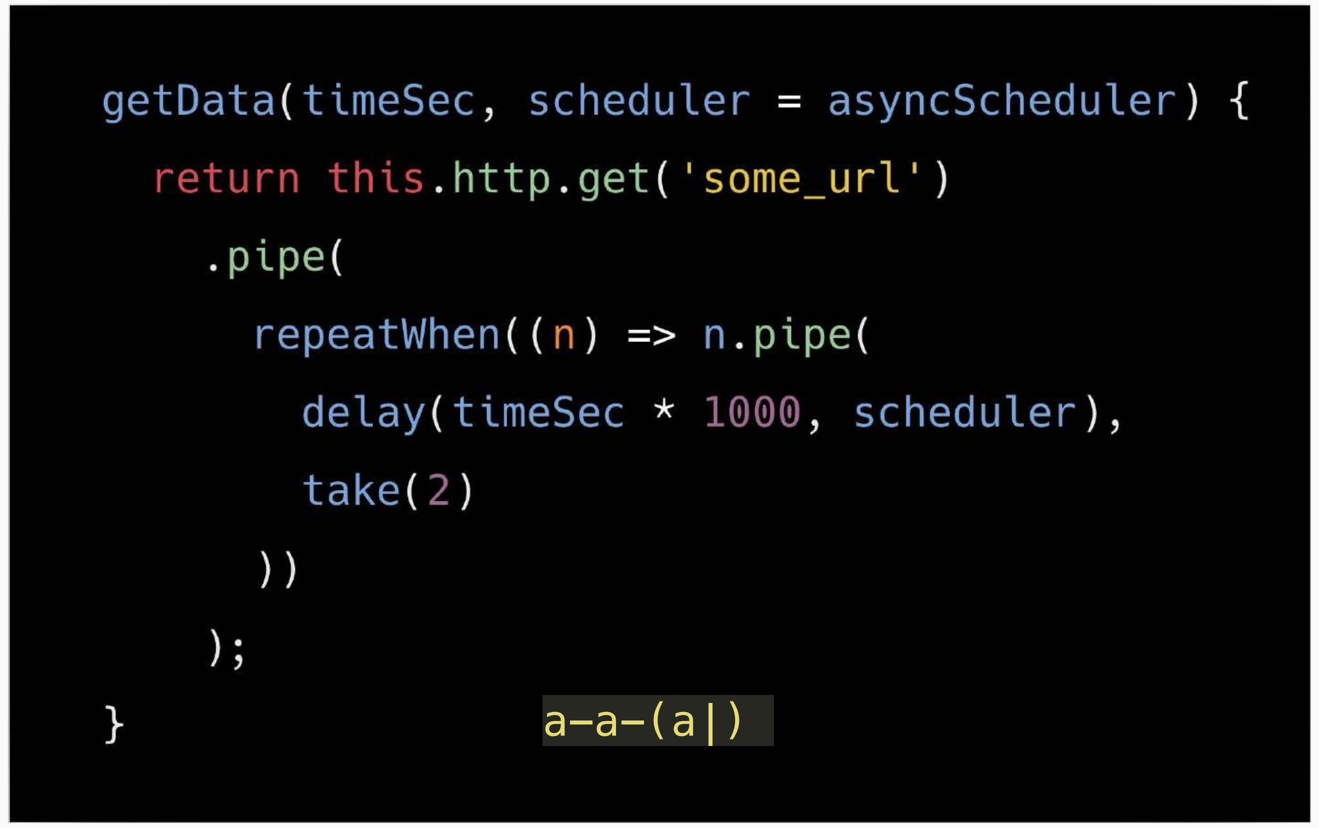
✓

1. Visual - we test all emitted values
2. No need for additional method scheduler param
3. TestScheduler flush() method is called implicitly

✗

1. Delay values are not prod one's
2. Demands some learning curve

How marbles look like for example #2



```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}                     a-a-(a|)
```

a-a-(a|)

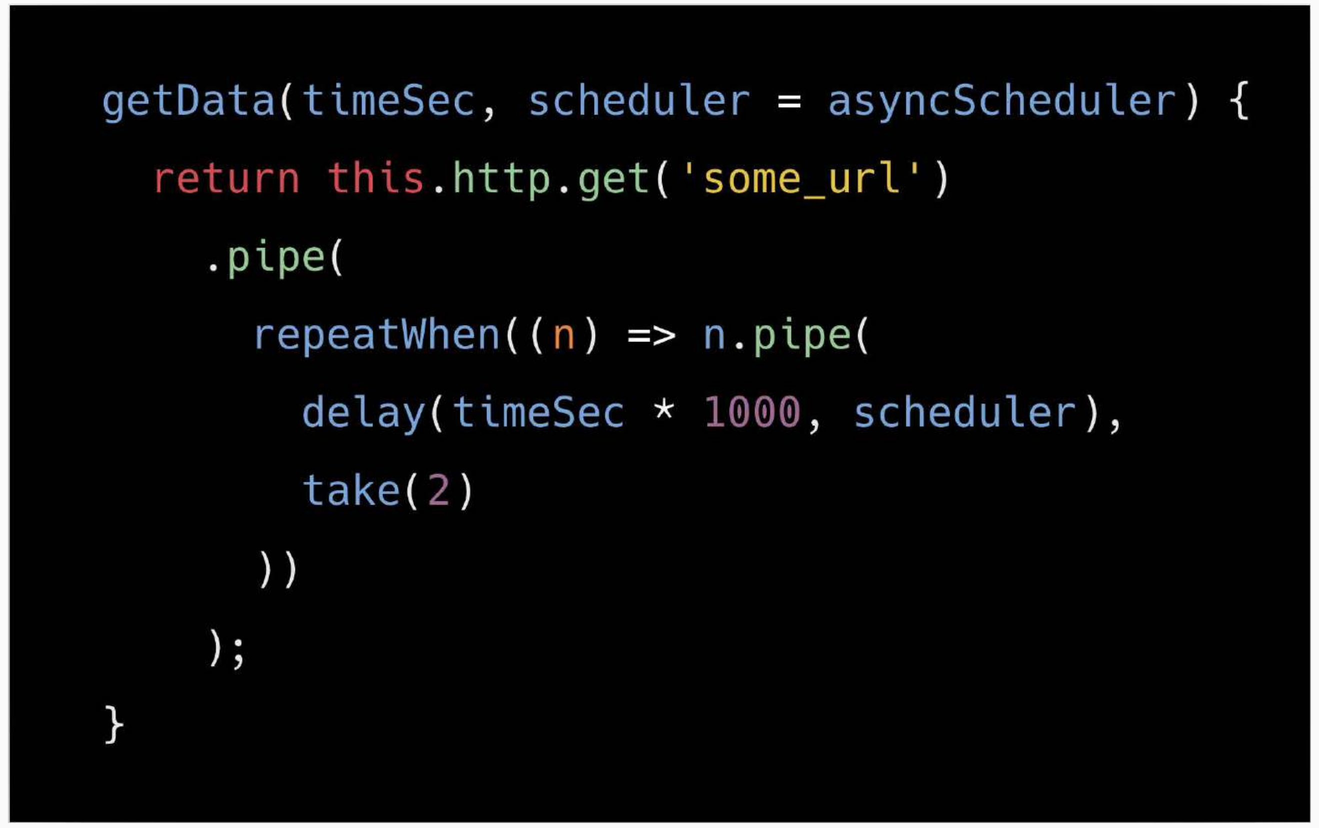
a 1000ms a 1000ms (a|)

RxJS unit testing

#3. Marbles

c) TestScheduler.run (v6+)

How marbles look like for example #2

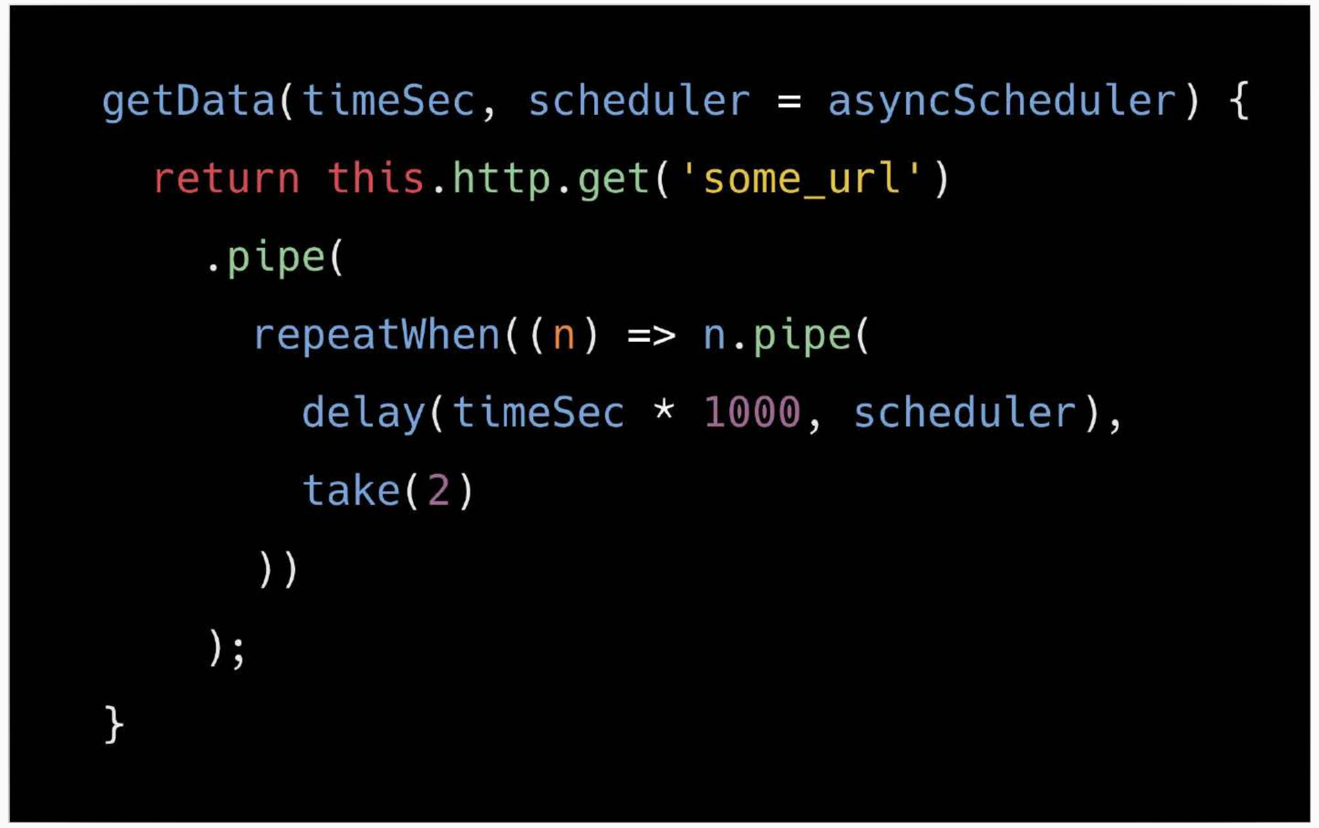


```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

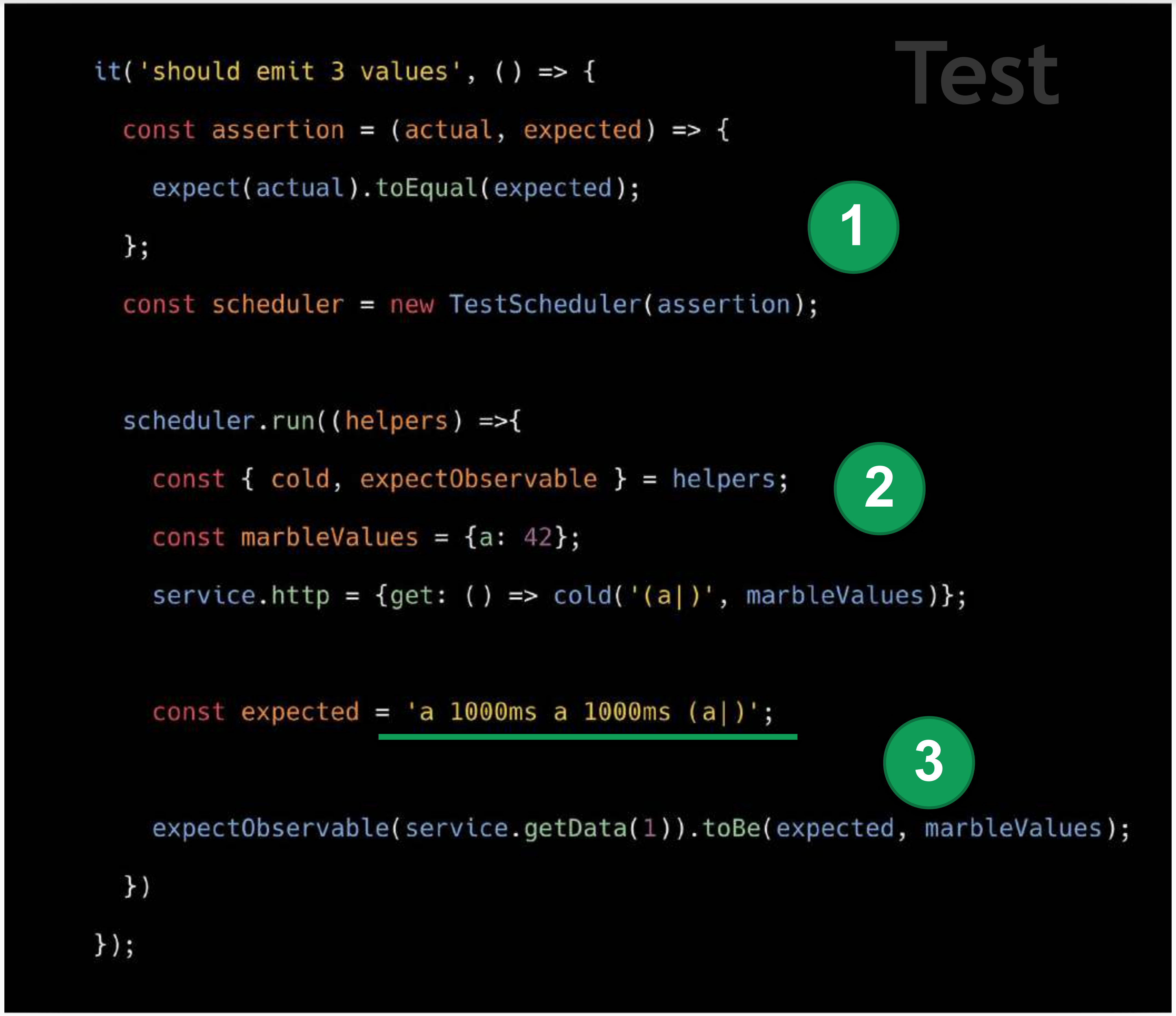
const expectedMarble = 'a 1000ms a 1000ms (a|)';

Example #2: TestScheduler.run method

Code



```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```



Test

```
it('should emit 3 values', () => {
  const assertion = (actual, expected) => {
    expect(actual).toEqual(expected);
  };
  const scheduler = new TestScheduler(assertion);

  scheduler.run((helpers) =>{
    const { cold, expectObservable } = helpers;
    const marbleValues = {a: 42};
    service.http = {get: () => cold('(a|)', marbleValues)};

    const expected = 'a 1000ms a 1000ms (a|)';

    expectObservable(service.getData(1)).toBe(expected, marbleValues);
  })
});
```

1

2

3

TestScheduler vs TestScheduler.run

TestScheduler	TestScheduler.run
createColdObservable	cold
createHotObservable	hot
expectObservable(…).toBe(…)	expectObservable(…).toBe(…)
flush()	<run implicitly>
AsyncScheeduler.delegate	<applied implicitly>
Frame length is 10 virtual milliseconds	Frame length is 1 virtual millisecond
No time progressive syntax, only ‘—a———b-’	Time progressive syntax available: ‘-a 1000ms b-|’

TestScheduler.run usage

```
it('name', () => {

    const assertion = (actual, expected) => {
          expect(actual).toEqual(expected);
    };

    const scheduler = new TestScheduler(assertion);

   testScheduler.run(helpers => {

     const { cold, hot, expectObservable, expectSubscriptions, flush } = helpers;

     // some tests

   });
})
```

Example #2: TestScheduler.run method

Code

```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

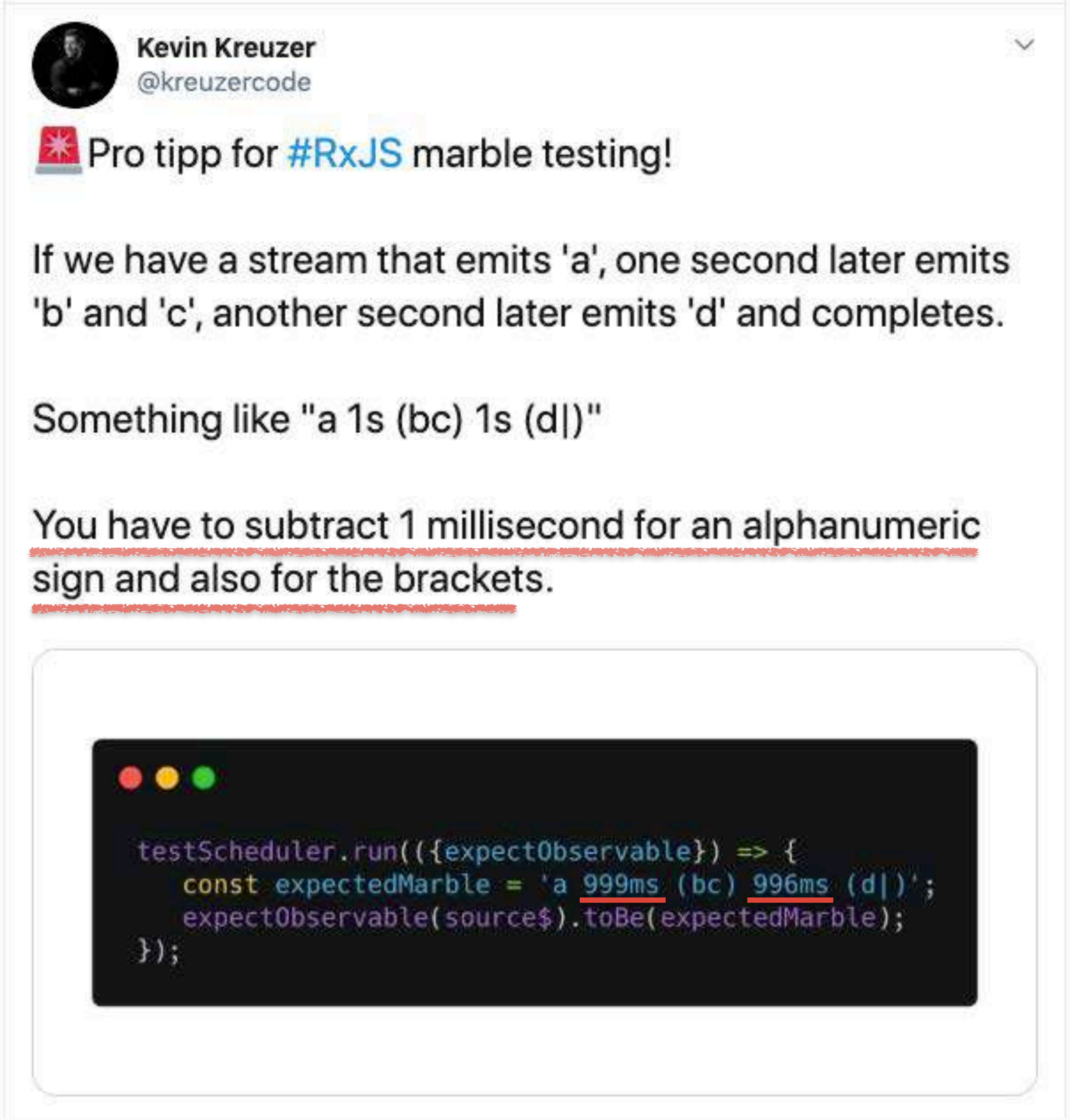
Test

```
it('should emit 3 values', () => {
  const assertion = (actual, expected) => {
    expect(actual).toEqual(expected);
  };
  const scheduler = new TestScheduler(assertion);

  scheduler.run((helpers) =>{
    const { cold, expectObservable } = helpers;
    const marbleValues = {a: 42};
    service.http = {get: () => cold('(a|)', marbleValues)};

    const expected = 'a 1000ms a 1000ms (a|)';

    expectObservable(service.getData(1)).toBe(expected, marbleValues);
  })
});
```

TestScheduler.run nuance

```
✖ should emit 3 values
  Chrome 76.0.3809 (Mac OS X 10.14.6)
Error: Expected $[1].frame = 1000 to equal 1001.
Expected $[2].frame = 2000 to equal 2002.
Expected $[3].frame = 2000 to equal 2002.
    at <Jasmine>
```

TestScheduler.run nuance



Kevin Kreuzer
@kreuzercode

🚨 Pro tipp for #RxJS marble testing!

If we have a stream that emits 'a', one second later emits 'b' and 'c', another second later emits 'd' and completes.

Something like "a 1s (bc) 1s (d|)"

You have to subtract 1 millisecond for an alphanumeric sign and also for the brackets.

```
testScheduler.run(({expectObservable}) => {
   const expectedMarble = 'a 999ms (bc) 996ms (d|)';
   expectObservable(source$).toBe(expectedMarble);
});
```

Expected Marble

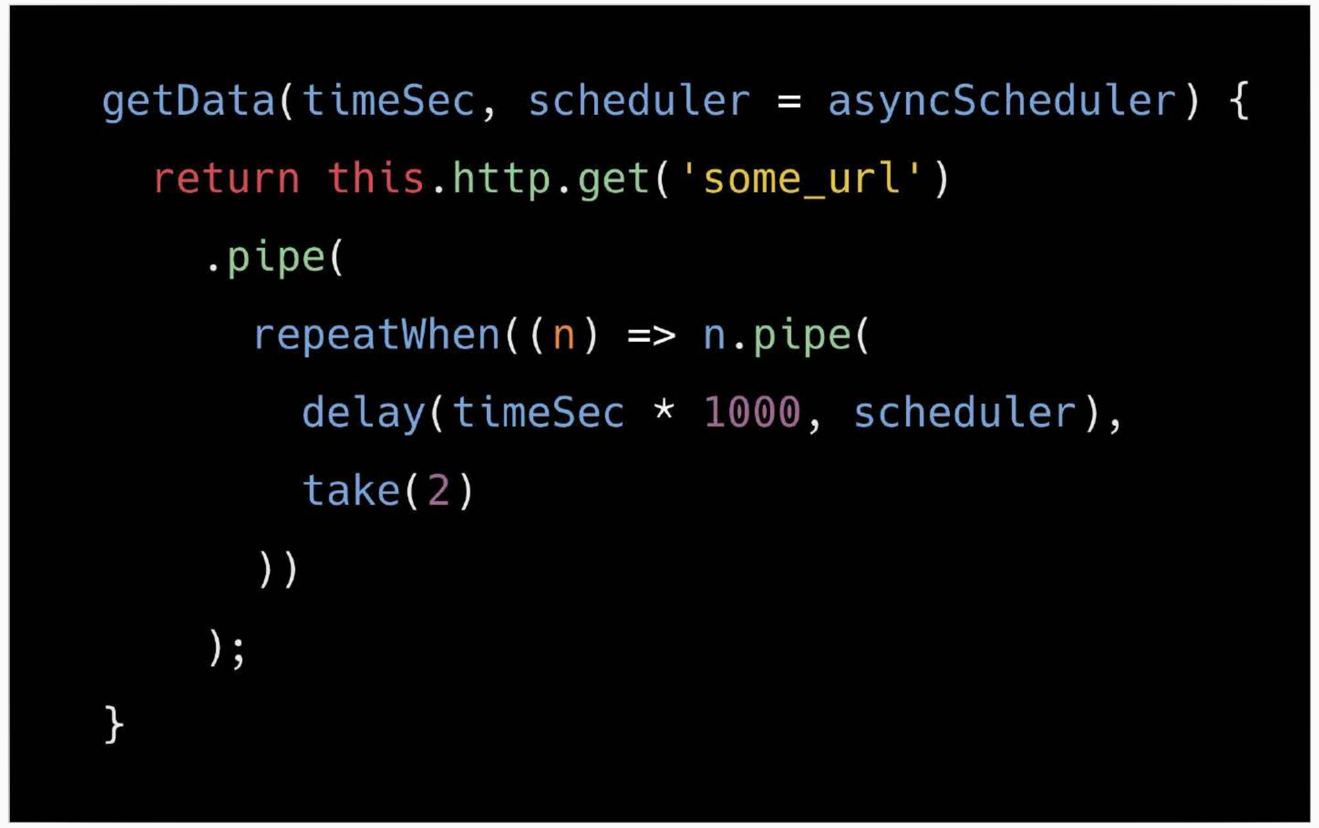
~~'a 1000ms a 1000ms (a|)'~~

-1ms -1ms

'a 999ms a 999ms (a|)'

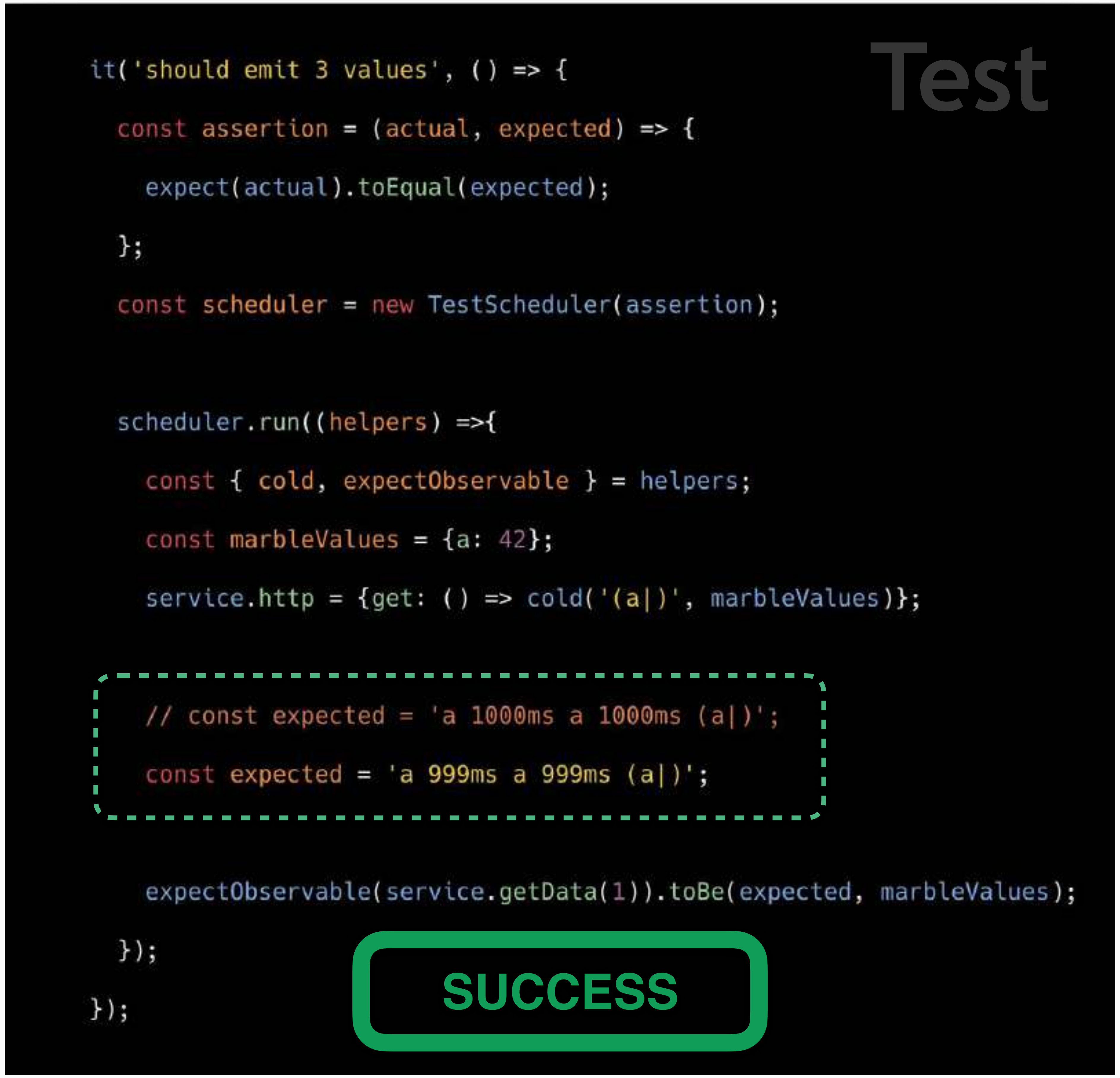
Example #2 with corrected timings

Code



```
getData(timeSec, scheduler = asyncScheduler) {
  return this.http.get('some_url')
    .pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000, scheduler),
        take(2)
      ))
    );
}
```

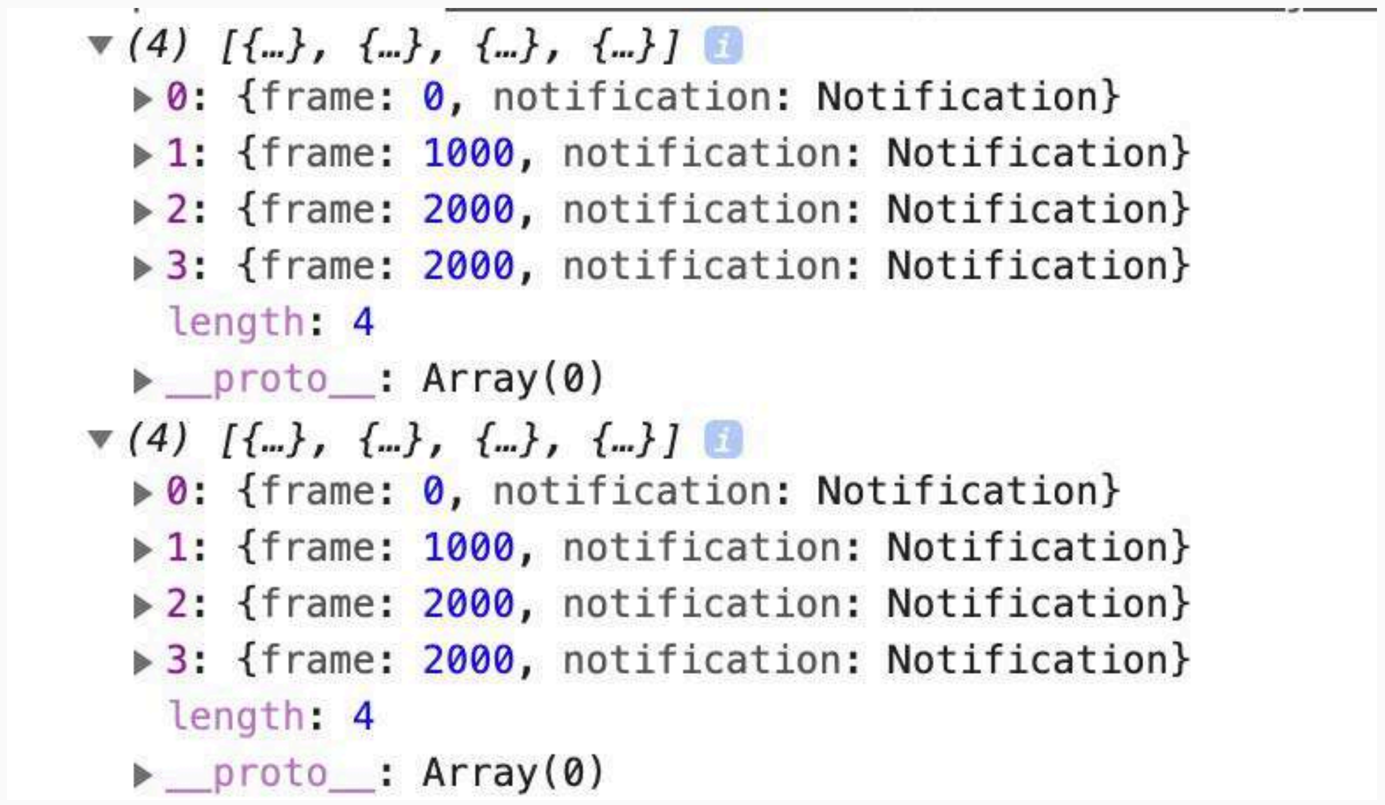
Test



```
it('should emit 3 values', () => {
  const assertion = (actual, expected) => {
    expect(actual).toEqual(expected);
  };
  const scheduler = new TestScheduler(assertion);

  scheduler.run((helpers) =>{
    const { cold, expectObservable } = helpers;
    const marbleValues = {a: 42};
    service.http = {get: () => cold('(a|)', marbleValues)};

    // const expected = 'a 1000ms a 1000ms (a|)';
    const expected = 'a 999ms a 999ms (a|)';

    expectObservable(service.getData(1)).toBe(expected, marbleValues);
  });
});
```

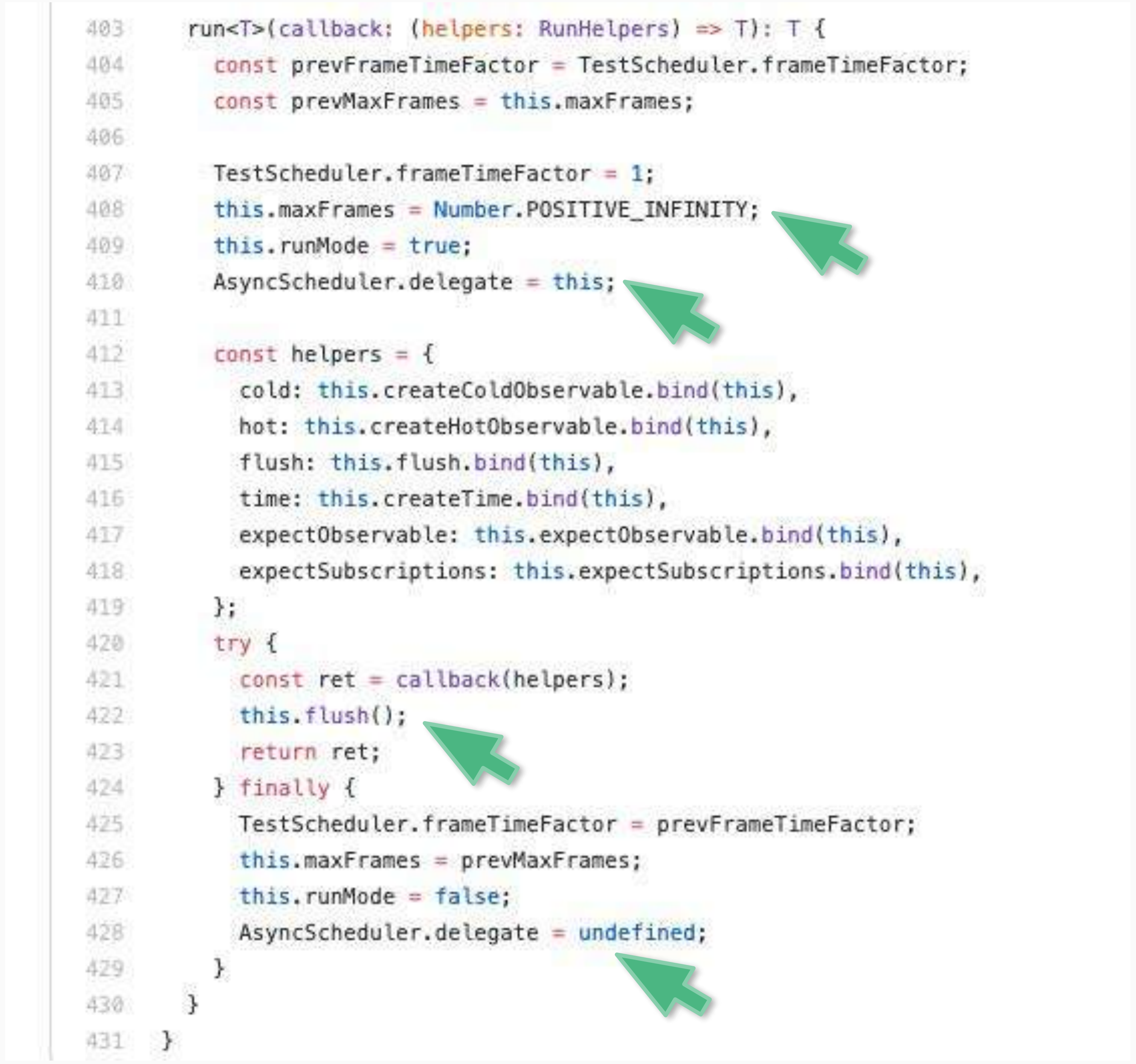
TestScheduler.run nuance - how to handle

```
const assertion = (actual, expected) => {

  console.log(expected, actual); // to check timings
  expect(actual).toEqual(expected);
};

const scheduler = new TestScheduler(assertion);

scheduler.run((helpers) =>{
   …
})
```



```
▼(4) [{…}, {…}, {…}, {…}]
  ▶0: {frame: 0, notification: Notification}
  ▶1: {frame: 1000, notification: Notification}
  ▶2: {frame: 2000, notification: Notification}
  ▶3: {frame: 2000, notification: Notification}
   length: 4
  ▶__proto__: Array(0)
▼(4) [{…}, {…}, {…}, {…}]
  ▶0: {frame: 0, notification: Notification}
  ▶1: {frame: 1000, notification: Notification}
  ▶2: {frame: 2000, notification: Notification}
  ▶3: {frame: 2000, notification: Notification}
   length: 4
  ▶__proto__: Array(0)
```

TestScheduler.run under-the-hood



```
  run<T>(callback: (helpers: RunHelpers) => T): T {
    const prevFrameTimeFactor = TestScheduler.frameTimeFactor;
    const prevMaxFrames = this.maxFrames;

    TestScheduler.frameTimeFactor = 1;
    this.maxFrames = Number.POSITIVE_INFINITY;
    this.runMode = true;
    AsyncScheduler.delegate = this;

    const helpers = {
      cold: this.createColdObservable.bind(this),
      hot: this.createHotObservable.bind(this),
      flush: this.flush.bind(this),
      time: this.createTime.bind(this),
      expectObservable: this.expectObservable.bind(this),
      expectSubscriptions: this.expectSubscriptions.bind(this),
    };
    try {
      const ret = callback(helpers);
      this.flush();
      return ret;
    } finally {
      TestScheduler.frameTimeFactor = prevFrameTimeFactor;
      this.maxFrames = prevMaxFrames;
      this.runMode = false;
      AsyncScheduler.delegate = undefined;
    }
  }
}
```

https://github.com/ReactiveX/rxjs/blob/master/src/internal/testing/TestScheduler.ts

TestScheduler.run pros and cons

✓

1. Visual - we test all emitted values
2. Prod timings value with convenient progressive timings syntax (‘a 999ms a …')
3. AsyncScheduler.delegate trick is applied implicitly
4. flush() is also called implicitly
5. Not tied to any testing framework

✗

1. Nuances with timings calculations
2. Demands some learning curve

RxJS unit testing

#3. Marbles
d) rxjs-marbles

rxjs-marbles vs jasmine-marbles

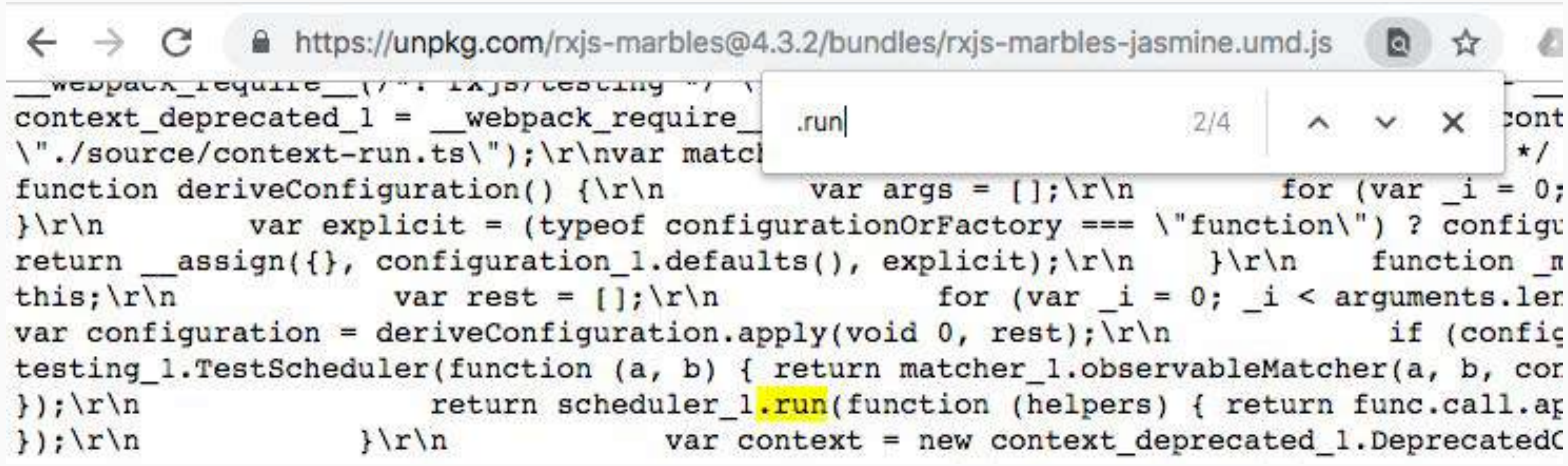
jasmine-marbles wraps TestScheduler

rxjs-marbles wraps TestScheduler.run

But not only…

rxjs-marbles vs jasmine-marbles

https://unpkg.com/rxjs-marbles@4.3.2/bundles/rxjs-marbles-jasmine.umd.js

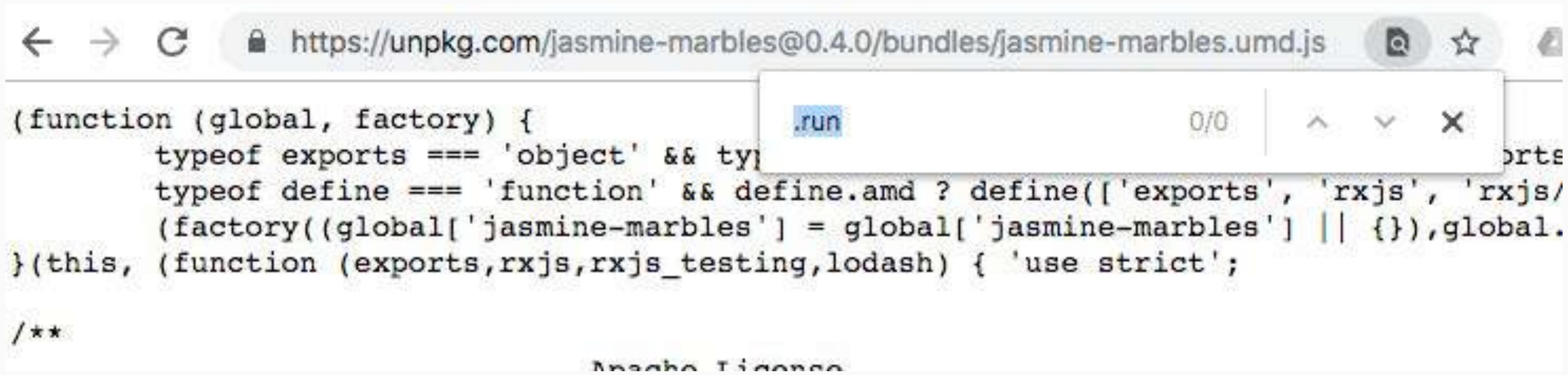


https://unpkg.com/rxjs-marbles@4.3.2/bundles/rxjs-marbles-jasmine.umd.js

.run 2/4

```
__webpack_require__(/*! rxjs/testing */ \
context_deprecated_1 = __webpack_require_                                              cont
\"./source/context-run.ts\");\r\nvar matcl                                              */
function deriveConfiguration() {\r\n        var args = [];\r\n        for (var _i = 0;
}\r\n        var explicit = (typeof configurationOrFactory === \"function\") ? configu
return __assign({}, configuration_1.defaults(), explicit);\r\n    }\r\n    function _m
this;\r\n            var rest = [];\r\n            for (var _i = 0; _i < arguments.ler
var configuration = deriveConfiguration.apply(void 0, rest);\r\n            if (config
testing_1.TestScheduler(function (a, b) { return matcher_1.observableMatcher(a, b, cor
});\r\n                return scheduler_1.run(function (helpers) { return func.call.ap
});\r\n          }\r\n            var context = new context_deprecated_1.DeprecatedC
```

https://unpkg.com/jasmine-marbles@0.4.0/bundles/jasmine-marbles.umd.js



https://unpkg.com/jasmine-marbles@0.4.0/bundles/jasmine-marbles.umd.js

.run 0/0

```
(function (global, factory) {
        typeof exports === 'object' && ty                                          orts
        typeof define === 'function' && define.amd ? define(['exports', 'rxjs', 'rxjs/
        (factory((global['jasmine-marbles'] = global['jasmine-marbles'] || {}),global.
}(this, (function (exports,rxjs,rxjs_testing,lodash) { 'use strict';

/**
                              Apache License
```

Rxjs-marbles

1. **Testing framework agnostic** (supports jasmine, mocha, jest)

2. **Many examples in git repo:** github.com/cartant/rxjs-marbles

3. **Except standard functionality it has specific helpers:**

   a) **cases** - allows to apply different marble input data-sets for tests

   b) **observe** - Observable wrapper for async code unit tests with 'done' callback

Example #1 with rxjs-marles

Code

```
getData(timeSec) {
  return this.http.get('some_url').pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000),
        take(2)
      ))
    );
}
```

```
it('should emit 3 values', marbles((m) => {
    const marbleValues = {a: 42};
    service.http = {get: () => m.cold('(a|)', marbleValues)};

    // const expected = 'a 1000ms a 1000ms (a|)';
    const expected = 'a 999ms a 999ms (a|)';

    m.expect(service.getData(1))
      .toBeObservable(expected, marbleValues);

  })
);
```

Test

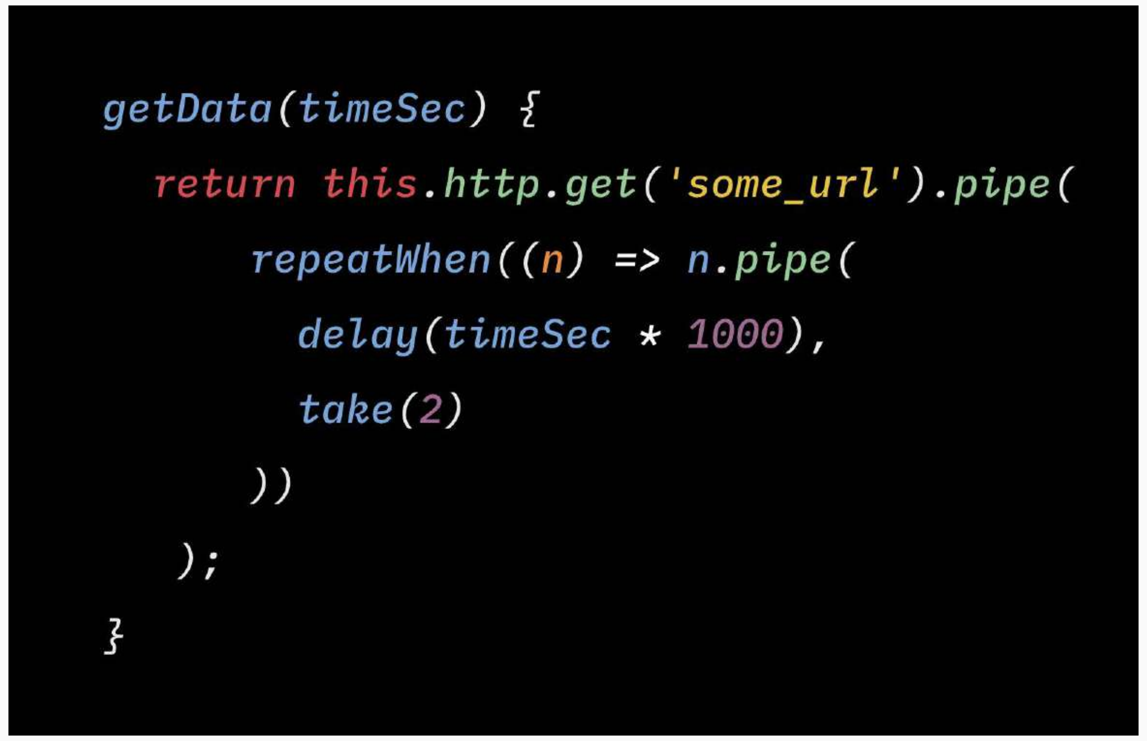
rxjs-marles supports different testing frameworks

```
import { marbles } from “rxjs-marbles/jest”;

import { marbles } from “rxjs-marbles/jasmine";

import { marbles } from "rxjs-marbles/mocha";
```

https://github.com/cartant/rxjs-marbles/tree/master/examples

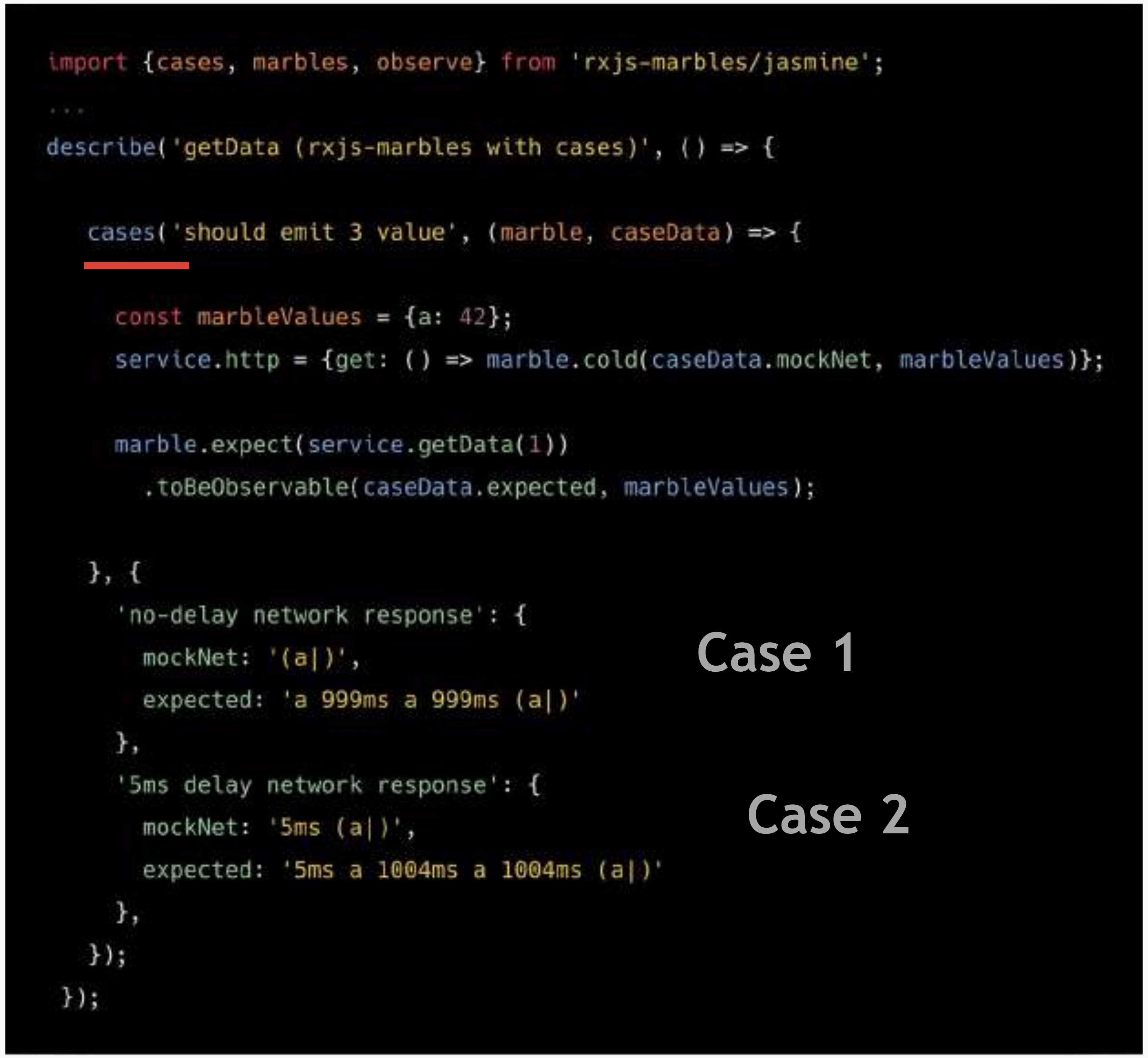
rxjs-marles - cases

Code



```
getData(timeSec) {
  return this.http.get('some_url').pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000),
        take(2)
      ))
    );
}
```

Test



```
import {cases, marbles, observe} from 'rxjs-marbles/jasmine';
...
describe('getData (rxjs-marbles with cases)', () => {

  cases('should emit 3 value', (marble, caseData) => {

    const marbleValues = {a: 42};
    service.http = {get: () => marble.cold(caseData.mockNet, marbleValues)};

    marble.expect(service.getData(1))
      .toBeObservable(caseData.expected, marbleValues);

  }, {
    'no-delay network response': {
      mockNet: '(a|)',
      expected: 'a 999ms a 999ms (a|)'
    },
    '5ms delay network response': {
      mockNet: '5ms (a|)',
      expected: '5ms a 1004ms a 1004ms (a|)'
    },
  });
});
```

Case 1

Case 2

rxjs-marles - observe

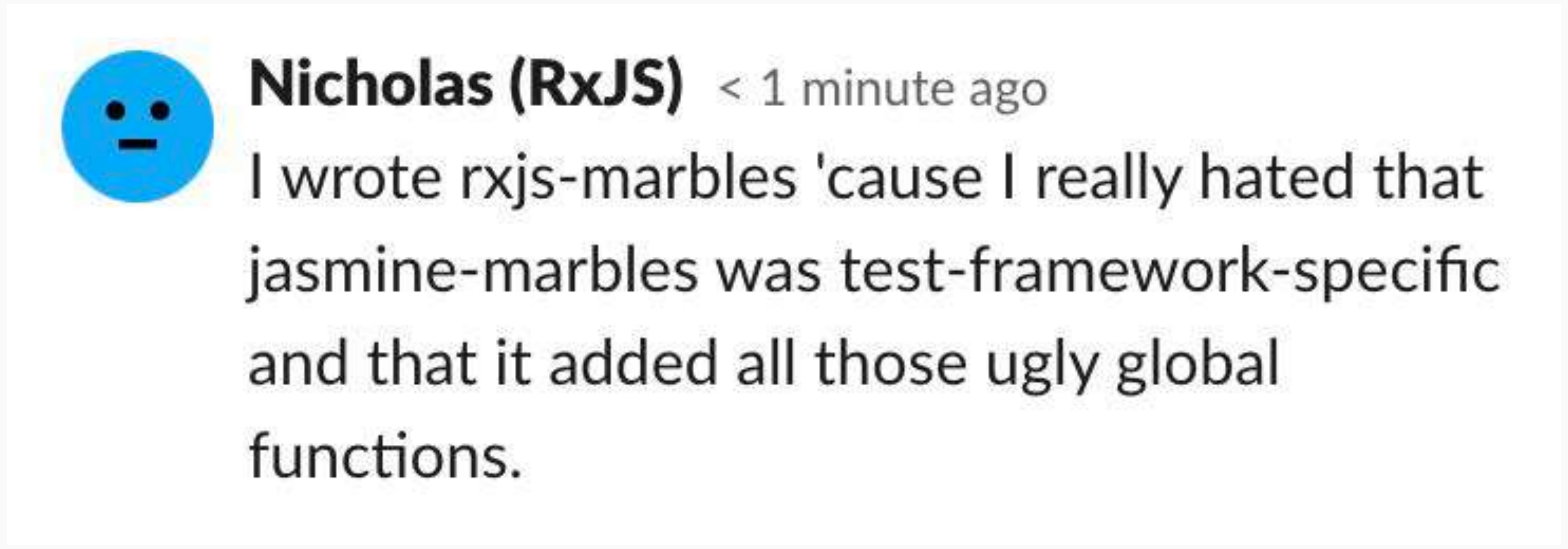
Code

```
getData(timeSec) {
  return this.http.get('some_url').pipe(
      repeatWhen((n) => n.pipe(
        delay(timeSec * 1000),
        take(2)
      ))
    );
}
```

Test

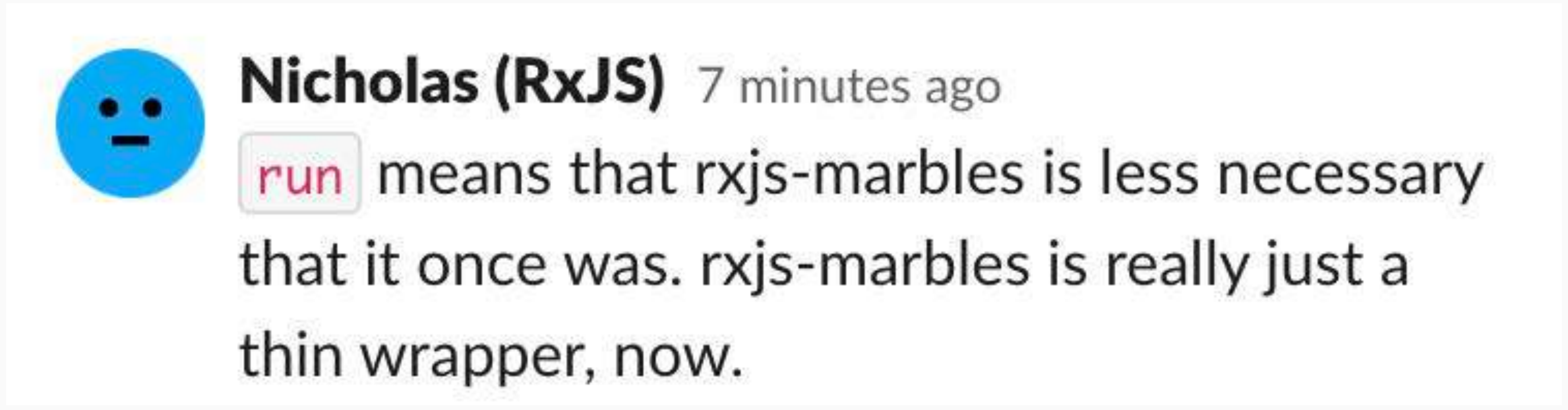
```
import {cases, marbles, observe} from 'rxjs-marbles/jasmine';
...
 it('should call this.http.get twice and get result twice',
   observe(() => {

     service.http = {get: () => of(42, asyncScheduler)};

     return service.getData(0.01)
       .pipe(
         toArray(),
         tap((result) => expect(result).toEqual([42, 42, 42]))
       );
   })
 );
```

rxjs-marbles vs TestScheduler.run



Nicholas (RxJS) < 1 minute ago

I wrote rxjs-marbles 'cause I really hated that jasmine-marbles was test-framework-specific and that it added all those ugly global functions.



Nicholas (RxJS) 7 minutes ago

`run` means that rxjs-marbles is less necessary that it once was. rxjs-marbles is really just a thin wrapper, now.

rxjs-mabrles pros and cons

✓

1. Visual - we test all emitted values

2. Prod timings value with convenient progressive timings syntax (‘a 999ms a …')

2. No scheduler param in method (AsyncScheduler.delegate trick is applied implicitly)

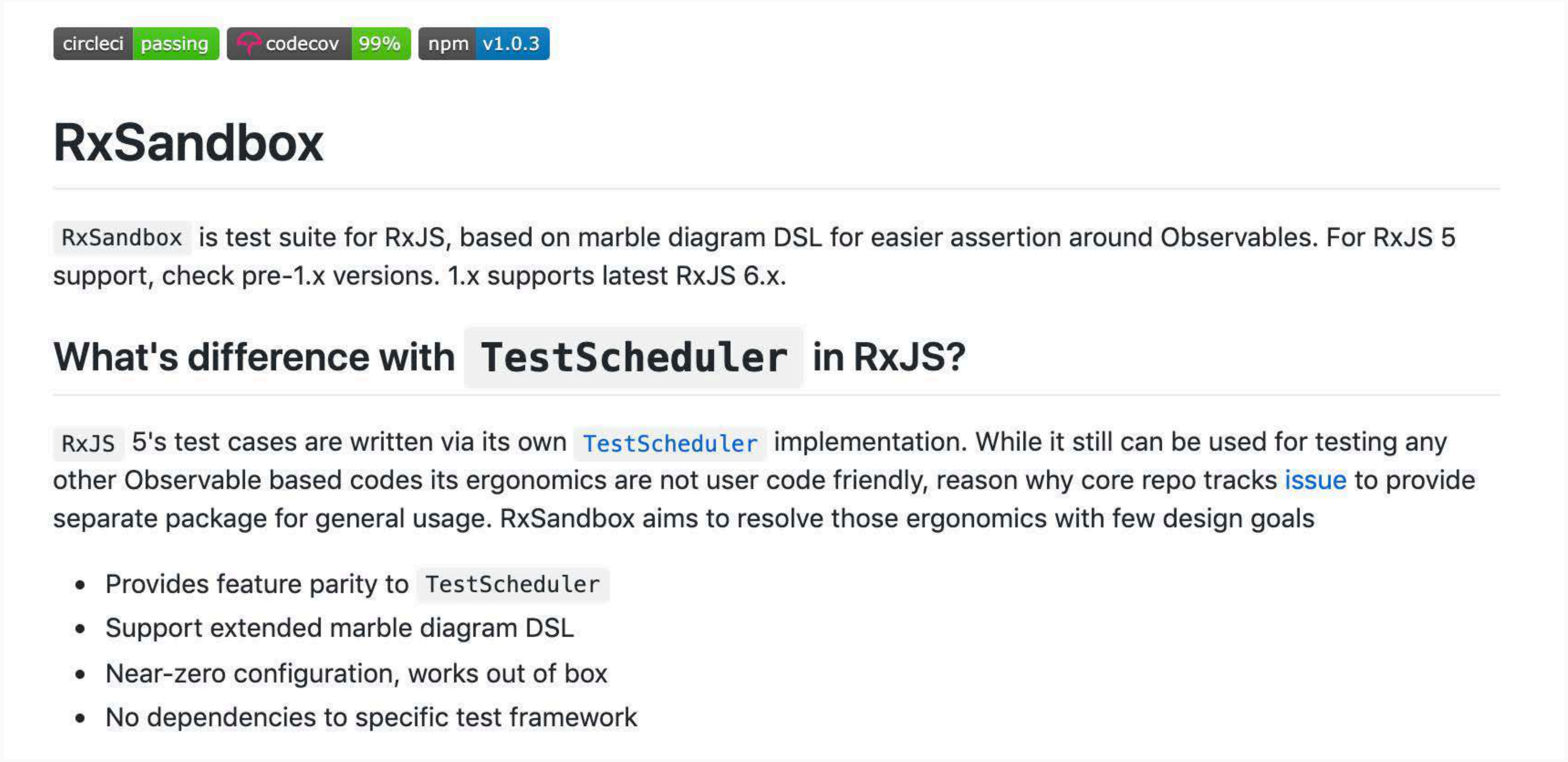
4. flush() is also called implicitly

✗

1. Nuances with timings calculations

2. Demands some learning curve

Another TestScheduler/TestScheduler.run wrapper - Rx-sandbox



RxSandbox

`RxSandbox` is test suite for RxJS, based on marble diagram DSL for easier assertion around Observables. For RxJS 5 support, check pre-1.x versions. 1.x supports latest RxJS 6.x.

What's difference with `TestScheduler` in RxJS?

`RxJS` 5's test cases are written via its own `TestScheduler` implementation. While it still can be used for testing any other Observable based codes its ergonomics are not user code friendly, reason why core repo tracks issue to provide separate package for general usage. RxSandbox aims to resolve those ergonomics with few design goals

- Provides feature parity to `TestScheduler`
- Support extended marble diagram DSL
- Near-zero configuration, works out of box
- No dependencies to specific test framework

https://github.com/kwonoj/rx-sandbox

Links

Official Rx.JS manual
https://github.com/ReactiveX/rxjs/blob/master/doc/marble-testing.md

Many marble-testing examples for all RxJS operators
https://github.com/ReactiveX/rxjs/blob/master/spec/operators/

jasmine-marbles
https://github.com/synapse-wireless-labs/jasmine-marbles

rxjs-marbles
https://github.com/cartant/rxjs-marbles

rxSandbox
https://github.com/kwonoj/rx-sandbox

Special thanks

Nicholas Jamieson
RxJS Core Team Member

@ncjamieson

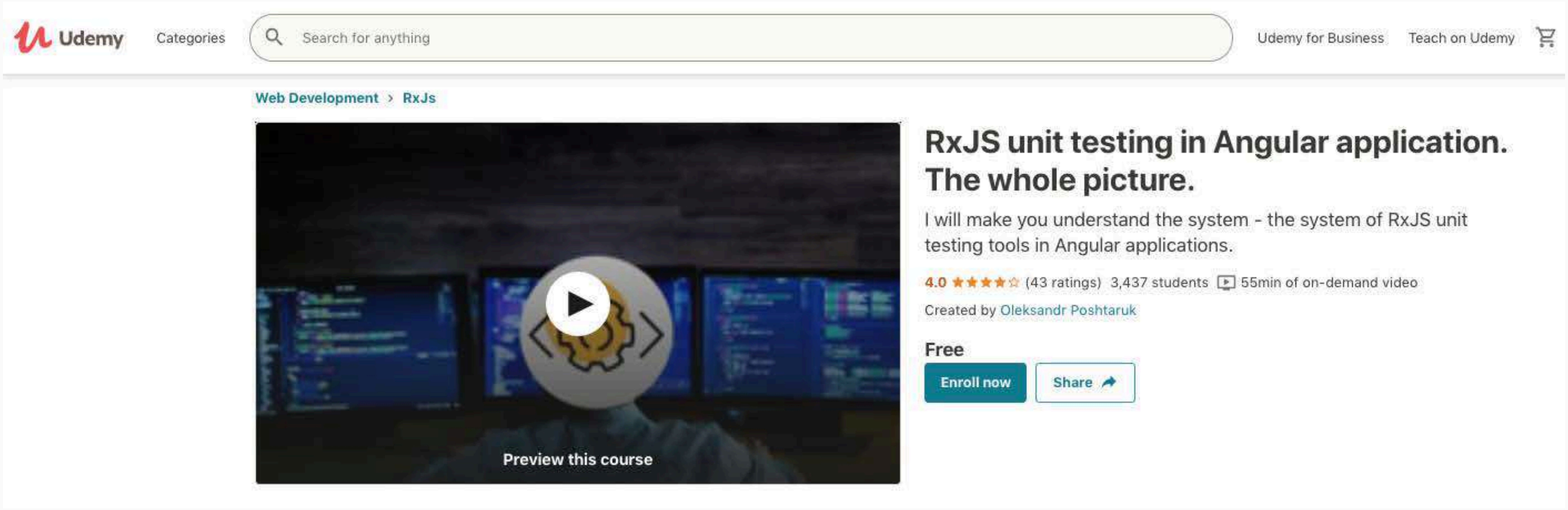
rxjs-marbles
rxjs-spy-devtools

Kevin Kreuzer
#Javascript enthusiast

@kreuzercode

Angular In Depth

Free video course



Web Development › RxJs

RxJS unit testing in Angular application. The whole picture.

I will make you understand the system - the system of RxJS unit testing tools in Angular applications.

4.0 ★★★★☆ (43 ratings) 3,437 students 55min of on-demand video

Created by Oleksandr Poshtaruk

Free

THANK YOU!

Q&A ?

Oleksandr Poshtaruk

Twitter: @El_extremal

dev.to: dev.to/oleksandr

medium: medium.com/@alexanderposhtaruk

Free video-course on RxJS Unit-testing: https://bit.ly/2rqwpqO

“Angular can waste you time” on Youtube