Learn to TEST Awesome Angular Apps

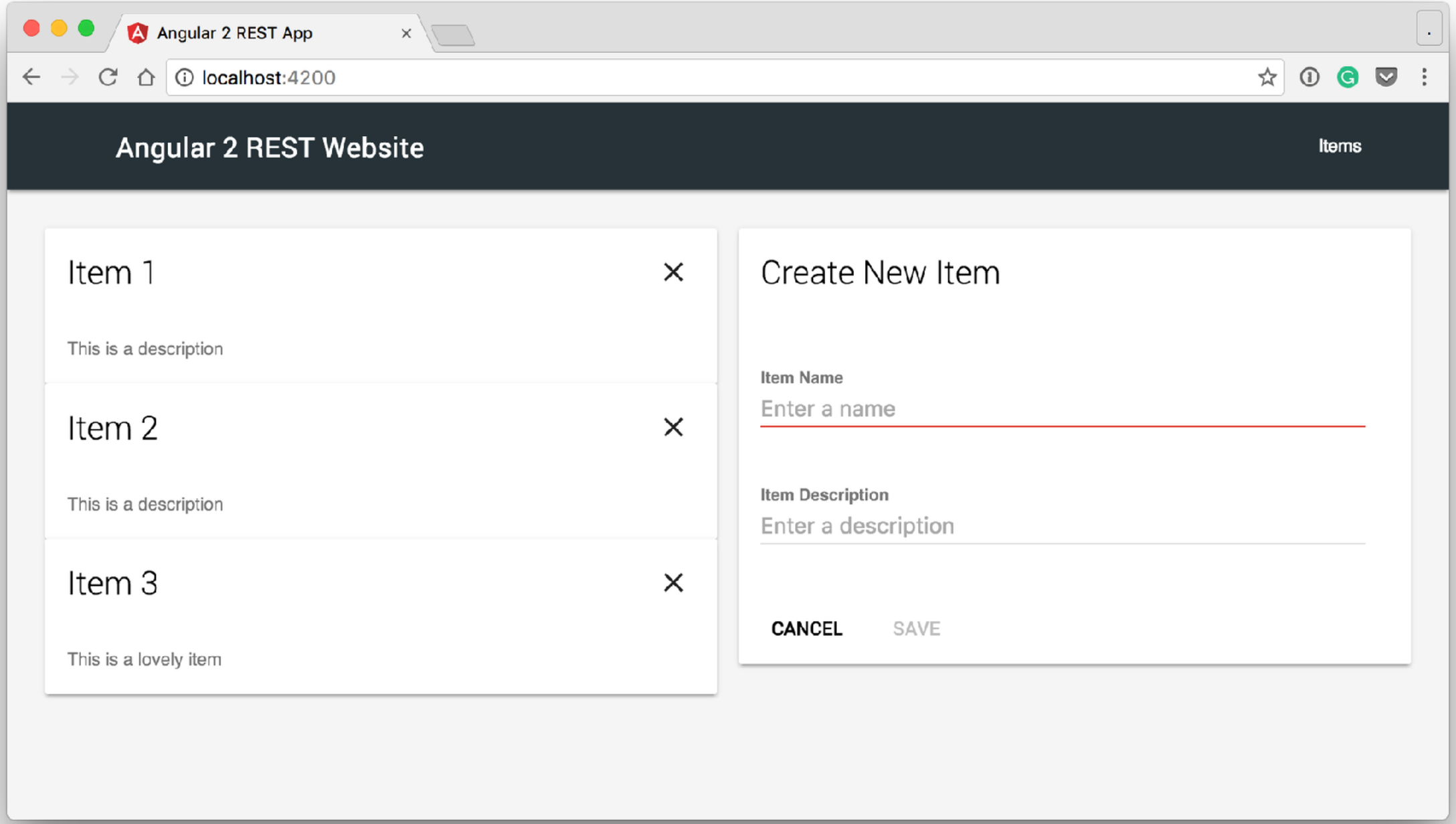
Strong grasp on the **basic patterns** necessary for **testing** an Angular application

Agenda

- The Demo Application
- The Testing Big Picture
- Your First Test
- Basic Component Test
- Component Testing Patterns
- Isolated Testing Patterns

Getting **Started**

https://github.com/onehungrymind/angular-testing-examples


Angular 2 REST App
localhost:4200
Angular 2 REST Website
Items
Item 1
This is a description
Item 2
This is a description
Item 3
This is a lovely item
Create New Item
Item Name
Enter a name
Item Description
Enter a description
CANCEL
SAVE

The Demo Application

- A simple RESTful master-detail application built using Angular and the Angular CLI
- Testing examples are in the **examples** folder
- We will be working through the examples and applying the patterns to a real feature

Challenges

- Make sure you can run the application
- Make sure you can run the tests

The **Testing** Big Picture

TESTING **IS HARD!**

WRITING
SOFTWARE
IS HARD!

The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand.

Out of the Tarpit - Ben Mosely Peter Marks

We believe that the major contributor to this complexity in many systems is the handling of state and the burden that this adds when trying to analyse and reason about the system. Other closely related contributors are code volume, and explicit concern with the flow of control through the system.

Out of the Tarpit - Ben Mosely Peter Marks

Complexity and **purgatory**

```
class ItemsComponent {
  total: number = 0;
  currentCategory: string = 'cool';
  inbound(item) {
    const newTotal: number;
    switch(this.currentCategory) {
      case 'fun':
        // calculate total based on fun factor
        break;
      case 'cool':
        // calculate total based on cool factor
        break;
      case 'dangerous':
        // calculate total based on dangerous factor
        break;
      default:
        // do nothing at all
    }
    return newTotal;
  }
}
```

```
class ItemsComponent {
  total: number = 0;
  currentCategory: string = 'cool';
  inbound(item) {
    const newTotal: number;
    switch(this.currentCategory) {
      case 'fun':
        // calculate total based on fun factor
        break;
      case 'cool':
        // calculate total based on cool factor
        break;
      case 'dangerous':
        // calculate total based on dangerous factor
        break;
      default:
        // do nothing at all
    }
    return newTotal;
  }
}
```

```
const itemsComponents = new ItemsComponent();
const myItem = {name:'My Item'};
itemsComponents.inbound(myItem); // Some result

itemsComponents.currentCategory = 'fun'; // Changing state
itemsComponents.inbound(myItem); // Same parameter but different result

itemsComponents.currentCategory = 'cool'; // Changing state
itemsComponents.inbound(myItem); // Same parameter but different result

itemsComponents.currentCategory = 'dangerous'; // Changing state
itemsComponents.inbound(myItem); // Same parameter but different result
```

```
class ItemsComponent {
  total: number = 0;
  currentCategory: string = 'cool';
  currentAgeGroup: string = 'child';
  inbound(item) {
    const newTotal: number;
    switch(this.currentCategory) {
      //...
      case 'dangerous':
        if(this.currentAgeGroup !== 'child') {
          // calculate total based on dangerous factor
          this.currentCategory = 'dangerous';
        } else {
          // calculate total based on alternate dangerous factor
        }
        break;
      default:
      // do nothing at all
    }
    return newTotal;
  }
}
```

State **management**

```
class Inventory {
  ledger = { total: 1200 };
}
class ItemsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
class WidgetsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
```

```
class Inventory {
  ledger = { total: 1200 };
}
class ItemsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
class WidgetsComponent {
  ledger: any;
  constructor(private inventory:Inventory) {
    this.ledger = inventory.ledger;
  }
  add(x) { this.ledger.total += x; }
}
```

Controlling flow

```
function doWork() {
  return $http.post('url')
    .then(function(response){
      if(response.data.success)
        return response.data;
      else
        return $q.reject('some error occurred');
    })
}
doWork().then(console.log, console.error);
```

```
var retriesCount = 0;
function doWork() {
  return $http.post('url')
    .then(function(response){
      if(response.data.success)
        return response.data;
      else
        return $q.reject('some error occurred');
    })
    .then(null, function(reason){
      if(retriesCount++ < 3)
        return doWork();
      else
        return $q.reject(reason);
    });
}
doWork().then(console.log, console.error);
```

Code **volume**

TESTING **IS HARD!**

Testing can be summarized with some **basic patterns**

Small methods are easier to test

Pure methods are easier to test

Don’t use **real** services

Don’t use **real** services,
use a **test double**

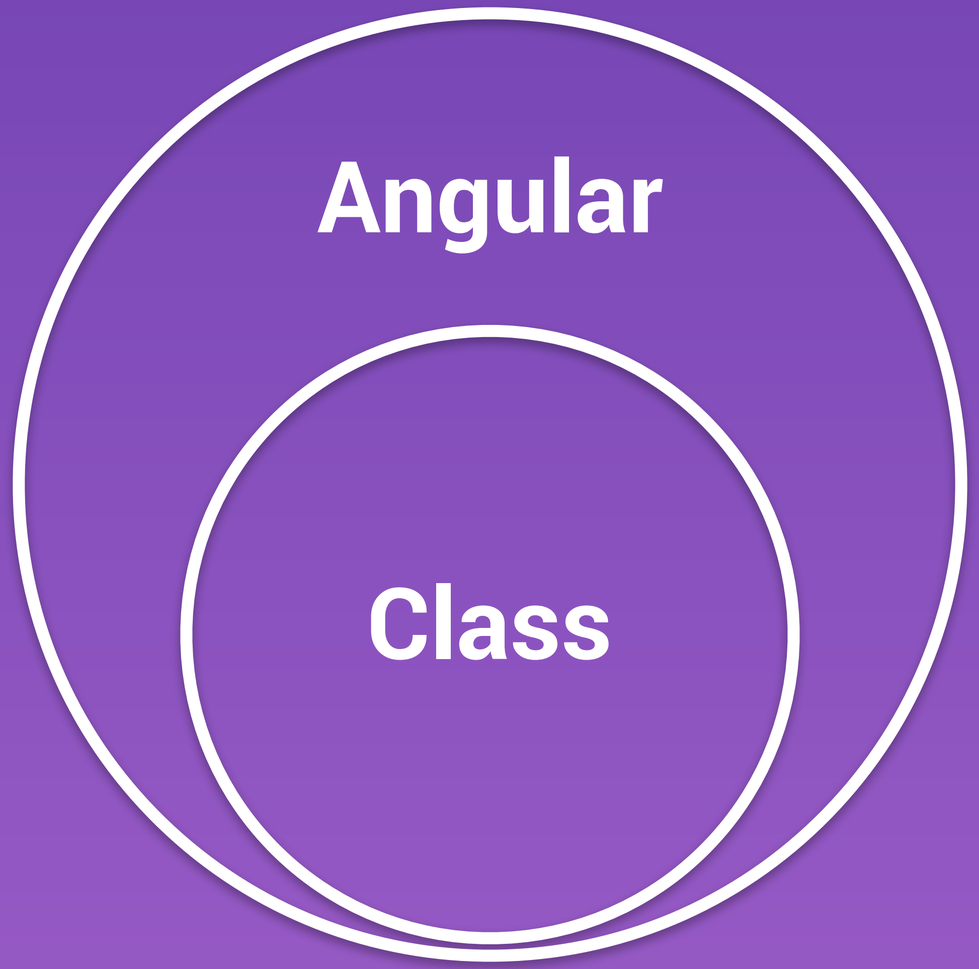
Don’t use **real** services,
use a **stub**

Don’t use **real** services,
use a **spy**

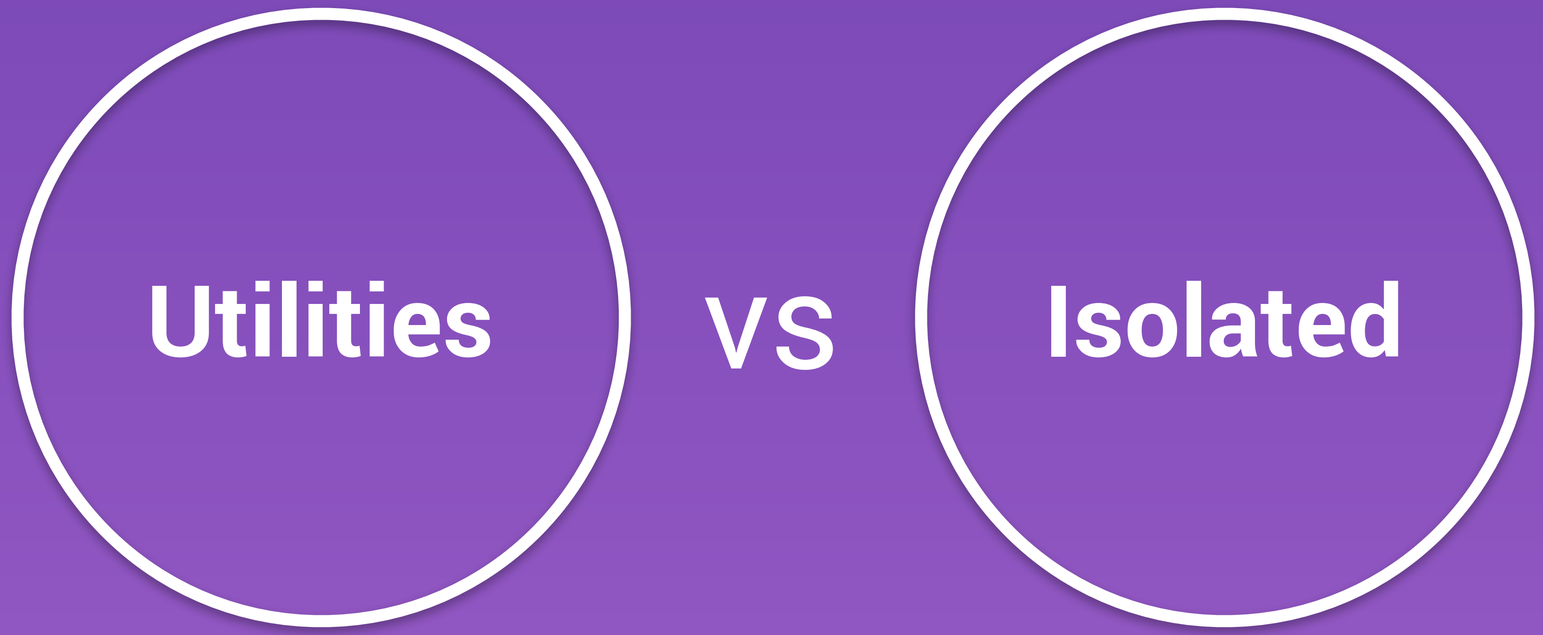
Faking and **spying** are both **great** options

Faking and **spying** are both **great** options, start with what is **easiest**

Faking and **spying** are both **great** options, you can use **both**

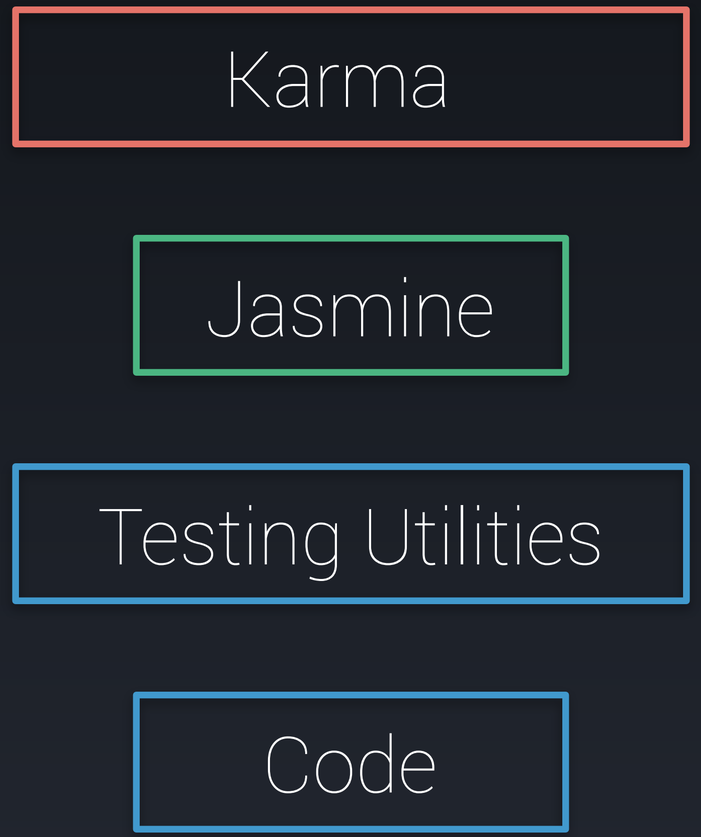


Basic Structure



Two Approaches

The Testing Big Picture



Your **First** Test

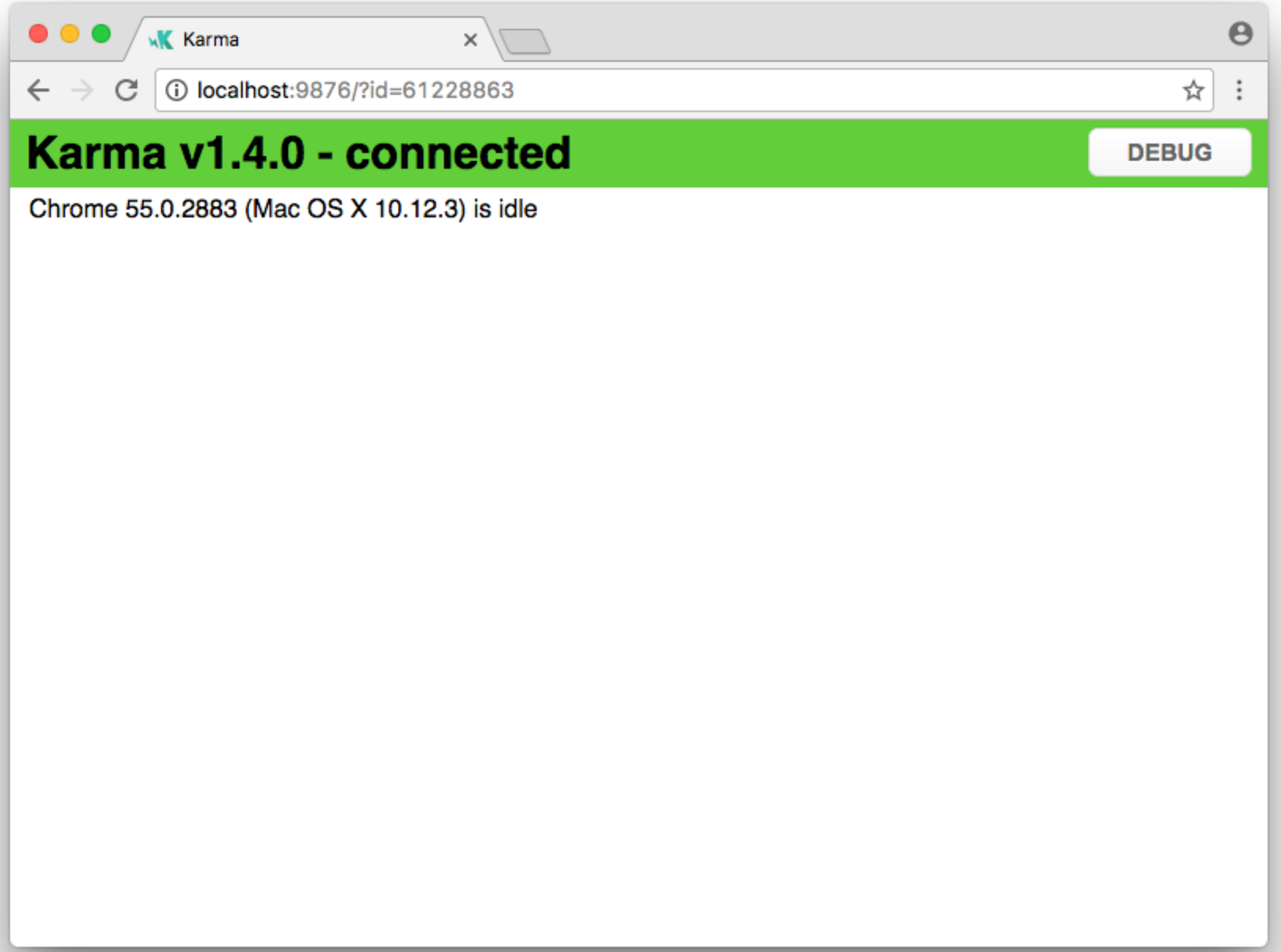
Karma

- Karma is the test runner that is used to execute Angular unit tests
- You can manually install and configure Karma
- Karma is **installed** and **configured** by default when you create a project with the Angular CLI
- Karma is configured via the **karma.conf.js** file
- Tests (specs) are identified with a **.spec.ts** naming convention

```
➜  angular-testing-examples git:(master) npm test

> ng2-simple-app@0.0.1 test /Users/lukas/Projects/angular-testing-examples
> ng test

31 01 2017 08:01:22.968:WARN [karma]: No captured browser, open http://localhost:9876/
31 01 2017 08:01:22.979:INFO [karma]: Karma v1.4.0 server started at http://0.0.0.0:9876/
31 01 2017 08:01:22.979:INFO [launcher]: Launching browser Chrome with unlimited concurrency
31 01 2017 08:01:23.009:INFO [launcher]: Starting browser Chrome
31 01 2017 08:01:23.823:INFO [Chrome 55.0.2883 (Mac OS X 10.12.3)]: Connected on socket JVLu4djT_nHtiQE-AAAA with id 31675908
Chrome 55.0.2883 (Mac OS X 10.12.3): Executed 25 of 25 SUCCESS (0.64 secs / 0.627 secs)
```


Karma
localhost:9876/?id=61228863
Karma v1.4.0 - connected
DEBUG
Chrome 55.0.2883 (Mac OS X 10.12.3) is idle

Debugging with Karma

- Use the developer console in the Karma browser window to debug your unit tests
- If something is throwing an error, you will generally see it in the console
- If you need to step through something, you can do some from a breakpoint in the developer tools
- Logging to the console is also a handy tool for observing data and events

```
describe('First spec', () => {
  it('should pass', () => {
    expect(false).toBeTruthy();
  });
});
```

Simple Test Fail

```
describe('First spec', () => {
  it('should pass', () => {
    expect(true).toBeTruthy();
  });
});
```

Simple Test Pass

Challenges

- Check out the **start** branch
- Locate the **first.spec.ts** file
- Create a test to first **fail** and then **pass**
- Execute your test

Basic **Component** Test

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-simple',
  template: '<h1>Hello {{subject}}!</h1>'
})
export class SimpleComponent implements OnInit {
  subject: string = 'world';
  constructor() { }
  ngOnInit() { }
}
```

The Component

1. Configure Module

TestBed

- The most important piece of the Angular testing utilities
- Creates an Angular testing module which is an **@NgModule** class
- You can configure the module by calling **TestBed.configureTestingModule**
- Configure the testing module in the **BeforeEach** so that it gets reset before each spec

```
import { TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

describe('SimpleComponent', () => {
  let component: SimpleComponent;
  let fixture: any;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    });
  });
});
```

Configure Module

1. Configure Module
2. Create Fixture

TestBed.createComponent

- Creates an instance of the component under test
- Returns a component test fixture
- Calling **createComponent** closes the **TestBed** from further configuration

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

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

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);
  });
});
```

The Fixture

1. Configure Module
2. Create Fixture
3. Get Component Instance

ComponentFixture

- Handle to the test environment surrounding the component
- Provides access to the component itself via **fixture.componentInstance**
- Provides access to the **DebugElement** which is a handle to the component's DOM element
- **DebugElement.query** allows us to query the DOM of the element
- **By.css** allows us to construct our query using CSS selectors

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

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

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);

    component = fixture.componentInstance;
  });
});
```

The Component Instance

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { SimpleComponent } from './simple.component';

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

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);

    component = fixture.componentInstance;
  });

  it('sets the `subject` class member', () => {
    expect(component.subject).toBe('world');
  });
});
```

The Component Instance

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';
import { SimpleComponent } from './simple.component';

describe('SimpleComponent', () => {
  let component: SimpleComponent;
  let fixture: ComponentFixture<SimpleComponent>;
  let de: DebugElement;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ SimpleComponent ]
    })
    .createComponent(SimpleComponent);

    component = fixture.componentInstance;
    de = fixture.debugElement;
    fixture.detectChanges();
  });

  it('greets the subject', () => {
    const h1 = de.query(By.css('h1'));
    expect(h1.nativeElement.innerText).toBe('Hello world!');
  });
});
```

ComponentFixture.detectChanges

- We tell Angular to perform change detection by calling **ComponentFixture.detectChanges**
- **TestBed.createComponent** does not automatically trigger a change detection
- This is intentional as it gives us greater control over how we inspect our components pre-binding and post-binding

```
it('greets the subject', () => {
  const h1 = de.query(By.css('h1'));
  expect(h1.nativeElement.innerText).toBe('Hello world!');
});

it('updates the subject', () => {
  component.subject = 'developer';
  fixture.detectChanges();
  const h1 = de.query(By.css('h1'));
  expect(h1.nativeElement.innerText).toBe('Hello developer!');
});
```

detectChanges

Challenges

- Locate the **simple.component.spec.ts** file
- Configure the component's **module**
- Initialize the component's **fixture**
- Get a reference to the actual component
- Write a test against the component
- Create a reference to the component's **debugElement**
- Write a test against the **debugElement**
- Write a test that uses **detectChanges**

Component Testing
Patterns

Component with an External Template

- With an external template, Angular needs to read the file before it can create a component instance. This is problematic because **TestBed.createComponent** is **synchronous**.
- The first thing we do is break our initial **beforeEach** into an **asynchronous beforeEach** call and a **synchronous beforeEach** call
- We then use the **async** testing utility to load our external templates
- And then call **TestBed.compileComponents** to compile our components
- **WebPack users can skip this slide**

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

async

```
beforeEach(() => {
  fixture = TestBed.createComponent(TemplateComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

fixture

Challenges

- Locate the **template.component.spec.ts** file
- Set up the **asynchronous** and **synchronous beforeEach** blocks
- Test that the component exists

Component with a Service Dependency

- Components do not need to be injected with real services
- Use test doubles to stand in for the real service since we are testing the component and not the service
- We can override the provider with **useValue** or **useClass** and provide our custom test double
- Use **debugElement.injector** to get a reference to the service from the component's injector

```
@Component({
  selector: 'app-service',
  template: '<h1>Hello {{subject.name}}!</h1>'
})
export class ServiceComponent implements OnInit {
  subject: {name: string} = this.service.subject;
  constructor(private service: GreetingService) { }
  ngOnInit() { }
}
```

Component

```
export class GreetingService {
  subject: {name: string} = { name: 'world' };
}
```

Service

```
describe('ServiceComponent', () => {
  let component: ServiceComponent;
  let fixture: ComponentFixture<ServiceComponent>;
  let de: DebugElement;
  let greetingServiceStub;
  let greetingService;
});
```

Local Members

```
beforeEach(() => {
  greetingServiceStub = {
    subject: {name: 'world'},
  };

  fixture = TestBed.configureTestingModule({
    declarations: [ ServiceComponent ],
    providers: [{ provide: GreetingService, useValue: greetingServiceStub }]
  })
  .createComponent(ServiceComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();

  greetingService = de.injector.get(GreetingService);
});
```

Test Double

```
beforeEach(() => {
  greetingServiceStub = {
    subject: {name: 'world'},
  };

  fixture = TestBed.configureTestingModule({
    declarations: [ ServiceComponent ],
    providers: [{ provide: GreetingService, useValue: greetingServiceStub }]
  })
  .createComponent(ServiceComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();

  greetingService = de.injector.get(GreetingService);
});
```

Test Double

```
beforeEach(() => {
  greetingServiceStub = {
    subject: {name: 'world'},
  };

  fixture = TestBed.configureTestingModule({
    declarations: [ ServiceComponent ],
    providers: [{ provide: GreetingService, useValue: greetingServiceStub }]
  })
  .createComponent(ServiceComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();

  greetingService = de.injector.get(GreetingService);
});
```

Test Double

```
it('updates component subject when service subject is changed', () => {
  greetingService.subject.name = 'cosmos';
  fixture.detectChanges();
  expect(component.subject.name).toBe('cosmos');
  const h1 = de.query(By.css('h1')).nativeElement;
  expect(h1.innerText).toBe('Hello cosmos!');
});
```

Actual Test

Challenges

- Locate the **service.component.spec.ts** file
- Set up the basic structure of the spec file
- Create a stub for the **GreetingService**
- Override the provider to use the **greetingService** stub instead of the real service
- Write a test verifying that when you update a property on the **greetingService** stub that the component updates correctly

Component with an Async Service

- We should emulate remote server calls and not actually make them
- Emulating server calls is easy with **Jasmine spies**
- A **spy** intercepts a call to a method and performs some custom operation such as returning a hardcoded value
- An asynchronous test must wait at least one full turn of the JavaScript engine for a value to return
- We can use the **async** with **fixture.whenStable**, **fakeAsync** with **tick** or **jasmine.done** to accomplish this asynchronous behavior

```
export class AsyncServiceComponent implements OnInit {
  greeting: string;
  subject: string;
  punctuation: string;
  constructor(private service: GreetingService) { }

  ngOnInit() {
    this.service.getGreeting()
      .then(res => this.greeting = res);
    this.service.getSubject()
      .then(res => this.subject = res);
    this.service.getPunctuation()
      .then(res => this.punctuation = res);
  }
}
```

Component

```
@Injectable()
export class GreetingService {
  subject: {name: string} = { name: 'world' };
  getGreeting() { return Promise.resolve('Hello'); }
  getSubject() { return Promise.resolve(this.subject.name); }
  getPunctuation() { return Promise.resolve('!'); }
}
```

Service

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ AsyncServiceComponent ],
    providers: [ GreetingService ]
  })
  .createComponent(AsyncServiceComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement.query(By.css('h1'));
  greetingService = de.injector.get(GreetingService);
});
```

Setup

```
it('should ensure `greeting`, `subject`, or `punctuation` are initially undefined', () => {
  fixture.detectChanges();
  expect(component.greeting).toBeUndefined();
  expect(component.subject).toBeUndefined();
  expect(component.punctuation).toBeUndefined();
});
```

Setup

```
it('gets `greeting` after promise (async)', async(() => {
  spyOn(greetingService, 'getGreeting')
    .and.returnValue(Promise.resolve('Greetings'));

  fixture.detectChanges();

  fixture.whenStable().then(() => {
    fixture.detectChanges();
    expect(component.greeting).toBe('Greetings');
  });
}));
```

Spy with fixture.whenStable

```
it('gets `subject` after promise (fakeAsync)', fakeAsync(() => {
  spyOn(greetingService, 'getSubject')
    .and.returnValue(Promise.resolve('universe'));

  fixture.detectChanges();
  tick();
  fixture.detectChanges();
  expect(component.subject).toBe('universe');
}));
```

Spy with tick

```
it('gets `punctuation` after promise (done) - use with caution', done => {
  spyOn(greetingService, 'getPunctuation')
    .and.returnValue(Promise.resolve(' :)'));

  fixture.detectChanges();
  greetingService.getPunctuation().then(() => {
    fixture.detectChanges();
    expect(component.punctuation).toBe(' :)');
    done();
  });
});
```

Spy with done

Challenges

- Locate the **async-service.component.spec.ts** file
- Set up the basic structure of the spec file
- Create an **async** test that uses a **spy** to intercept a **GreetingService** method and then verifies that the component responds correctly.
- Create a **fakeAsync** test that uses a **spy** to intercept a **GreetingService** method and then verifies that the component responds correctly.

Component with Inputs and Outputs

- We can test a component with inputs and outputs as a standalone component or within a host component
- The goal is to ensure that binding works as expected
- This is fairly easy thanks to Angular testing utilities which allows us to verify a component's visual representation
- To test an input, we simple update the value and ensure that it renders
- To test an output, we can trigger the output using **triggerEventHandler** and subscribing to the **EventEmitter**

```
@Component({
  selector: 'app-input-output',
  template: `
    <h1>Hello {{subject}}!</h1>
    <button (click)="depart()">We Out</button>
  `
})
export class InputOutputComponent {
  @Input('subject') subject: string;
  @Output('leave') leave: EventEmitter<string> = new EventEmitter();
  depart() {
    this.leave.emit(`Ciao ${this.subject}!`);
  }
}
```

Component

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ InputOutputComponent ]
  })
  .createComponent(InputOutputComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  button = de.query(By.css('button'));

  component.subject = 'galaxy';
  fixture.detectChanges();
});
```

Setup

```
it('has `subject` as an @Input', () => {
  expect(component.subject).toBe('galaxy');
});
```

Input

```
it('says goodbye to the `subject`', () => {
  let farewell;
  component.leave.subscribe(event => farewell = event);

  button.triggerEventHandler('click', null);
  expect(farewell).toBe('Ciao galaxy!');
});
```

Output

Challenges

- Locate the **input-output.component.spec.ts** file
- Set up the basic structure of the spec file
- Create a test to verify that the **input** binding is working
- Create a test to verify that the **output** binding is working

Component Inside a Host Component

- We can also create a host component to verify that our component works properly as a subcomponent
- The main difference is that we call **TestBed.createComponent** on the host component and not the component under test
- We can then query the host component to get a reference to the component under test
- These references allows us to verify our bindings just like we would if we were testing as a stand alone component

```
@Component({
  template: `
  <app-input-output
    [subject]="subject"
    (leave)="onLeave($event)">
  </app-input-output>
  `
})
class TestInputOutputHostComponent {
  subject: string = 'galaxy';
  completeGreeting: string;
  onLeave(greeting: string) { this.completeGreeting = greeting; }
}
```

Host Component

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ InputOutputComponent, TestInputOutputHostComponent ]
  })
  .createComponent(TestInputOutputHostComponent);

  component = fixture.componentInstance;
  de = fixture.debugElement;
  button = de.query(By.css('button'));
  h1 = de.query(By.css('h1'));
  fixture.detectChanges();
});
```

Setup

```
it('greets the @Input `subject`', () => {
  expect(h1.nativeElement.innerText).toBe('Hello galaxy!');
});
```

Input

```
it('says goodbye to the `subject`', () => {
  button.triggerEventHandler('click', null);
  expect(component.completeGreeting).toBe('Ciao galaxy!');
});
```

Output

Challenges

- Locate the **input-output-with-host.component.spec.ts** file
- Set up the basic structure of the spec file
- Create a test to verify that the **input** binding is working from the host
- Create a test to verify that the **output** binding is working from the host

Routed Component

- The router complexity is generally avoided as we are testing the component and not the router
- Testing that a component navigates to a proper route is important to us
- It is easy to override the router with a simple router stub
- We can also stub out the activated route and using a **BehaviorSubject** to emulate route parameters

```
const routes: Routes = [
  {path: '',      redirectTo: '/items', pathMatch: 'full' },
  {path: 'items', component: ItemsComponent},
  {path: 'routed/:subject', component: RoutedComponent},
  {path: 'widgets', component: WidgetsComponent},
  {path: '**',     redirectTo: '/items', pathMatch: 'full'}
];
```

Routes

```
export class RoutedComponent implements OnInit{
  subject: string;
  constructor(
    private router: Router,
    private route: ActivatedRoute
  ) { }

  ngOnInit() {
    this.route.params
      .map(p => p && p['subject'])
      .forEach(subject => this.subject = subject);
  }

  goToItems() {
    this.router.navigateByUrl('/items');
  }
}
```

Component

```
class RouterStub {
  navigateByUrl(url) { return url; }
}
```

Router Stub

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs/BehaviorSubject';

@Injectable()
export class ActivatedRouteStub {
  private subject = new BehaviorSubject({subject: 'planet'});
  params = this.subject.asObservable();
}
```

Activated Route Stub

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ RoutedComponent ],
    providers: [
      { provide: Router, useClass: RouterStub },
      { provide: ActivatedRoute, useClass: ActivatedRouteStub }
    ]
  })
  .createComponent(RoutedComponent);

  component = fixture.componentInstance;
  router = fixture.debugElement.injector.get(Router);
  activatedRoute = fixture.debugElement.injector.get(ActivatedRoute);

  fixture.detectChanges();
});
```

Setup

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ RoutedComponent ],
    providers: [
      { provide: Router, useClass: RouterStub },
      { provide: ActivatedRoute, useClass: ActivatedRouteStub }
    ]
  })
  .createComponent(RoutedComponent);

  component = fixture.componentInstance;
  router = fixture.debugElement.injector.get(Router);
  activatedRoute = fixture.debugElement.injector.get(ActivatedRoute);

  fixture.detectChanges();
});
```

Setup

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ RoutedComponent ],
    providers: [
      { provide: Router, useClass: RouterStub },
      { provide: ActivatedRoute, useClass: ActivatedRouteStub }
    ]
  })
  .createComponent(RoutedComponent);

  component = fixture.componentInstance;
  router = fixture.debugElement.injector.get(Router);
  activatedRoute = fixture.debugElement.injector.get(ActivatedRoute);

  fixture.detectChanges();
});
```

Setup

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ RoutedComponent ],
    providers: [
      { provide: Router, useClass: RouterStub },
      { provide: ActivatedRoute, useClass: ActivatedRouteStub }
    ]
  })
  .createComponent(RoutedComponent);

  component = fixture.componentInstance;
  router = fixture.debugElement.injector.get(Router);
  activatedRoute = fixture.debugElement.injector.get(ActivatedRoute);

  fixture.detectChanges();
});
```

Setup

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ RoutedComponent ],
    providers: [
      { provide: Router, useClass: RouterStub },
      { provide: ActivatedRoute, useClass: ActivatedRouteStub }
    ]
  })
  .createComponent(RoutedComponent);

  component = fixture.componentInstance;
  router = fixture.debugElement.injector.get(Router);
  activatedRoute = fixture.debugElement.injector.get(ActivatedRoute);

  fixture.detectChanges();
});
```

Setup

```
it('#goToItems navigates to `/items`', () => {
  spyOn(router, 'navigateByUrl');
  component.goToItems();
  expect(router.navigateByUrl).toHaveBeenCalledWith('/items');
});
```

```
it('sets the `subject` based on route parameters', () => {
  expect(component.subject).toBe('planet');
});
```

Route Params

Challenges

- Locate the **routed.component.spec.ts** file
- Set up the basic structure of the spec file
- Create a **stub** for **Router**
- Create a test to that the component calls **router.navigateByUrl** properly
- Create a **stub** for **ActivatedRoute**
- Create a test to verify the component is properly handling the **route parameter**

Isolated Testing

Patterns

Isolated tests are
simpler because we are
only testing the **class**

We will still use
test doubles as our
primary approach to
dependencies

```
const pipe = new ExclaimPipe();
const service = new GreetingService();

let component: SimpleComponent;
beforeEach(() => component = new SimpleComponent());
```

Class

```
describe('GreetingService', () => {
  let service = new GreetingService();

  it('#getGreeting returns a `greeting`', done => {
    service.getGreeting().then((res) => {
      expect(res).toBe('Hello');
      done();
    });
  });
});
```

Isolated Service

```
describe('ExclaimPipe', () => {
  const pipe = new ExclaimPipe();
  it('adds an exclamation mark to input', () => {
    expect(pipe.transform('Hello universum magna'))
      .toBeTruthy('Hello universum magna!');
  });
});
```

Isolated Pipe

```
describe('SimpleComponent', () => {
  let component: SimpleComponent;

  beforeEach(() => component = new SimpleComponent());

  it('sets the `subject` class member', () => {
    expect(component.subject).toBe('world');
  });
});
```

Isolated Component

Challenges

- Locate the **exclaim.pipe.spec.ts** file
- Write an isolated test for the pipe
- Locate the **greeting.service.spec.ts** file
- Write an isolated test for the service
- Locate the **simple.component.isolated.spec.ts**
- Write an isolated test for the component

@simpulton

Thanks!