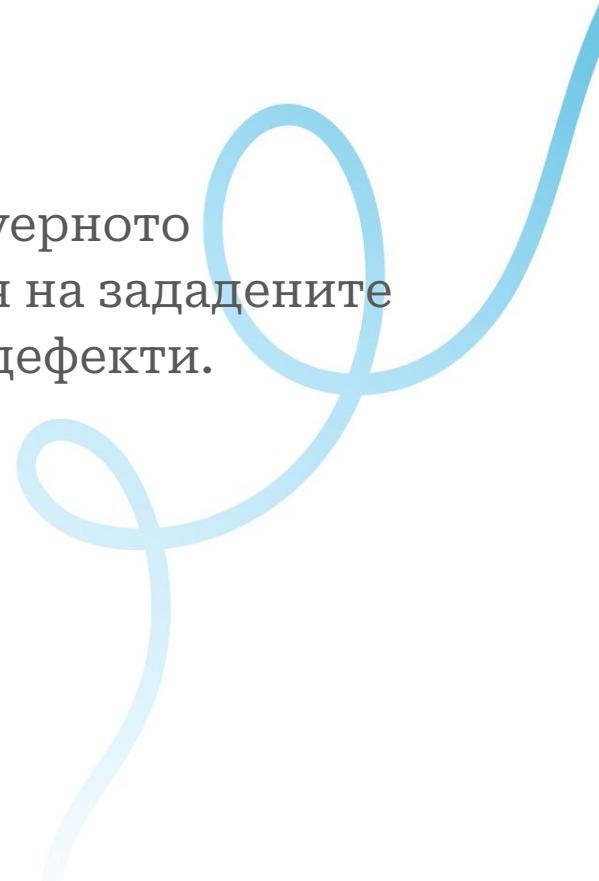


Testing

Лектор: Петър Маламов

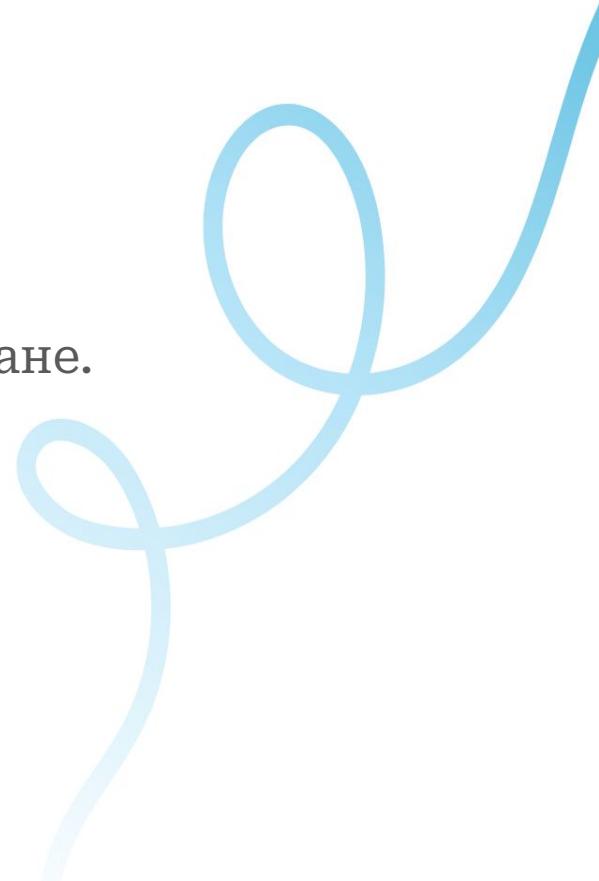
Какво е Testing?

Testing е процес на проверка и оценка на софтуерното приложение, за да се гарантира, че то отговаря на зададените изисквания, работи правилно и е свободно от дефекти.



Цел на тестването

- Идентифициране и коригиране на грешки.
- Подобряване на качеството.
- Подобряване на потребителското изживяване.



Testing b Angular

Testing в Angular

Angular CLI изтегля и инсталира всичко необходимо за тестване на Angular приложение с тестовата рамка **Jasmine**.

Angular използва **Jasmine** като тестова рамка за писане на тестове и **Karma** за стартирането им в браузър.

В Angular тестовете се стартират с помощта на командата:

```
ng test
```

Видове тестване в Angular

- **Unit Testing** - Проверява отделни функции, услуги и компоненти в изолация.
- **Integration Testing** - Оценява взаимодействието между различни модули и компоненти.
- **End-to-End Testing** - Симулира реални потребителски действия, за да се провери цялостното функциониране на приложението.

Основни части на тестовете

TS spec.ts

```
describe('MyService', () => {
  // Тук ще поставим тестовете за MyService
});
```

Групира свързани тестове.
Описва какво се тества

Основни части на тестовете

TS spec.ts

```
describe('MyService', () => {
  it('should return the correct value', () => {
    const result = 2 + 3; // Изпълняваме тестваната логика
  });
});
```

it съдържа конкретен тест.
Той описва какво точно се
тества и включва логиката
на теста.



www.diadraw.com

Основни части на тестовете

```
ts spec.ts

describe('MyService', () => {
  it('should return the correct value', () => {
    const result = 2 + 3; // Изпълняваме тестваната логика
    expect(result).toBe(5); // Проверяваме резултата
  );
});
```

expect служи за проверка дали тестът е минал успешно

Основни части на тестовете

TS spec.ts

```
describe('MyService', () => {
  let service: MyService;

  beforeEach(() => {
    service = new MyService(); // Създаваме нов екземпляр на услугата преди всеки тест
  });
  it('should return the correct value', () => {
    const result = service.add(2, 3); // Изпълняваме тестваната логика
    expect(result).toBe(5); // Проверяваме резултата
  });
});
```

beforeEach е функция, която се изпълнява преди всеки тест. Използва се за настройка на общо състояние.

TestBed

TestBed създава динамично конструиран тестов модул в Angular, който имитира **@NgModule**.

Позволява конфигуриране и създаване на компоненти и услуги за тестване, както и взаимодействие с тествания компонент.

Методът **TestBed.configureTestingModule()** приема обект, който съдържа свойства на **@NgModule**.

TS spec.ts

```
describe('MyService', () => {
  let service: MyService;

  beforeEach(() => {
    TestBed.configureTestingModule({ providers: [MyService] });
    service = TestBed.inject(MyService);
  });
  it('should return the correct value', () => {
    const result = service.add(2, 3);
    expect(result).toBe(5);
  });
});
```

Използване на service-а чрез **TestBed.inject()**

Важно!

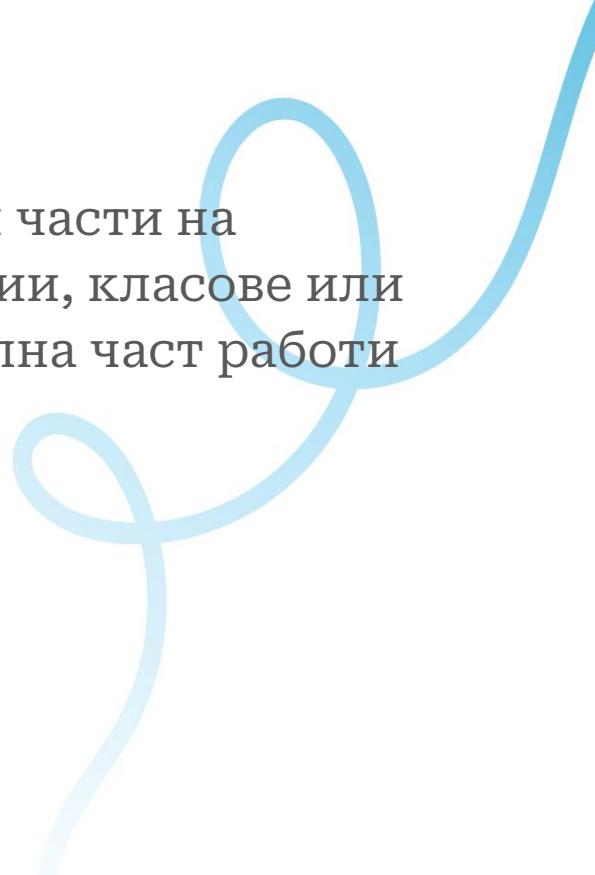
Когато използваме TestBed за конфигуриране и създаване на компоненти и услуги, той автоматично разрешава всички зависимости, свързани с тях.

```
@Injectable({
  providedIn: 'root',
})
class MyService {
  constructor(private os: OtherService) {}
}
```

Unit Testing

Unit Testing

Процес на тестване на най-малките изолирани части на софтуерното приложение – обикновено функции, класове или методи. Целта е да се гарантира, че всяка отделна част работи правилно сама по себе си.



Тестване на Service

TS service.ts

```
@Injectable({
  providedIn: 'root',
})
export class CounterService {
  count: number = 0;
  constructor() {}

  add(number: number) {
    this.count += number;
  }

  subtract(number: number) {
    this.count -= number;
  }
}
```

TS service.spec.ts

```
describe('CounterService', () => {
  let service: CounterService;

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

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('add to count', () => {
    service.add(1);
    expect(service.count).toBe(1);
  });

  it('subtract from count', () => {
    service.count = 2;
    service.subtract(1);
    expect(service.count).toBe(1);
  });
});
```

Service with Observables

TS service.ts

```
@Injectable({
  providedIn: 'root',
})
export class CounterService {
  private countSubject = new BehaviorSubject<number>(0);
  count$ = this.countSubject.asObservable();

  constructor() {}

  add(value: number): void {
    const current = this.countSubject.value;
    this.countSubject.next(current + value);
  }

  subtract(value: number): void {
    const current = this.countSubject.value;
    this.countSubject.next(current - value);
  }
}
```

TS service.spec.ts

```
describe('CounterService', () => {
  let service: CounterService;

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

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('add to count', (done) => {
    service.add(10);
    service.count$.subscribe((value) => {
      expect(value).toBe(10);
      done();
    });
  });

  it('subtract from count', () => {
    service.add(10);
    service.subtract(4);
    service.count$.subscribe((value) => {
      expect(value).toBe(6);
    });
  });
});
```

HTTP Services

TS service.spec.ts

```
describe('CartService', () => {
  let service: CartService;
  let httpServiceMock: jasmine.SpyObj<HttpClient>;
  const cartMockData: Cart = {
    items: [{ productId: 1, name: 'test', quantity: 1, price: 1 }],
    total: 1,
  };

  beforeEach(() => {
    httpServiceMock = jasmine.createSpyObj('HttpClient', ['get']);
    httpServiceMock.get.and.returnValue(of(cartMockData));
    TestBed.configureTestingModule({
      providers: [
        CartService,
        {
          provide: HttpClient,
          useValue: httpServiceMock,
        },
      ],
    });
    service = TestBed.inject(CartService);
  });

  it('should return expected cart', (done: DoneFn) => {
    service.getCartFromServer().subscribe((cart) => {
      expect(cart).toEqual(cartMockData);
      done();
    });

    expect(httpServiceMock.get.calls.count()).toBe(1);
  });
});
```

TS service.ts

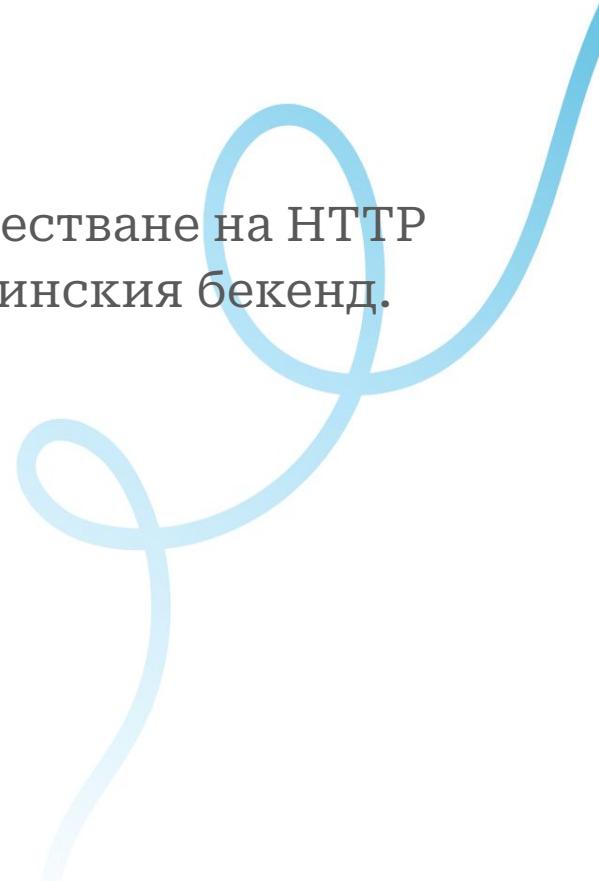
```
@Injectable({
  providedIn: 'root',
})
export class CartService {
  private cartSubject = new BehaviorSubject<Cart>({ items: [], total: 0 });
  cart$ = this.cartSubject.asObservable();

  constructor(private http: HttpClient) {}

  getCartFromServer() {
    return this.http.get<Cart>('https://your-api-url.com/cart').pipe(
      tap((cart) => {
        this.cartSubject.next(cart);
      })
    );
  }
}
```

HttpClientTestingModule

Модул, който предоставя функционалност за тестване на HTTP заявки, като използва mock сървър вместо истинския бекенд.



Как работи HttpClientTestingModule?

За тестване на HttpClient се конфигурира TestBed с **provideHttpClient()** и **provideHttpClientTesting()**, които карат HttpClient да използва тестови бекенд вместо реалната мрежа.

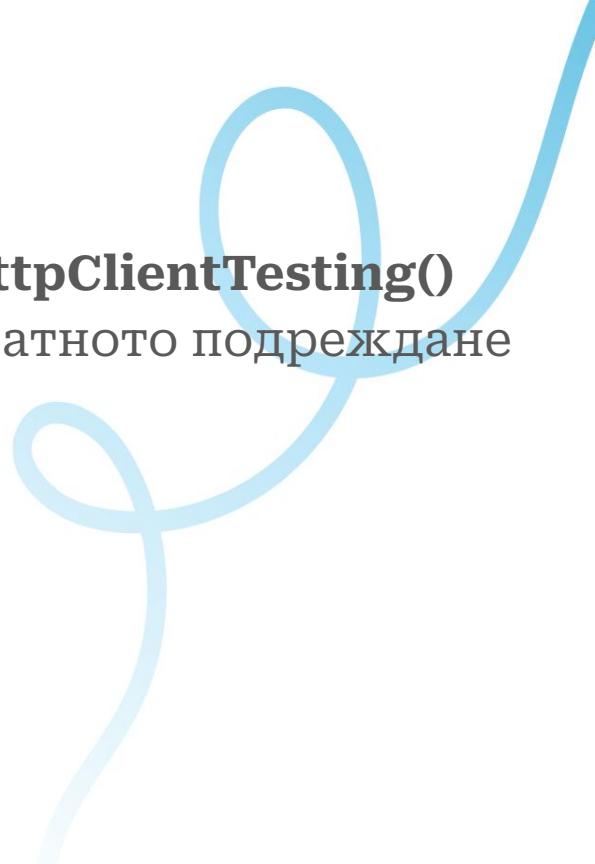
Това осигурява достъп до **HttpTestingController**, който позволява взаимодействие с тестовия бекенд, задаване на очаквания за заявките и изпращане на отговори към тях.

TS service.spec.ts

```
beforeEach(() => {
  TestBed.configureTestingModule({
    providers: [provideHttpClient(), provideHttpClientTesting()],
  });
  httpTesting = TestBed.inject(HttpTestingController);
});
```

Важно!

`provideHttpClient()` трябва да бъде преди `provideHttpClientTesting()`, тъй като `provideHttpClientTesting()` ще презапише част от `provideHttpClient()`. Обратното подреждане може да наруши тестовете.



Обработка на заявки

Завършване на заявката с очаквания резултат

Проверка на получения резултат

Проверка за наличие на незавършени заявки

TS service.spec.ts

```
it('should be fetch cart', () => {
  const cartRequest$ = service.getCartFromServer();

  const cartPromise = firstValueFrom(cartRequest$);

  const req = httpTesting.expectOne('/cart', 'Request to load the cart');

  expect(req.request.method).toBe('GET');

  req.flush(cartMockData);

  expect(await cartPromise).toEqual(cartMockData);

  httpTesting.verify();
});
```

Взима observable от service-a

Абонира се за Observable създава Promise за отговора.

Проверка на заявката чрез HttpTestingController

Тестване на HTTP грешки

TS service.spec.ts

```
it('should be fetch cart', () => {
  const cartRequest$ = service.getCartFromServer();

  const cartPromise = firstValueFrom(cartRequest$);

  const req = httpTesting.expectOne('/cart', 'Request to load the cart');

  expect(req.request.method).toBe('GET');

  req.flush('Failed!', {status: 500, statusText: 'Internal Server Error'});

  expect(await cartPromise).toEqual(cartMockData);

  httpTesting.verify();
});
```

Тестване на Component

Компонентът в Angular представлява повече от неговия клас, тъй като взаимодейства с DOM и други компоненти.

За да се провери правилното му рендиране, реакциите на потребителски действия и интеграцията с други компоненти, е необходимо да се създадат и изследват DOM елементите, свързани с него.

Такива тестове използват TestBed и допълнителни инструменти за симулиране на взаимодействия и потвърждаване на очакваното поведение на компонента.

TS service.spec.ts

```
describe('AppComponent', () => {
  let fixture: ComponentFixture<AppComponent>;
  let component: AppComponent;

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

    fixture = TestBed.createComponent(AppComponent);
    component = fixture.componentInstance;
  });

  it('should create the app', () => {
    expect(component).toBeTruthy();
  });
});
```



www.diadraw.com

TS service.spec.ts

```
describe('AppComponent', () => {
  let fixture: ComponentFixture<AppComponent>;
  let component: AppComponent;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [AppComponent],
    }).compileComponents(); ←

    fixture = TestBed.createComponent(AppComponent);
    component = fixture.componentInstance;
  });

  it('should create the app', () => {
    expect(component).toBeTruthy();
  });
});
```

Използва за компилиране на шаблоните и стиловете на компонентите в тестовия модул.

TS service.spec.ts

```
describe('AppComponent', () => {
  let fixture: ComponentFixture<AppComponent>;
  let component: AppComponent;

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

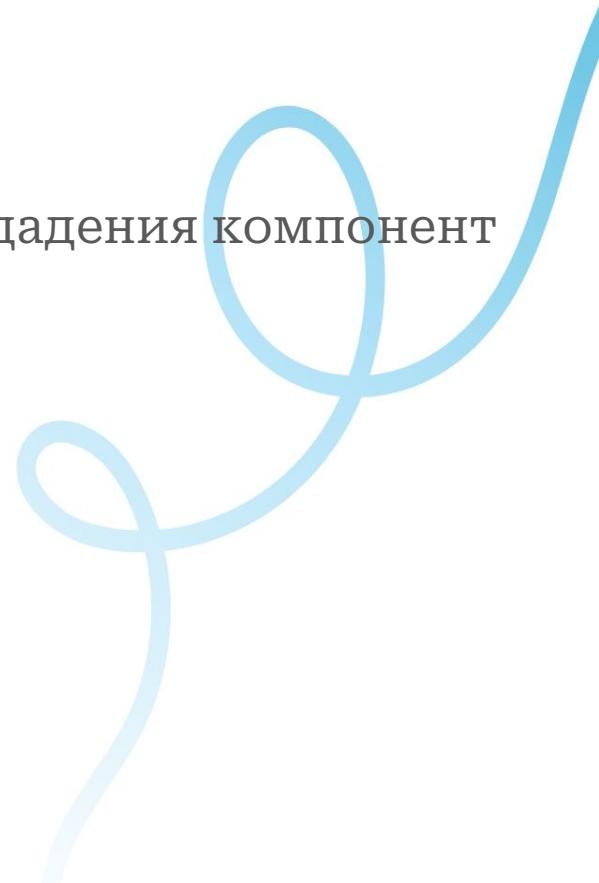
    fixture = TestBed.createComponent(AppComponent);
    component = fixture.componentInstance;
  });

  it('should create the app', () => {
    expect(component).toBeTruthy();
  });
});
```

Създава инстанция на компонента, добавя съответния елемент в DOM на тестовата среда и връща обект от тип **ComponentFixture**.

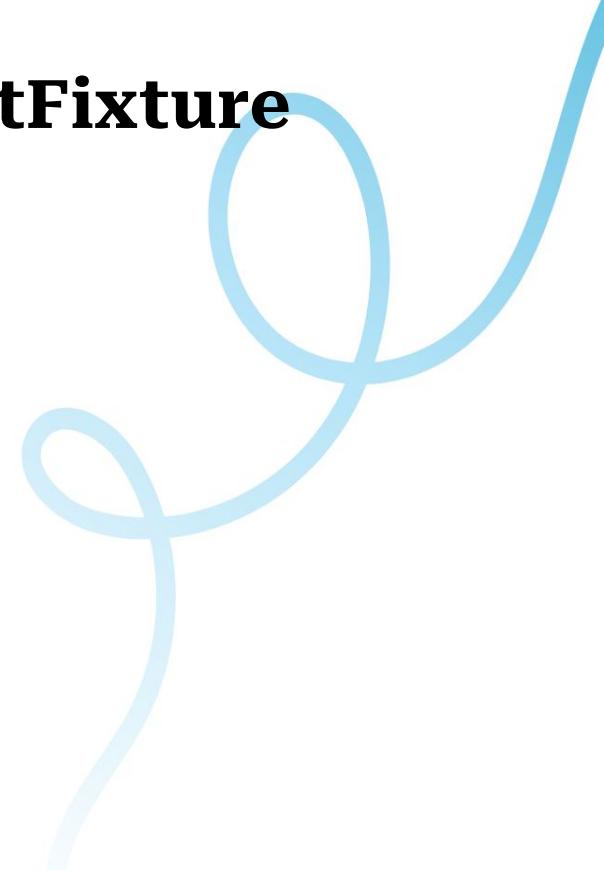
ComponentFixture

Тестов инструмент за взаимодействие със създавания компонент и неговия съответен DOM елемент.



Основни функции на ComponentFixture

- Достъп до инстанцията на компонента
- Достъп до DOM елемента
- Задействане на Change Detection
- Симулиране на потребителски действия



Достъп до DOM елемента

ComponentFixture ни предоставя 2 начина, чрез които можем да получим достъп до DOM елемента на компонента:

- **nativeElement** - предоставя достъп до коренния DOM елемент на компонента. Типът му е any, тъй като Angular не може предварително да определи дали това е HTML елемент или не, особено при работа на различни платформи
- **debugElement** - debugElement предоставя абстракция над DOM елементите, която работи на всички платформи, поддържани от Angular. DebugElement.nativeElement връща платформено-специфичния елемент

By.css()

Метод, който се използва с **DebugElement.query()** за селектиране на елементи чрез CSS селектори в тестова среда.

Той е създаден за кросплатформена съвместимост, осигурявайки селекция на елементи дори на платформи без пълна поддръжка на DOM API (като при SSR).

TS component.spec.ts

```
it('should find the <div> tag', () => {
  const appDe: DebugElement = fixture.debugElement;
  const divDe: DebugElement = appDe.query(By.css('div'));
  const div: HTMLElement = divDe.nativeElement;
  expect(div.textContent).toEqual('div works!');
});
```

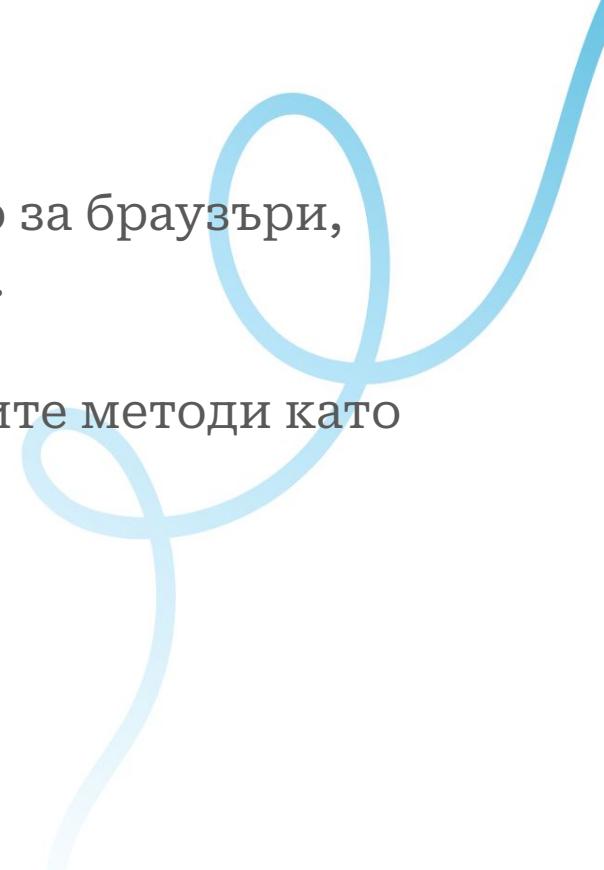


www.diadraw.com

Важно!

Когато приложението работи и се билдва само за браузъри,
използването на **By.css** може да бъде излишно.

По-просто и ясно е да се използват стандартните методи като
querySelector() или **querySelectorAll()**.



TS component.spec.ts

```
describe('CardComponent', () => {
  let component: CardComponent;
  let fixture: ComponentFixture<CardComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CardComponent]
    })
    .compileComponents();

    fixture = TestBed.createComponent(CardComponent);
    component = fixture.componentInstance;
  });

  it('should have title', () => {
    const p = fixture.nativeElement.querySelector('p');
    expect(p.textContent).toBe('Test Title');
  });
});
```

TOTAL: 1 FAILED,

TS component.spec.ts

```
@Component({
  selector: 'app-card',
  standalone: true,
  template: '<p>{{ title() }}</p>',
})
export class CardComponent {
  title = signal('Test Title');
}
```

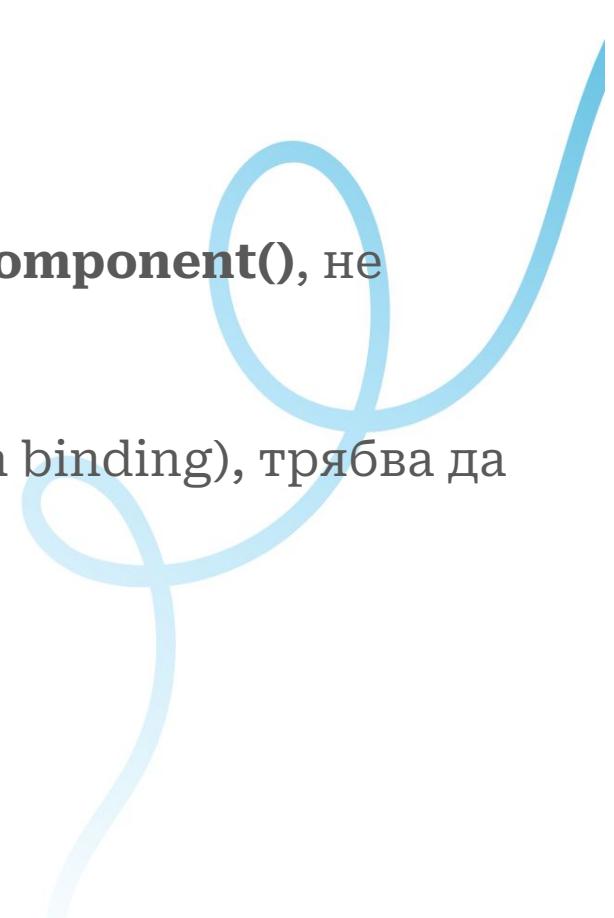


www.diadraw.com

Change detection

При съдаване на компонент с **TestBed.createComponent()**, не задейства автоматично change detection.

За да се задейства обвързването на данни (data binding), трябва да се използва **fixture.detectChanges()**.



TS component.spec.ts

```
describe('CardComponent', () => {
  let component: CardComponent;
  let fixture: ComponentFixture<CardComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CardComponent]
    })
    .compileComponents();

    fixture = TestBed.createComponent(CardComponent);
    component = fixture.componentInstance;

    fixture.detectChanges();
  });

  it('should have title', () => {
    const p = fixture.nativeElement.querySelector('p');
    expect(p.textContent).toBe('Test Title');
  });
});
```

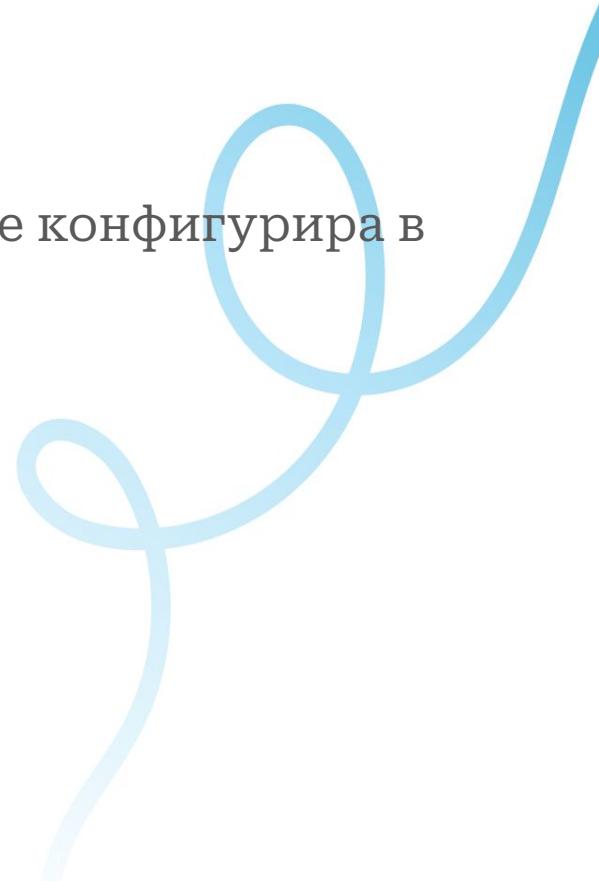
SUCCESS



www.diadraw.com

Automatic change detection

Автоматично откриване на промени може да се конфигурира в теста чрез **ComponentFixtureAutoDetect** или **ComponentFixture.autoDetectChanges()**



TS component.spec.ts

```
describe('CardComponent', () => {
  let component: CardComponent;
  let fixture: ComponentFixture<CardComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CardComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(CardComponent);
    component = fixture.componentInstance;

    fixture.autoDetectChanges();
  });

  it('should have title', () => {
    const p = fixture.nativeElement.querySelector('p');
    expect(p.textContent).toBe('Test Title');
  });
});
```

TS component.spec.ts

```
describe('CardComponent', () => {
  let component: CardComponent;
  let fixture: ComponentFixture<CardComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [CardComponent],
      providers: [{ provide: ComponentFixtureAutoDetect, useValue: true }],
    }).compileComponents();
  });

  fixture = TestBed.createComponent(CardComponent);
  component = fixture.componentInstance;
});

it('should have title', () => {
  const p = fixture.nativeElement.querySelector('p');
  expect(p.textContent).toBe('Test Title');
});
```

Dumb Component

Тестване на входните
параметри

Input Data



UI
(HTML + CSS)

Logic

Data

Тестване на изходно
събитие

Output data
& Events



Input Data

TS component.spec.ts

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

  fixture = TestBed.createComponent(CardComponent);
  component = fixture.componentInstance;

  fixture.detectChanges();
});

it('should have default title', () => {
  const p = fixture.nativeElement.querySelector('p');
  expect(p.textContent).toBe('Default');
});

it('should display title from input', () => {
  fixture.componentInstance.title = 'Custom title';
  fixture.detectChanges();
  const p = fixture.nativeElement.querySelector('p');
  expect(p.textContent).toBe('Custom title');
});
```

TS component.ts

```
@Component({
  selector: 'app-card',
  standalone: true,
  template: '<p>{{ title() }}</p>',
})
export class CardComponent {
  title = input('Default');
}
```



www.diadraw.com

Output Data

TS component.spec.ts

```
it('should emit action', () => {
  spyOn(component.action, 'emit');

  const button = fixture.debugElement.query(
    By.css('[data-testid="action-button"]')
  ).nativeElement;

  button.click();

  expect(component.action.emit).toHaveBeenCalled();
});
```

TS component.ts

```
@Component({
  selector: 'app-card',
  standalone: true,
  template: `<div>
    <p>{{ title() }}</p>
    <button data-testid="action-button"
      (click)="action.emit()">Action</button>
  </div>`,
})
export class CardComponent {
  title = input('Default');
  @Output() action = new EventEmitter<void>();
}
```

spyOn

Използва за създаване на "шпионин" върху съществуващ метод или функция, за да се проследи дали е извикан, с какви аргументи е извикан и колко пъти е извикан.

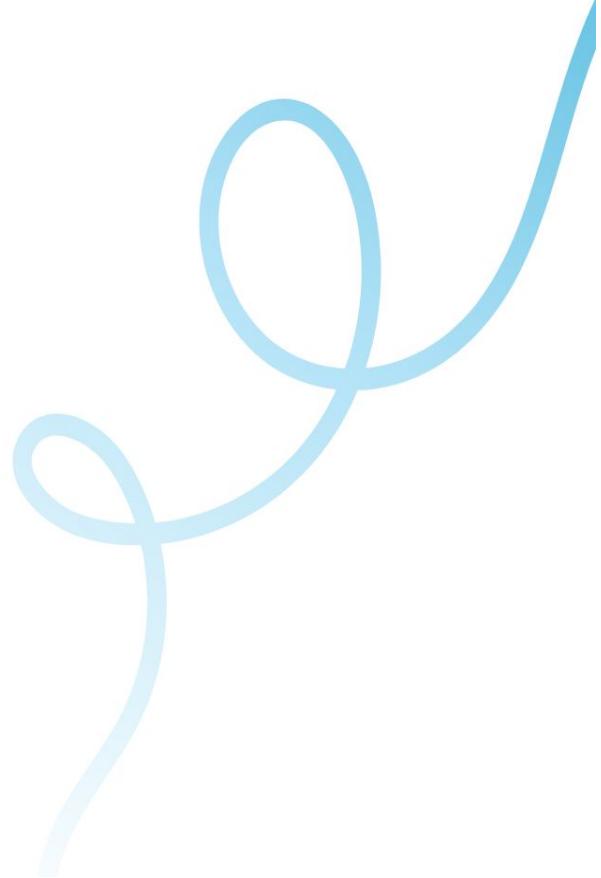
```
it('should call the add method once', () => {
  const myObject = { add: (a: number, b: number) => a + b };

  spyOn(myObject, 'add');

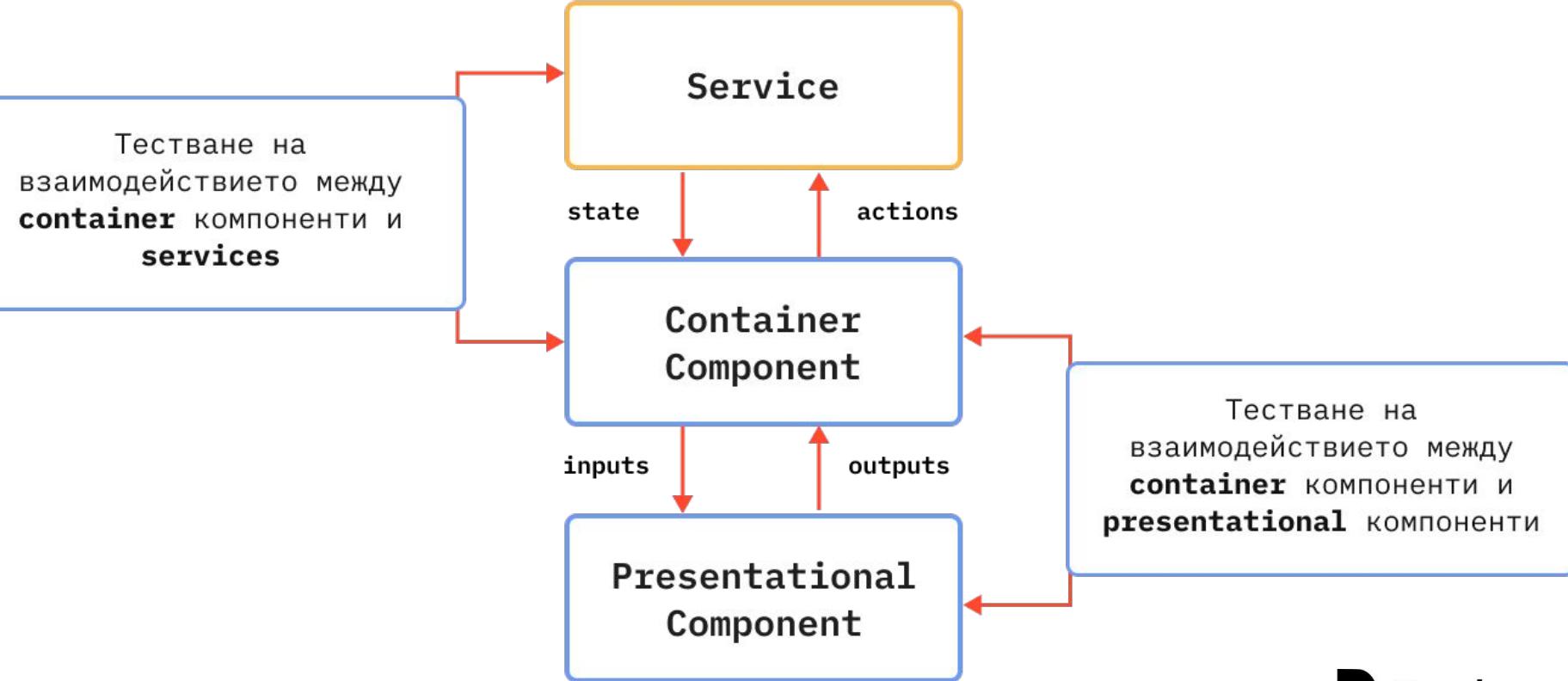
  myObject.add(2, 3);

  expect(myObject.add).toHaveBeenCalled();
  expect(myObject.add).toHaveBeenCalledWith(2, 3);
});
```

Упражнение



Integration Testing



Integration Testing

Процес на проверка на взаимодействието между различни компоненти или модули на система, за да се увери, че те работят правилно заедно.

Този тип тестване цели да открие проблеми в комуникацията между модули и да гарантира, че данни и функционалности се предават и обработват коректно между тях.

TS list.component.ts

```
@Component({
  selector: 'app-list',
  standalone: true,
  imports: [],
  template: `<div>
    <h1>{{ product.name }}</h1>
    <button (click)="select.emit(product.id)">Select</button>
  </div>
</div>`,
})
export class ListComponent {
  products = input<Product[]>([[]]);
  @Output() select = new EventEmitter<number>();
}
```

TS products.component.ts

```
@Component({
  selector: 'app-products',
  standalone: true,
  imports: [ListComponent],
  template: `<div>
    <app-list [products]="products" (select)="onSelect($event)"></app-list>
  </div>`,
})
export class ProductsComponent {
  products = this.productsService.getProducts();
  constructor(private productService: ProductsService) {}

  onSelect(productId: number) {
    this.productService.selectProduct(productId);
  }
}
```

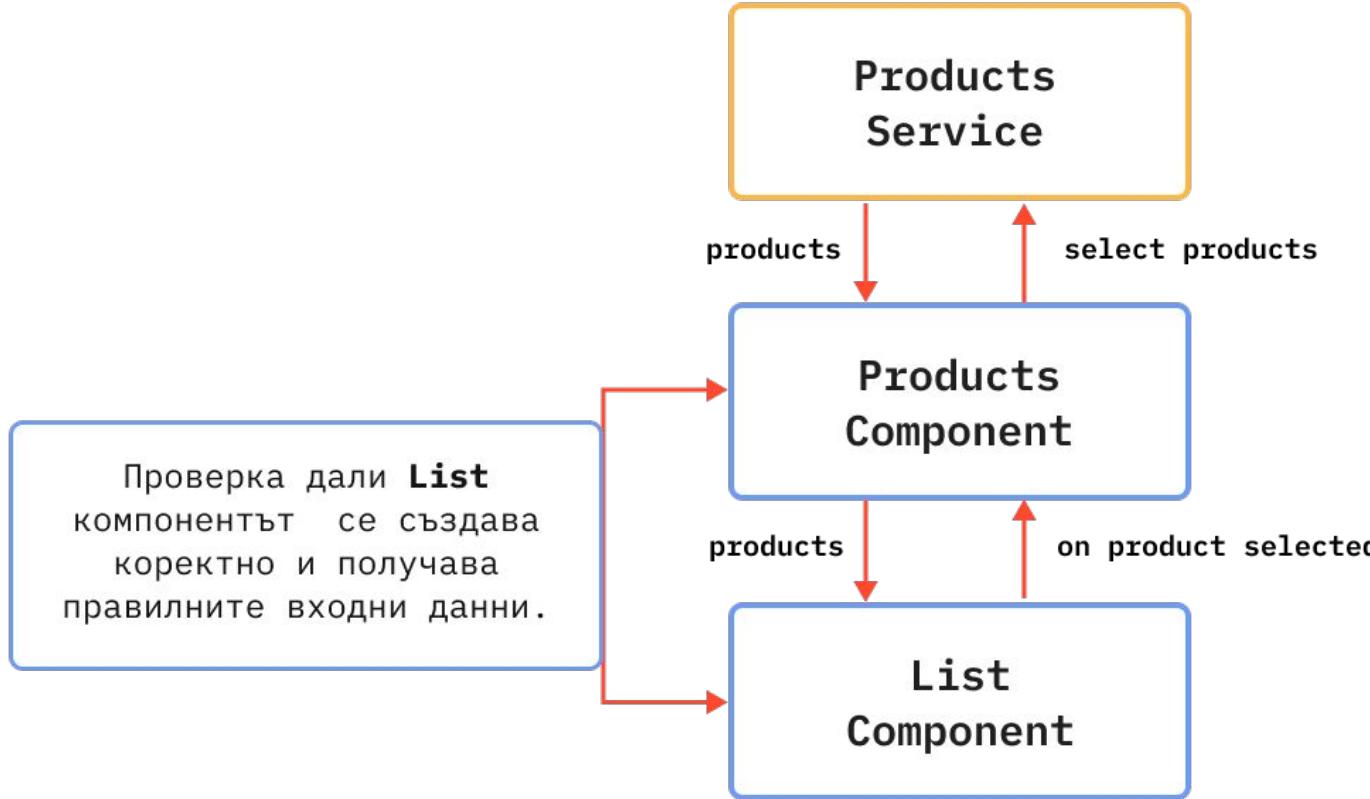
TS products.service.ts

```
@Injectable({
  providedIn: 'root',
})
export class ProductsService {
  private products = signal<Product[]>([[]]);
  private selectedProduct = signal<Product[]>([[]]);

  constructor() {}

  getProducts() {
    return this.products();
  }

  selectProduct(productId: number) {
    const product = this.products().find((p) => p.id === productId);
    if (!product) return;
    this.selectedProduct.update((products) => [...products, product]);
  }
}
```



Подготовка за тестовете

TS component.spec.ts

```
describe('ProductsComponent', () => {
  let component: ProductsComponent;
  let fixture: ComponentFixture<ProductsComponent>;
  let mockProductsService: jasmine.SpyObj<ProductsService>;
  const mockProducts = [
    { id: 1, name: 'test', price: 10 },
    { id: 1, name: 'test', price: 10 },
  ];

  beforeEach(async () => {
    mockProductsService = jasmine.createSpyObj(['getProducts', 'selectProduct']);
    mockProductsService.getProducts.and.returnValue(mockProducts);

    await TestBed.configureTestingModule({
      imports: [ProductsComponent],
      providers: [
        {
          provide: ProductsService,
          useValue: mockProductsService,
        },
      ],
    }).compileComponents();
  });

  fixture = TestBed.createComponent(ProductsComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

Създава **mock** инстанция на service, позволявайки проследяване на извикванията на дефинираните методи.

TS component.spec.ts

```
describe('app-list component', () => {
  let list: DebugElement;

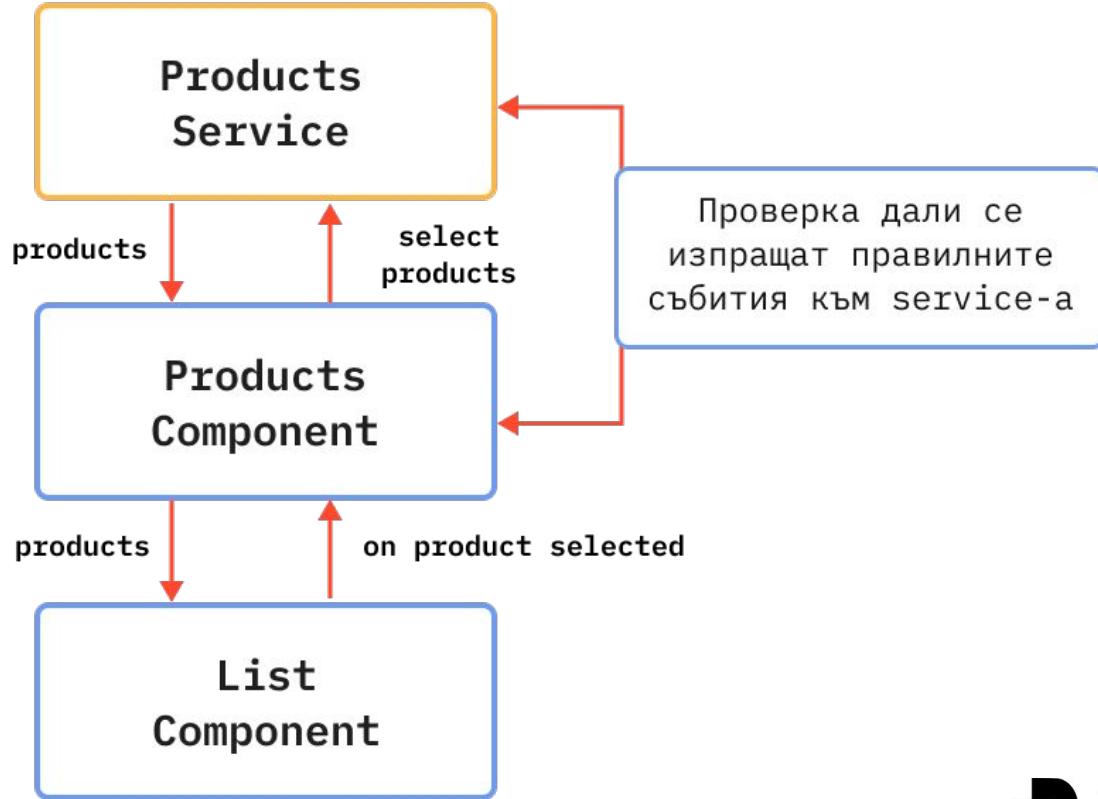
  beforeEach(() => {
    list = fixture.debugElement.query(By.css('app-list'));
  });

  it('should be created', () => {
    expect(list).toBeTruthy();
  });

  it('should list be same as products from service', () => {
    expect(list.componentInstance.products()).toEqual(mockProducts);
  });
});
```



www.diadraw.com

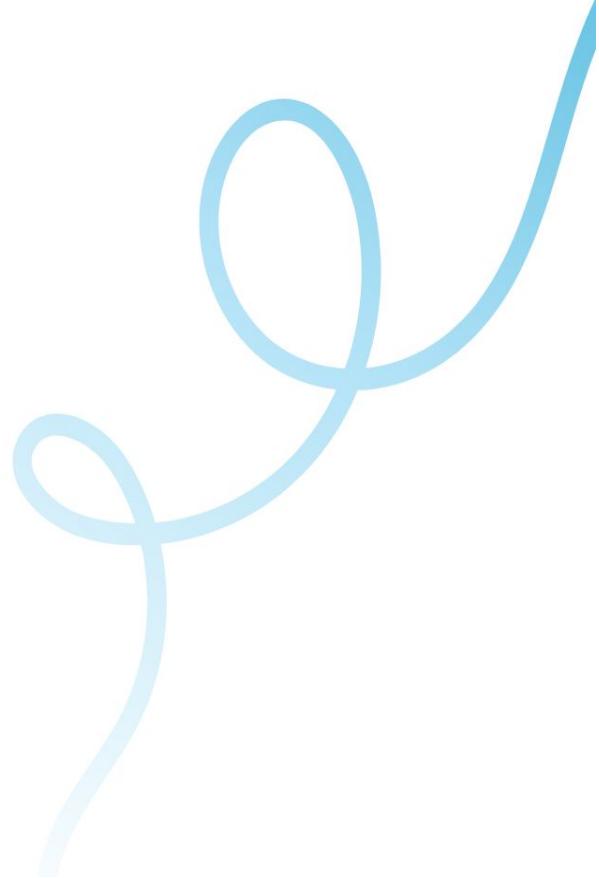


TS component.spec.ts

```
it('should emit select event', () => {
  const productId = 1;
  list.triggerEventHandler('select', productId);

  expect(mockProductsService.selectProduct).toHaveBeenCalledWith(productId);
});
```

Упражнение



End-to-End Testing

End-to-End(E2E) Testing

Метод за тестване на цялостното поведение на приложение, като се имитира истинската потребителска интеракция с него.

Целта е да се проверят всички функционалности на системата, от началото до края, и да се гарантира, че компонентите работят правилно в реална среда, като се симулират реални сценарии и потоци от действия на потребителя.

E2E Testing в Angular

Angular предоставя възможност за интеграция на външни библиотеки за изпълнение на E2E тестове. Добавянето и настройката на такава библиотека може да се извърши с помощта на командата **ng e2e** в Angular CLI.

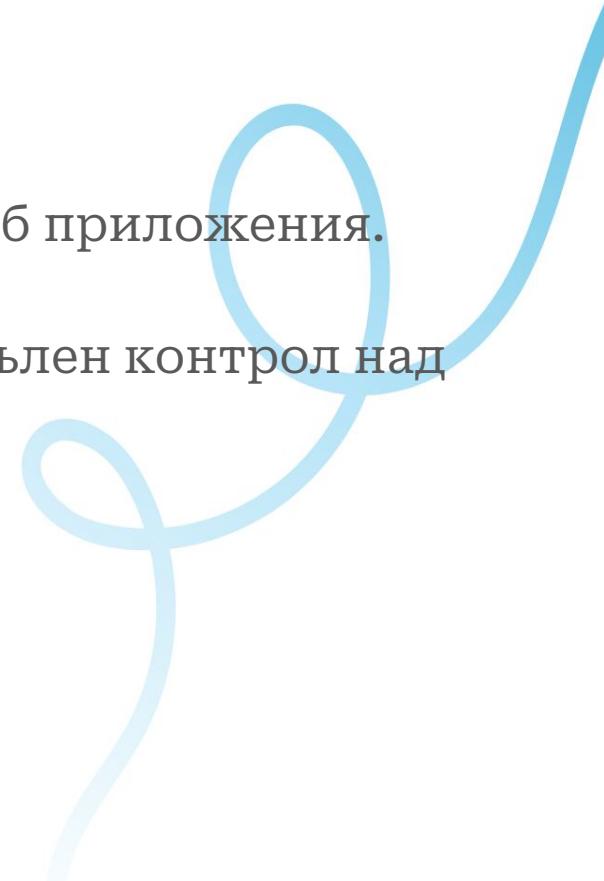
Наличните библиотеки, от които може да се избира, са:

- Cypress
- Nightwatch
- WebdriverIO
- Playwright
- Puppeteer

Cypress

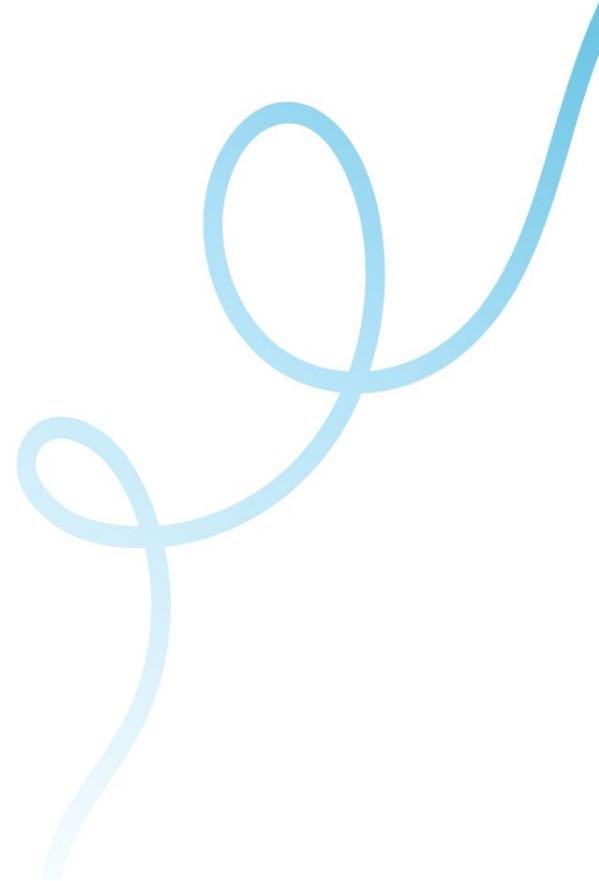
Инструмент за автоматизирано тестване на уеб приложения.

Работи директно в браузъра, предоставяйки пълен контрол над тествания процес.



Видове тестове със Cypress

- End-to-end testing
- Component testing
- Accessibility testing
- UI Coverage

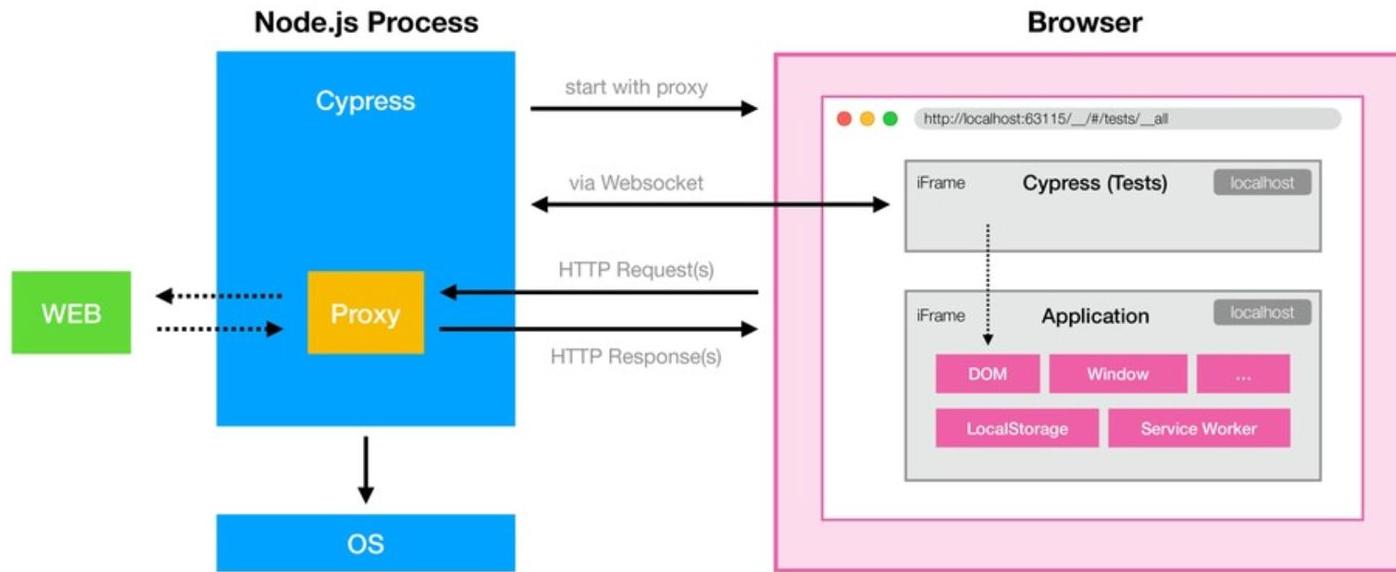


Как работи Cypress?

Cypress работи чрез Node.js сървър, който управлява тестовия процес и синхронизира действията с браузъра в реално време.

Тестовият код се инжектира директно в браузъра, което позволява на Cypress да има пълен контрол над DOM-а, мрежовите заявки и потребителските събития.

Cypress може да изпълнява задачи извън браузъра, като директно изпълнение на команди в терминала или създаване на екранни снимки и видеозаписи.



Ресурс - [блог](#)

Как работи Cypress?

Cypress използва няколко ключови библиотеки и технологии, за да предостави функционалността си за автоматизирано тестване:

- Mocha
- Chai
- Chai-jQuery
- Sinon.js
- Sinon-Chai
- jQuery



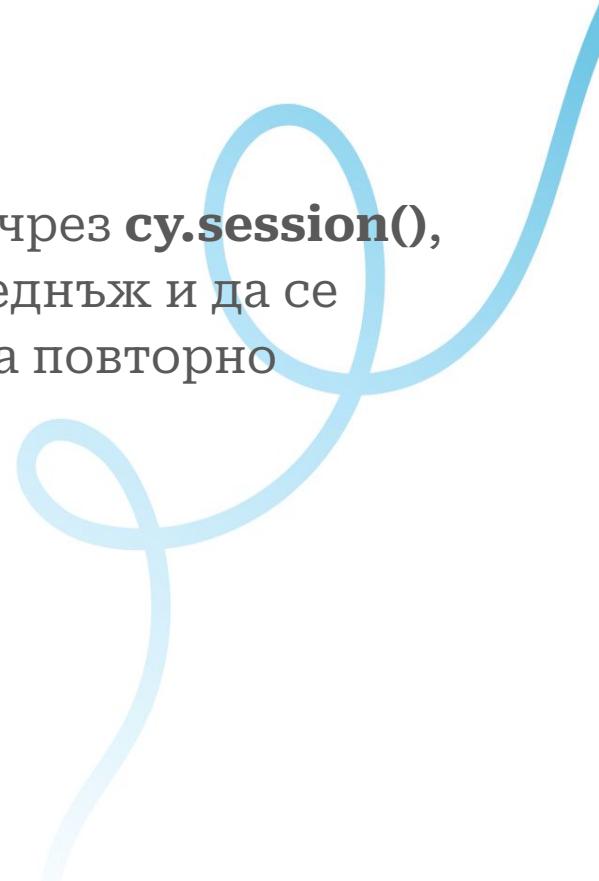
Важно!

Cypress синхронно следи всички събития в приложението, като моментално открива кога страницата се зарежда или премахва.

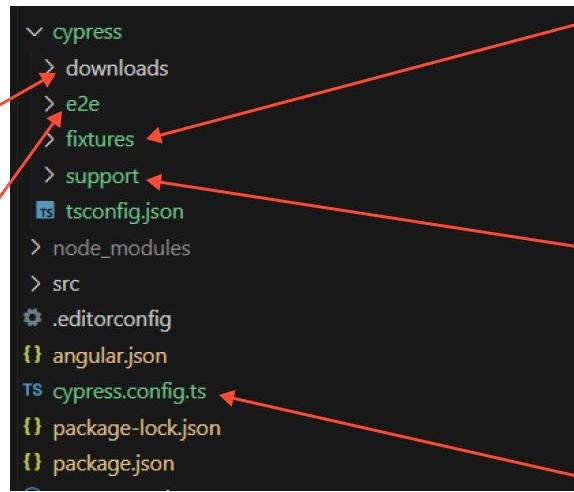
Автоматично изчаква елементи да станат видими, активни или да спрат анимацията, и спира изпълнението на команди, докато страницата или мрежовите заявки не се заредят напълно.

Важно!

Cypress позволява кеширане на браузър сесия чрез **cy.session()**, така че автентификацията да се извършва само веднъж и да се възстановява между тестовете, без да се налага повторно въвеждане на данни за вход.



Файлова структура



Файловете, изтеглени по време на тестовете

End-to-end (E2E) тестови файлове

Статични данни, които могат да се използват от тестовете.

Съхранение на **custom commands** и **global overrides**

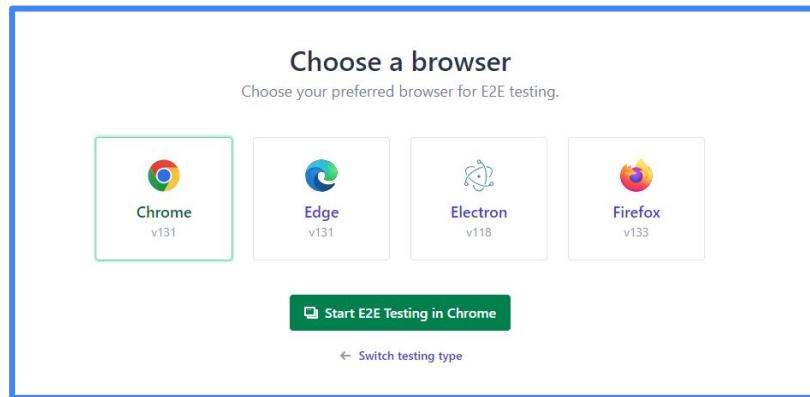
Конфигурационни настройки за Cypress

Конфигурация на Cypress

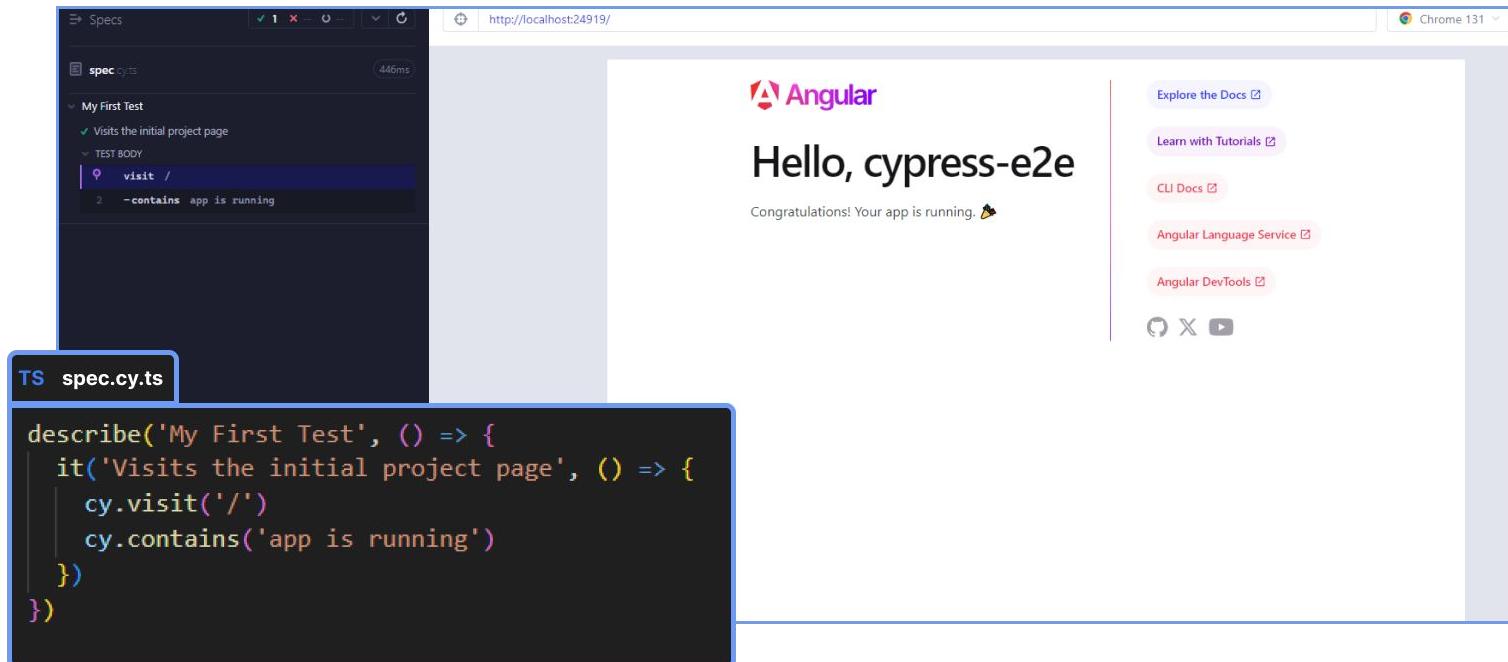
След конфигуриране на приложението за end-to-end тестване, може да се изпълни същата команда за стартиране на тестовете:

ng e2e

При стартиране на Cypress се отваря прозорец, в който се избира браузър за изпълнение на тестовете.



Изпълнение на тестове



Cypress Commands

- **visit** - Посещава даден URL адрес.
- **get** - Намира DOM елементи по селектор.
- **contains** - Намира елемент, съдържащ текст.
- **click** - Извършва клик върху елемент.
- **type** - Въвежда текст в поле за въвеждане.
- **should** - Валидира състояние или поведение на елемент.
- **url** - Проверява текущия URL.
- **intercept** - Прихваща и манипулира мрежови заявки.
- **wait** - Изчаква изпълнението на действие или заявка.
- **reload** - Презарежда текущата страница.
- [други](#)

Взаимодействие с елементи

Cypress предоставя команди за взаимодействие с DOM, като **click**, **type**, **check** и други, които симулират потребителски действия.

Преди изпълнението на всяка команда, Cypress проверява състоянието на елемента, за да се увери, че е готов за взаимодействие (видим, активен, неанимиран и т.н.).

Взаимодействие с елементи

Cypress проверява видимостта на елементите чрез множество фактори, включително размери (ширина или височина), CSS свойства като **visibility: hidden**, **display: none** или **position: fixed**.

Елементи с **opacity: 0** се считат за скрити, но остават достъпни за взаимодействие.

Също така се проверява дали елементите, като бутони или полета, са маркирани като **disabled**, което блокира действията върху тях.

Работа със заявки

Cypress предоставя пълен контрол върху HTTP комуникацията.

- Проверка на тялото на заявката
- Проверка на URL адреса
- Проверка на headers
- Заместване на отговора с друго съдържание
- Заместване на статус кода
- Заместване на заглавните части на отговора
- Забавяне на отговора
- Изчакване на отговор

TS spec.cy.ts

```
describe('My First Test', () => {
  it('Visits the initial project page', () => {
    cy.intercept(
      {
        method: 'GET',
        url: '/products/*',
      },
      []
    ).as('getProducts');

    cy.visit('/');
    cy.wait('@getProducts').then((interception) => {
      assert.isNotNull(interception?.response?.body);
    });
  });
});
```

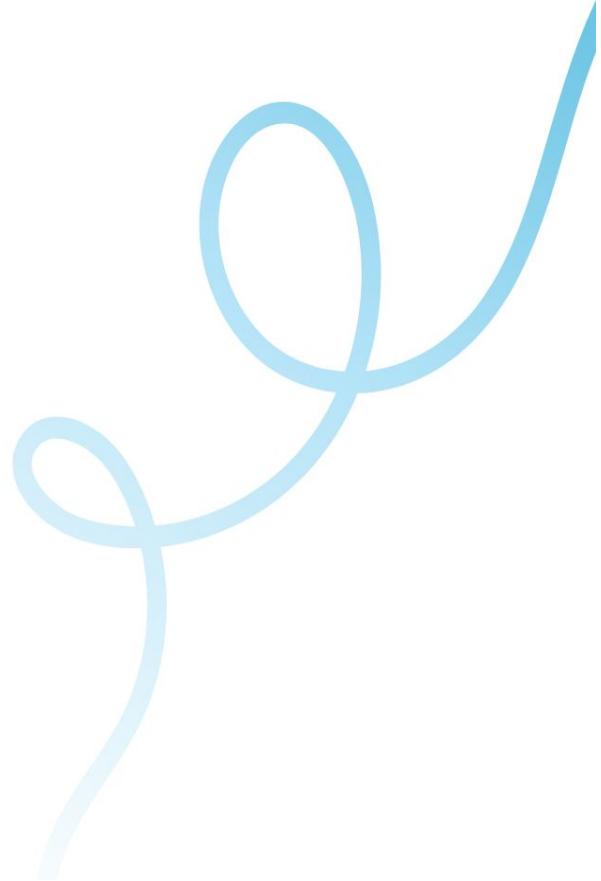
Присвояване на псевдоним на заявката

Прихващане на всички GET заявки към products/*

Очакване на отговор от заявката

Задаване на отговора от заявката да бъде []

Демо



www.diadraw.com

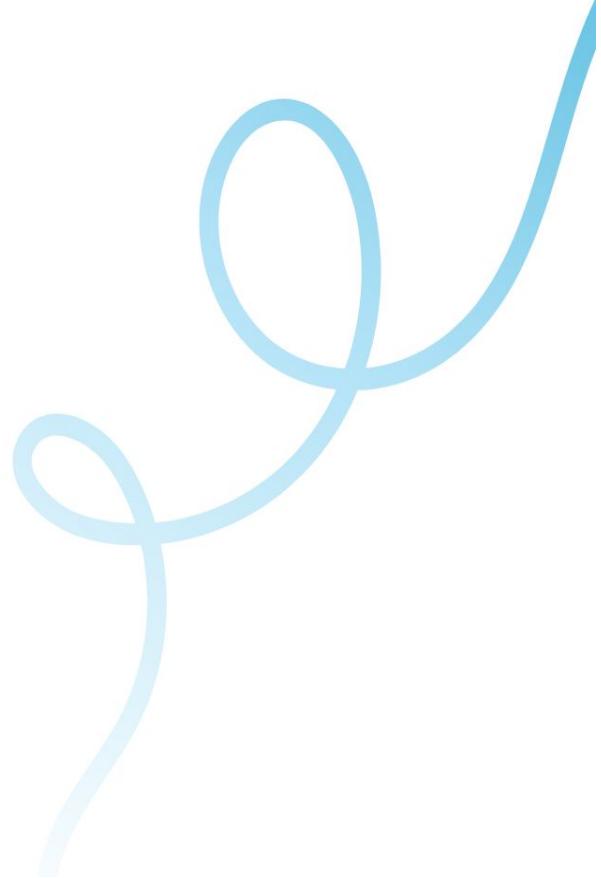
Създаване на Cypress Command

Cypress позволява създаване на commands, които позволяват лесно преизползване на тестова логика.

TS cypress/support/commands.ts

```
Cypress.Commands.add('checkToken', (token) => {
  cy.window().its('localStorage.token').should('eq', token);
});
```

Упражнение



Благодаря за вниманието!