




Front End Development 1

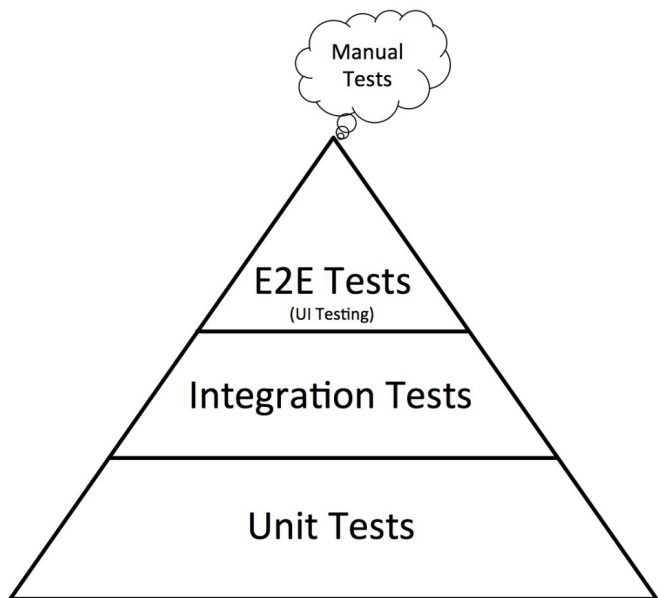
Sesi 27





Angular ⁺ Unit Testing

Software Testing



Membuat testing pada aplikasi, membantu mengecek apakah aplikasi berjalan sesuai dengan ekspektasi.

Terdapat 3 macam level testing:

1. **Unit tests:** Mengetes bagian-bagian terkecil dari kode. (low level testing)
2. **Integration tests:** Mengetes integrasi antara bagian-bagian kode tersebut.
3. **End-to-End (e2e) tests:** Mengetes proses dari awal sampai akhir. (High level testing)

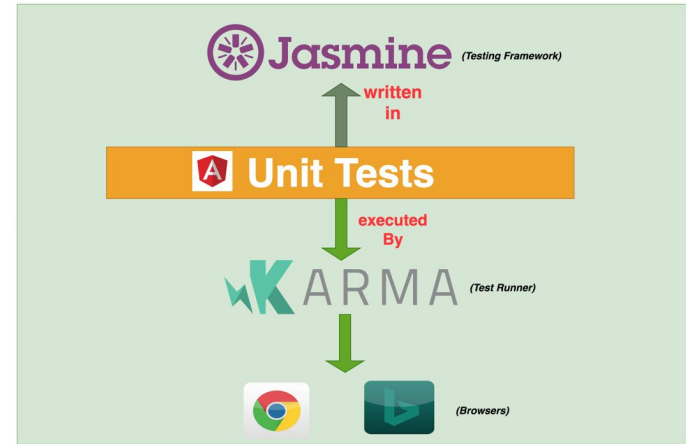
Pada sesi ini akan berfokus pada pembuatan **unit test**.

Pengenalan Jasmine & Karma

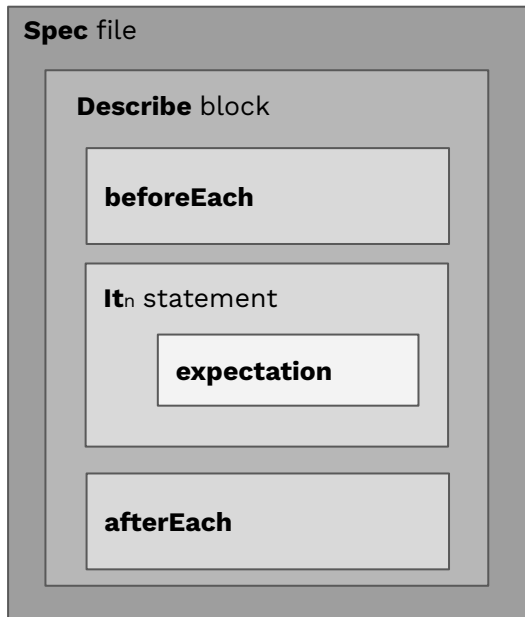
Jasmine (<https://jasmine.github.io/>) merupakan open-source testing framework yang dalam angular sudah di-instal via Angular CLI. Menyediakan beberapa function untuk menuliskan test.

Karma (<https://karma-runner.github.io/latest/index.html>) merupakan tool untuk menjalankan test. Mempunyai fitur **live-reload** dimana dapat me-refresh browser/aplikasi secara otomatis apabila kita melakukan perubahan pada aplikasi.

Jasmine dan Karma sudah diinstal oleh Angular CLI sebagai default tools test



Anatomi Test Jasmine



Test case akan berada di file yang berekstensi **.spec**.

- **describe** merupakan wadah yang akan berisi rangkaian test block (beforeEach, it, etc). Yang kita membantu mengklusterkan rangkaian test. Pengelompokan tergantung kebutuhan.
- **beforeEach** merupakan hooks akan dijalankan sebelum block test lainnya.
- **afterEach** merupakan hooks akan dijalankan setelah block test lainnya.
- **It** statement berisi unit test itu sendiri. (tentang apa yang di test dan apa yang diekspektasi)
- **Expectation:** rangkaian statement ekspektasi
- **Matchers:** digunakan dalam rangkaian expectation statement untuk mengkomparasi aktual value dengan ekspektasi value.



Contoh Test Suite (Jasmine)

```
describe("A suite", function() {  
  let message  
  
  beforeEach(function() {  
    message = 'hello world!'  
  })  
  
  it("should expect message to contain 'hello'", () => {  
    expect(message).toContain('hello');  
  })  
  
  it('should return the sum of two numbers', () => {  
    let actual = add(1,1);  
    let expected = 2;  
  
    expect(actual).toBe(expected);  
  })  
})
```

beforeEach dijalankan sebelum block lainnya. Biasanya digunakan untuk mempersiapkan object yang digunakan untuk test.

Expectation dari it statement ini adalah, mengecek variable message mengandung kata 'hello'.

Expectation dari it statement ini adalah, hasil return function `add(1,1)` (actual) adalah angka 2 (expected).

Matchers yang digunakan untuk komparasi adalah **toContain** dan **toBe**

List matchers lainnya: <https://devhints.io/jasmine>

Contoh test case jasmine lainnya: https://jasmine.github.io/tutorials/your_first_suite

Test Focus

- **xdescribe**: untuk menganulir rangkaian test yang ada dalam block describe terkait
- **xit**: untuk menganulir statement it yang ada.
- **fdescribe**: hanya menjalankan rangkaian test dalam describe ini dan mengabaikan test lain.
- **xit**: hanya menjalankan statement it test ini dan mengabaikan test lain

```
xdescribe("A suite", function() {  
  // Status Pending. rangkaian test tidak akan dijalankan  
})
```

```
describe("A suite", function() {  
  xit("should bla bla", () => {  
    // Status Pending. statement it ini tidak dijalankan  
  })  
})
```

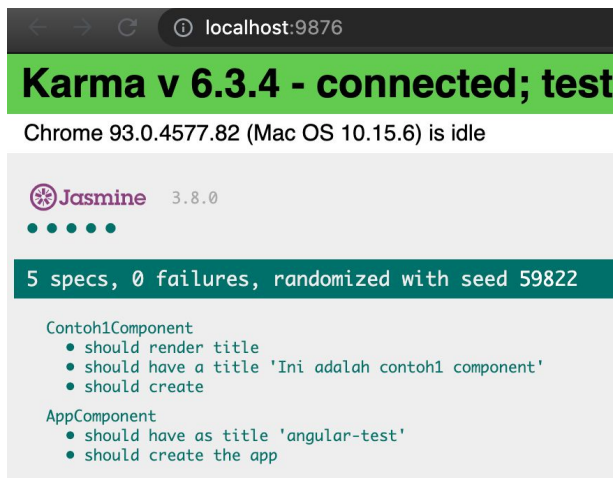
```
fdescribe("A suite3", function() {  
  // Prioritas. Hanya block ini yang dijalankan  
})
```

```
describe("A suite", function() {  
  fit("should bla bla", () => {  
    // Prioritas. Hanya statement it ini yang dijalankan  
  })  
})
```

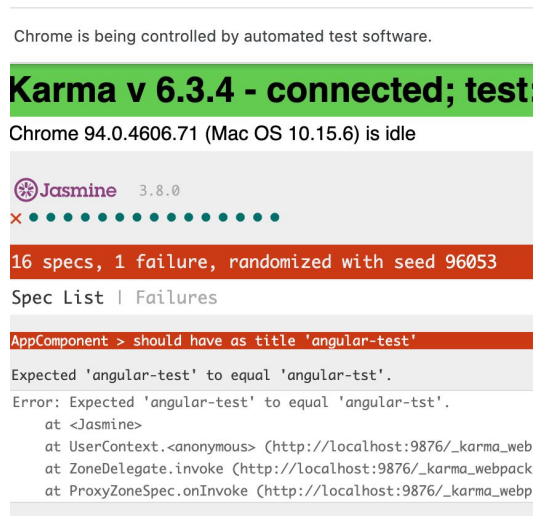


Angular Unit Test: Cara Run Testing

1. Command terminal `ng test`. Command ini akan menjalankan semua file .spec yang ada dalam aplikasi Angular
2. Command terminal `ng test --include=**/nama_file.spec` akan menjalankan file .spec terspesifik yang kita tentukan.



Contoh semua test case passed



Contoh 1 test case failed

Angular Unit Test: Simple Component Testing

File test akan berada di **.component.spec**. Biasanya sudah otomatis di-generate oleh Angular CLI. jika kamu menggunakan command `ng generate`.

Buatlah aplikasi angular baru dan generate 1 komponen (Disini, nama komponen adalah **contoh1**).

Dibawah ini file class component contoh1, dengan penambahan properti title.

```
contoh1.component.ts U x
src > app > contoh1 > A contoh1.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-contoh1',
5    templateUrl: './contoh1.component.html',
6    styleUrls: ['./contoh1.component.css']
7  })
8  export class Contoh1Component implements OnInit {
9    title = "Ini adalah contoh1 component"
10    constructor() { }
11
12    ngOnInit(): void {
13    }
14  }
15
```

```
contoh1.component.html U x
src > app > contoh1 > S contoh1.component.html >
1  <h1 class="title">{{title}}</h1>
2
```



Angular Unit Test: Simple Component Testing

```
contoh1.component.spec.ts U X
src > app > contoh1 > contoh1.component.spec.ts > ...
1  import { ComponentFixture, TestBed } from '@angular/core/testing';
2
3  import { Contoh1Component } from './contoh1.component';
4
5  describe('Contoh1Component', () => {
6    let component: Contoh1Component;
7    let fixture: ComponentFixture<Contoh1Component>;
8
9    beforeEach(async () => {
10      await TestBed.configureTestingModule({
11        declarations: [ Contoh1Component ]
12      })
13        .compileComponents();
14    });
15
16    beforeEach(() => {
17      fixture = TestBed.createComponent(Contoh1Component);
18      component = fixture.componentInstance;
19      fixture.detectChanges();
20    });
21
22    it('should create', () => {
23      expect(component).toBeTruthy();
24    });
25
26    it('should have a title 'Ini adalah contoh1 component'', () => {
27      fixture = TestBed.createComponent(Contoh1Component);
28      component = fixture.debugElement.componentInstance;
29      expect(component.title).toEqual('Ini adalah contoh1 component');
30    });
31
32    it('should render title', () => {
33      const fixture = TestBed.createComponent(Contoh1Component);
34      fixture.detectChanges();
35      const compiled = fixture.nativeElement as HTMLElement;
36      expect(compiled.querySelector('h1.title')?.textContent).toContain('Ini adalah contoh1 component');
37    });
38  });
```

Tambahkan 2 test case di file spec contoh1 untuk mengecek kesesuaian value properti title dan ter-render atau tidaknya title di template.



Angular Unit Test: Penjelasan

```
contoh1.component.spec.ts U x
src > app > contoh1 > contoh1.component.spec.ts > ...
1  import { ComponentFixture, TestBed } from '@angular/core/testing';
2
3  import { Contoh1Component } from './contoh1.component';
4
5  describe('Contoh1Component', () => {
6    let component: Contoh1Component;
7    let fixture: ComponentFixture<Contoh1Component>;
8
9    beforeEach(async () => {
10     await TestBed.configureTestingModule({
11       declarations: [ Contoh1Component ]
12     })
13     .compileComponents();
14   });
15
16   beforeEach(() => {
17     fixture = TestBed.createComponent(Contoh1Component);
18     component = fixture.componentInstance;
19     fixture.detectChanges();
20   });
21
22   it('should create', () => {
23     expect(component).toBeTruthy();
24   });
25
26   it('should have a title 'Ini adalah contoh1 component'', () => {
27     fixture = TestBed.createComponent(Contoh1Component);
28     component = fixture.debugElement.componentInstance;
29     expect(component.title).toEqual('Ini adalah contoh1 component');
30   });
31
32   it('should render title', () => {
33     const fixture = TestBed.createComponent(Contoh1Component);
34     fixture.detectChanges();
35     const compiled = fixture.nativeElement as HTMLElement;
36     expect(compiled.querySelector('h1.title')?.textContent).toContain('Ini adalah contoh1 component');
37   });
38 });
```

Sudah disediakan
saat pertama kali
generate
component

beforeEach Dimana didalamnya mempersiapkan dan membentuk komponen. **beforeEach pertama** untuk mengimport module/service yang dibutuhkan dalam test

it block pertama, mengecek apakah instance komponen dibuat atau tidak.

(Property test case)

It block kedua mengecek apakah component memiliki title yang sesuai ekspektasi.

(DOM display test case)

It block ketiga mengecek apakah title terrender di template tag h1 class title.



Angular Unit Test: Penjelasan (2)

```
contoh1.component.spec.ts U x
src > app > contoh1 > contoh1.component.spec.ts > ...
1  import { ComponentFixture, TestBed } from '@angular/core/testing';
2
3  import { Contoh1Component } from './contoh1.component';
4
5  describe('Contoh1Component', () => {
6    let component: Contoh1Component;
7    let fixture: ComponentFixture<Contoh1Component>;
8
9    beforeEach(async () => {
10      await TestBed.configureTestingModule({
11        declarations: [ Contoh1Component ]
12      })
13        .compileComponents();
14    });
15
16    beforeEach(() => {
17      fixture = TestBed.createComponent(Contoh1Component);
18      component = fixture.componentInstance;
19      fixture.detectChanges();
20    });
21
22    it('should create', () => {
23      expect(component).toBeTruthy();
24    });
25
26    it('should have a title 'Ini adalah contoh1 component'', () => {
27      fixture = TestBed.createComponent(Contoh1Component);
28      component = fixture.debugElement.componentInstance;
29      expect(component.title).toEqual('Ini adalah contoh1 component');
30    });
31
32    it('should render title', () => {
33      const fixture = TestBed.createComponent(Contoh1Component);
34      fixture.detectChanges();
35      const compiled = fixture.nativeElement as HTMLElement;
36      expect(compiled.querySelector('h1.title')?.textContent).toContain('Ini adalah contoh1 component');
37    });
38  });
```

Method **TestBed.createComponent()** dipanggil **agar** dapat digunakan untuk mengakses komponen dan template. Dengan adanya **TestBed**, tidak memberikan side-effect pada aplikasi aslinya.

fixture.detectChanges() Untuk mendeteksi perubahan dalam komponen.

.nativeElement, berguna untuk mengakses DOM template.

Maka otomatis, behaviour beserta method DOM, bisa dipanggil disini



Angular Unit Test: Scenario

Sebelum memulai testing, sangat penting menentukan skenario test. Tidak ada aturan baku dalam pembuatan skenario ini.

Sekarang, kita akan mencoba membuat unit test untuk login form. Dengan skenario:

1. Template merender email dan password dalam login form
2. Mengecek initial value dalam form group
3. Memastikan email memiliki validasi required, minLength, dan email type
4. Memastikan form tidak valid bila input kosong
5. Memastikan form valid ketika semua validasi terpenuhi
6. Memastikan form valid dapat di-submit dengan proses sesuai ekspektasi

Angular Unit Test: Membuat Login Form

```
reactive-form.component.ts U X
angular-test > src > app > components > reactive-form > reactive-form.component.ts
1 import { Component } from '@angular/core';
2 import { FormGroup, FormControl, Validators } from '@angular/forms';
3
4 interface User {
5   isLoggedIn: boolean;
6   email?: string;
7 }
8
9 @Component({
10   selector: 'app-reactive-form',
11   templateUrl: './reactive-form.component.html',
12   styleUrls: ['./reactive-form.component.css']
13 })
14
15 export class ReactiveFormComponent {
16   currentUser: User = { isLoggedIn: false };
17
18   loginForm: FormGroup = new FormGroup({
19     email: new FormControl('', [
20       Validators.minLength(5),
21       Validators.email,
22       Validators.required,
23     ]),
24     password: new FormControl('', [Validators.required])
25   });
26
27   get email() {
28     return this.loginForm.get('email');
29   }
30
31   get password() {
32     return this.loginForm.get('password');
33   }
34
35   onLogin() {
36     this.currentUser = {
37       isLoggedIn: true,
38       email: this.loginForm.value.email
39     };
40   }
41 }
42
43
```

```
reactive-form.component.html U X
angular-test > src > app > components > reactive-form > reactive-form.component.html
1 <div class="container">
2   <h1>Login Form</h1>
3
4   <form [formGroup]="loginForm" (ngSubmit)="onLogin()" id="loginForm">
5     <div class="form-group">
6       <label for="name"> Email</label>
7       <input
8         type="email"
9         class="form-control"
10         id="email"
11         name="email"
12         formControlName="email"
13       >
14       <span *ngIf="email && email.touched && email.invalid"
15         style="color: red">
16         Input Email is invalid
17       </span>
18     </div>
19
20     <div class="form-group">
21       <label for="name"> Password</label>
22       <input
23         type="password"
24         class="form-control"
25         id="password"
26         name="password"
27         formControlName="password"
28       >
29
30       <span *ngIf="password && password.touched && password.invalid"
31         style="color: red">
32         Input Password is invalid
33       </span>
34     </div>
35
36     <button
37       type="submit"
38       class="btn btn-success"
39       [disabled]="!loginForm.valid"
40     >
41     Login
42   </button>
43 </form>
44
45 <div *ngIf="currentUser.isLoggedIn">
46   Successfully login. Hello, {{currentUser.email}}!
47 </div>
48 </div>
```

Buatlah 1 komponen untuk login form menggunakan reactive form.

Contoh komponen disini bernama reactive-form.



Angular Unit Test: Reactive Form Testing

```
import { ComponentFixture, TestBed } from '@angular/core/testing';  
import { ReactiveFormComponent } from './reactive-form.component';  
import { ReactiveFormsModule } from '@angular/forms'
```

```
beforeEach(async () => {  
  await TestBed.configureTestingModule({  
    imports: [ReactiveFormsModule],  
    declarations: [ ReactiveFormComponent ]  
  })  
  .compileComponents();  
});
```

Dalam file form .spec,

1. Import **ReactiveFormsModule**
2. Masukkan ke metadata **imports** di block beforeEach pertama.



Angular Unit Test: Reactive Form - Test Case 1

```
it('should render email and password input elements', () => {  
  const compiled = fixture.debugElement.nativeElement;  
  const formElement = compiled.querySelector('#loginForm')  
  const emailInputElement = formElement.querySelector('input[id="email"]');  
  const passInputElement = formElement.querySelector('input[id="password"]');  
  
  expect(emailInputElement).toBeTruthy();  
  expect(passInputElement).toBeTruthy();  
});
```

Test case 1 - Memastikan bahwa input email dan password terrender dalam template

1. Karena berhubungan dengan template, gunakan **fixture.debugElement.nativeElement** untuk mengakses DOM.
2. Gunakan **.querySelector** untuk men-select element dalam dom.
3. Setelah memastikan elemen input email dan password ter-select, maka cek dengan matcher **toBeTruthy** untuk memastikan elemen ada.

Angular Unit Test: Reactive Form - Test Case 2

```
loginForm: FormGroup = new FormGroup({  
  email: new FormControl('', [  
    Validators.minLength(5),  
    Validators.email,  
    Validators.required,  
  ]),  
  password: new FormControl('', [Validators.required])  
})
```

.component.ts

```
it('Check initial value of login form group', () => {  
  const loginFormGroup = component.loginForm  
  const loginFormValue = {  
    email: '',  
    password: ''  
  }  
  expect(loginFormGroup.value).toEqual(loginFormValue)  
});
```

.spec.ts

Test case 2 - Memastikan initial value di login form group.

Karena form login memang kosong di awal, maka pengecekan test juga melihat apakah email dan password dalam loginForm juga kosong saat pertama kali terrender.

1. Untuk mengecek, bisa mengakses class FormGroup dari component. Disini, dimasukkan ke properti loginForm. (component.loginForm)
2. Kemudian membuat value dummy dengan email dan password berupa string kosong.
3. Membuat ekspektasi bahwa value component.loginForm akan sama dengan value dummy yang sebelumnya dibuat.



Angular Unit Test: Reactive Form - Test Case 3

```
get email() {  
  return this.loginForm.get('email')  
}
```

```
loginForm: FormGroup = new FormGroup({  
  email: new FormControl('', [  
    Validators.minLength(5),  
    Validators.email,  
    Validators.required,  
  ]),  
  password: new FormControl('', [Validators.required])  
})
```

.component.ts

```
it('Validate email input: `required, minLength(5), email type`', () => {  
  const email = component.email  
  
  email?.setValue('')  
  expect(email?.hasError('required')).toBeTruthy()  
  
  email?.setValue('abcd')  
  expect(email?.hasError('minlength')).toBeTruthy()  
  
  email?.setValue('abcdefghij')  
  expect(email?.hasError('email')).toBeTruthy()  
});
```

.spec.ts

Test case 3 - Memastikan email input memiliki validasi

Email input diskenariokan mempunyai validasi email, required dan minLength(5), yang didefinisikan di class component. Maka, dalam test case yang akan dibuat memastikan error validasi **pasti** terjadi bila input tidak sesuai dengan validasi.

1. Ambil email dalam loginForm di component (Disini memanggil via getter yang dibuat dalam component)
2. Set value ke email. Dengan **.setValue()**
3. Cek apakah benar ada error dengan **.hasError** dan matcher **toBeTruthy()**



Angular Unit Test: Reactive Form - Test Case 4

```
get email() {  
  return this.loginForm.get('email')  
}  
  
get password() {  
  return this.loginForm.get('password')  
}
```

```
loginForm: FormGroup = new FormGroup({  
  email: new FormControl('', [  
    Validators.minLength(5),  
    Validators.email,  
    Validators.required,  
  ]),  
  password: new FormControl('', [Validators.required])  
})
```

.component.ts

```
it('Invalid form when empty', () => {  
  const email = component.email  
  const password = component.password  
  
  email?.setValue('')  
  password?.setValue('')  
  expect(component.loginForm.valid).toBeFalsy();  
});
```

.spec.ts

Test case 4 - Memastikan form tidak valid bila input email dan password kosong

1. Ambil email dan password dalam **loginForm** dari component (Disini memanggil via getter yang dibuat dalam component)
2. Set value string kosong ke email dan password Dengan **.setValue()**
3. Ambil validity dari loginForm **component class valid**, dan pastikan nilainya false (tidak valid) dengan matcher **toBeFalsy()**



Angular Unit Test: Reactive Form - Test Case 5

```
const validUser = {  
  email: 'test@mail.com',  
  password: '12345'  
}
```

.spec.ts

```
it('Check form validity when validators are fulfilled', () => {  
  const compiled = fixture.debugElement.nativeElement  
  const emailInputElement = compiled.querySelector('input[id="email"]')  
  const passInputElement = compiled.querySelector('input[id="password"]');  
  
  if (!!emailInputElement && !!passInputElement) {  
    emailInputElement.value = validUser.email  
    emailInputElement.dispatchEvent(new Event('input'))  
  
    passInputElement.value = validUser.password  
    passInputElement.dispatchEvent(new Event('input'))  
    fixture.detectChanges()  
  }  
  
  expect(component.loginForm.valid).toBeTruthy()  
});
```

.spec.ts

Test case 5 - Memastikan form valid ketika semua validasi terpenuhi

Proses test case disini akan dimulai dari input ke DOM element.

1. Definisikan validUser object di file .spec.
2. Ambil dom element input email dan password, setValue dengan value validUser
3. Panggil **dispatchEvent('input')** untuk mentrigger perubahan input ke FormGroup (loginForm) di component class.
4. Cek apakah **loginForm** di component class valid.



Angular Unit Test: Reactive Form - Test Case 6

```
onLogin( ) {  
  this.currentUser = {  
    isLogin: true,  
    email: this.loginForm.value.email  
  }  
}
```

.component.ts

```
it('Submitting form and set currentUser', () => {  
  const compiled = fixture.debugElement.nativeElement  
  const emailInputElement = compiled.querySelector('input[id="email"]')  
  const passInputElement = compiled.querySelector('input[id="password"]');  
  
  if (!!emailInputElement && !!passInputElement) {  
    emailInputElement.value = validUser.email  
    emailInputElement.dispatchEvent(new Event('input'))  
  
    passInputElement.value = validUser.password  
    passInputElement.dispatchEvent(new Event('input'))  
    fixture.detectChanges()  
  }  
  
  const button = compiled.querySelector('button')  
  button.click()  
  
  expect(component.currentUser.email).toEqual(validUser.email)  
  expect(component.currentUser.isLogin).toBeTruthy()  
})
```

Test case 6 -Memastikan form valid dapat di-submit dengan proses sesuai ekspektasi

Proses submit form akan ke method onLogin() dimana, men-set properti currentUser dan mengubah status isLogin menjadi true.

1. Hampir sama dengan test case 5, ambil dom elemen input email dan password. Pastikan value dan event terpenuhi.
2. Karena proses submit di-trigger oleh button click, maka ambil elemen button dan panggil **.click()**
3. Pastikan currentUser saat ini memiliki status isLogin true dan email yang sama dengan validUser.



Angular Unit Test: Service Testing

Selain komponen, kita juga bisa membuat testing service HttpClient.

Sekarang, kita akan membuat unit test untuk fetch post list. Mengecek apakah endpoint dan http method yang dikonsumsi sesuai ekspektasi.

Pertama, buatlah 1 service disamping yang mengconsume API.

<https://jsonplaceholder.typicode.com/posts>

```
angular-test > src > app > shared > post.service.ts > PostService
1  import { Injectable } from '@angular/core';
2  import { Observable } from 'rxjs';
3  import { HttpClient } from '@angular/common/http';
4
5  export interface Post {
6    userId: number;
7    id: number;
8    title: string;
9    body: string;
10 }
11
12 @Injectable({
13   providedIn: 'root'
14 })
15
16 export class PostService {
17   ENDPOINT: string = 'https://jsonplaceholder.typicode.com/posts';
18
19   constructor(private http: HttpClient) { }
20
21   getPosts(): Observable<Post[]> {
22     return this.http.get<Post[]>(`${this.ENDPOINT}`)
23   }
24 }
```



Angular Unit Test: Service Testing

```
angular-test / src / app / shared > post.service.spec.ts / ...  
import { TestBed } from '@angular/core/testing';  
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';  
import { PostService } from './post.service';  
  
describe('PostService', () => {  
  let postService: PostService;  
  let httpMock: HttpTestingController;  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      imports: [  
        HttpClientTestingModule,  
      ],  
      providers: [  
        PostService  
      ],  
    });  
  
    postService = TestBed.inject(PostService);  
    httpMock = TestBed.inject(HttpTestingController);  
  });  
});
```

Karena kita ingin mengetest HTTP request maka, kita membutuhkan *fake / mock* http agar tidak mempengaruhi aplikasi aslinya. (import module HttpClientTestingModule dan HttpTestingController)

Konfigurasi module yang dibutuhkan di beforeEach pertama.



Angular Unit Test: Service Testing

```
it('should fetch posts successfully', () => {  
  const postItem = [  
    {  
      "userId": 1,  
      "id": 1,  
      "title": "sunt aut facere repellat pr",  
      "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et "  
    },  
    {  
      "userId": 1,  
      "id": 2,  
      "title": "qui est esse",  
      "body": "est rerum tempore vitae\nsequi sint "  
    },  
  ]  
  
  postService.getPosts()  
    .subscribe((posts: any) => {  
      expect(posts.length).toBe(2);  
    });  
  
  let req = httpMock.expectOne('https://jsonplaceholder.typicode.com/posts');  
  expect(req.request.method).toBe("GET");  
  
  req.flush(postItem);  
  httpMock.verify();  
});
```

Test case - mengetes apakah fetch posts berhasil

1. Siapkan fake data response.
2. Panggil method yang ingin di test dalam service. Karena `getPosts` adalah observable, maka kita panggil dengan `.subscribe`.
3. Buat ekspektasi response berdasar fake data response. Disini dengan mengecek panjang response
4. Pastikan endpoint dan http method sesuai dengan ekspektasi
5. Panggil **`req.flush()`** untuk memasukkan fake response ke hasil request
6. `httpMock.verify()` memvalidasi bahwa request berjalan dengan lancar.

