ULTIMATE ANGULAR FIELD GUIDE

How to TDD in angular:
The definitive guide to angular TDD

SHALLOW TESTING OVERVIEW

- When it comes to angular TDD, we can separate unit testing from IT testing. In a typical 'ng g c demo', it creates an HTML, CSS, .spec, and a TS file. By default this .spec file is an IT test since it uses Testbed. Testbed generates the page from the HTML level and tests it against the TS; including CSS. However, we can make a new file and call it .shallow.spec and put the TestBed into that file while keeping our regular .spec file purely for Unit Testing (TS testing only.) This guide will focus on Shallow Testing while covering other advanced topics such as forkjoins, services, sending HTTP requests, @Output and @Input.

BRIEF TDD OVERVIEW

- For testing purposes, every "describe" can be changed into a "fdescribe", and "it" can be changed into a "fit" (think of it as first, forced, or focused). Angular TDD will run all "fdescribes" and "fits" while ignoring everything else. This is only used for testing, do not push up to develop with "fits" or "fdescribes".
- If you wish to ignore a test or describe, simply put an "x" in front of the test or describe. (xit/xdescribe)

RXJS OVERVIEW

- When it comes to RXJS we typically use Promises and Observables. Promises handle a single event and are basically a placeholder for a future value that we are waiting on. While Observables can handle any number of events. Observables are a stream of data waiting for something to pass through (aka a response). A way to think about Observables is to imagine "Youtube", you can subscribe to a channel and when that channel posts a new video (sends something through the stream) it notifies everyone who subscribed to it. Observables work in the same ways you send a call to the backend and subscribe to the result. Once it gets a response back from the backend (something passes through our stream), it then processes its block of code. Furthermore, both are async and wait before executing their block of code.

GIT SETUP

- To begin setup for this guide, go to the following two links and clone the repo's.
- https://git.unreleased.work/enablement/comprehensive-Angular-TDD-Guide
- https://git.unreleased.work/enablement/enablementorium-backend
- The first link includes all the branches needed to follow this guide. The second link is the backend if you want to play the game yourself. To play the game, just run both applications locally, for the backend open up with gradle and run "./gradlew clean bJ bR", and for the frontend, start with the command "npm i" then run the command "npm start". All the commands need to run in the terminal. Enjoy!

ANGULAR DIRECTORY

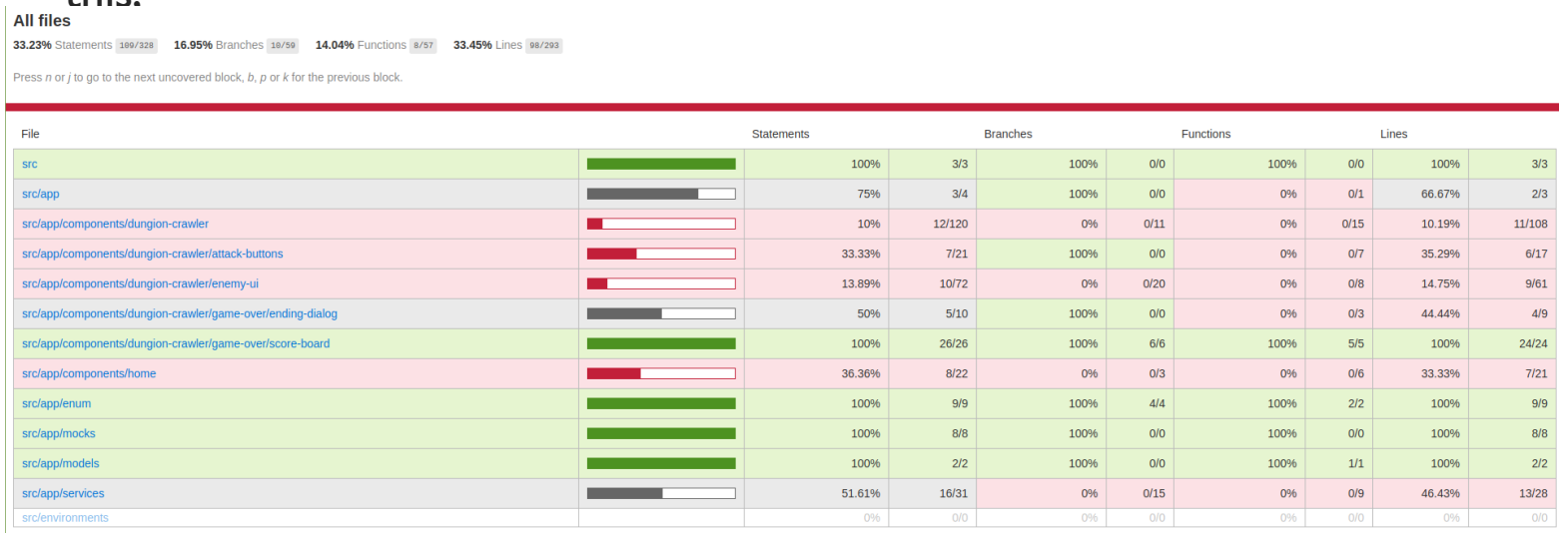
Table of Contents

CODE COVERAGE

1. During this guide, we can check our code coverage at any point. Instead of running the cli (command line interface) "ng test", we can run the following command;

```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ ng test --code-coverage
```

2. During any particular section in this guide, the code coverage should look like this.



All files

33.23% Statements `109/328` **16.95%** Branches `10/59` **14.04%** Functions `8/57` **33.45%** Lines `98/293`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

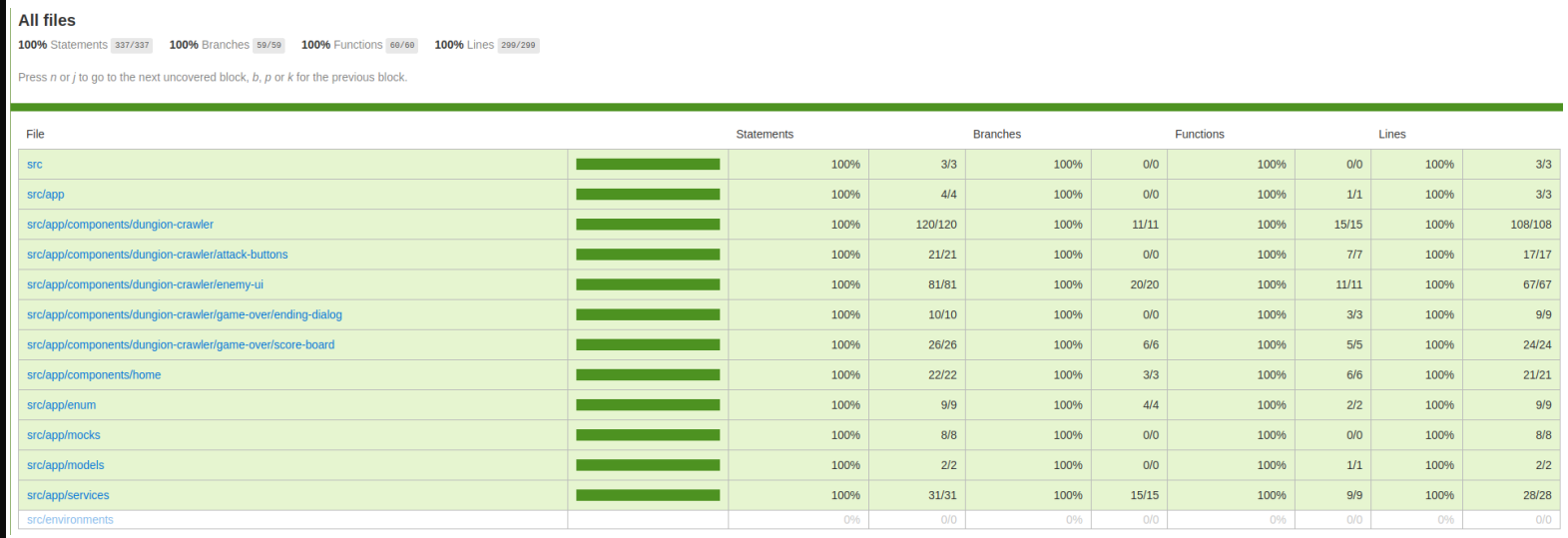
File		Statements		Branches		Functions		Lines	
src		100%	3/3	100%	0/0	100%	0/0	100%	3/3
src/app		75%	3/4	100%	0/0	0%	0/1	66.67%	2/3
src/app/components/dungion-crawler		10%	12/120	0%	0/11	0%	0/15	10.19%	11/108
src/app/components/dungion-crawler/attack-buttons		33.33%	7/21	100%	0/0	0%	0/7	35.29%	6/17
src/app/components/dungion-crawler/enemy-ui		13.89%	10/72	0%	0/20	0%	0/8	14.75%	9/61
src/app/components/dungion-crawler/game-over/ending-dialog		50%	5/10	100%	0/0	0%	0/3	44.44%	4/9
src/app/components/dungion-crawler/game-over/score-board		100%	26/26	100%	6/6	100%	5/5	100%	24/24
src/app/components/home		36.36%	8/22	0%	0/3	0%	0/6	33.33%	7/21
src/app/enum		100%	9/9	100%	4/4	100%	2/2	100%	9/9
src/app/mocks		100%	8/8	100%	0/0	100%	0/0	100%	8/8
src/app/models		100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/app/services		51.61%	16/31	0%	0/15	0%	0/9	46.43%	13/28
src/environments		0%	0/0	0%	0/0	0%	0/0	0%	0/0

The important thing to notice here, only look at the particular component or service we're currently working on. In the case above, that was from finishing scoreboard component.

Notice, to view this file in your browser, once the command has been ran, refresh your directory and open up the "coverage" folder. Inside of that folder theres a file "index.html". Right click that file and click "Copy Path". Paste that path into your browser.

CODE COVERAGE CONTINUED

3. At the end of this guide, our code coverage should be exact to the one below;



All files

100% Statements `337/337` **100%** Branches `59/59` **100%** Functions `60/60` **100%** Lines `299/299`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
src	100%	3/3	100%	0/0	100%	0/0	100%	3/3
src/app	100%	4/4	100%	0/0	100%	1/1	100%	3/3
src/app/components/dungion-crawler	100%	120/120	100%	11/11	100%	15/15	100%	108/108
src/app/components/dungion-crawler/attack-buttons	100%	21/21	100%	0/0	100%	7/7	100%	17/17
src/app/components/dungion-crawler/enemy-ui	100%	81/81	100%	20/20	100%	11/11	100%	67/67
src/app/components/dungion-crawler/game-over/ending-dialog	100%	10/10	100%	0/0	100%	3/3	100%	9/9
src/app/components/dungion-crawler/game-over/score-board	100%	26/26	100%	6/6	100%	5/5	100%	24/24
src/app/components/home	100%	22/22	100%	3/3	100%	6/6	100%	21/21
src/app/enum	100%	9/9	100%	4/4	100%	2/2	100%	9/9
src/app/mocks	100%	8/8	100%	0/0	100%	0/0	100%	8/8
src/app/models	100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/app/services	100%	31/31	100%	15/15	100%	9/9	100%	28/28
src/environments	0%	0/0	0%	0/0	0%	0/0	0%	0/0

4. We're aiming for 100% code coverage in this guide, including every if, else, switch, &&, and || statements. All of the work anyone produces should always aim for 100% test coverage.

ANGULAR HOMEPAGE COMPONENT TDD

1. HomePage Setup Pg 10-12
2. PlayerClass Method TDD Pg 13-17
3. Checkpoint 1 (Creating Difficulty Method with TDD) Pg 18-19
4. StartGame Method Testing Pg 20-24
5. Checkpoint 2 (Creating the startGame Method implementation) Pg 23-24
6. CheckStartButton Method TDD Pg 25-27

ANGULAR HOMEPAGE COMPONENT TDD

1. CD into enablementorium, and check branch.

```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git branch
* develop
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

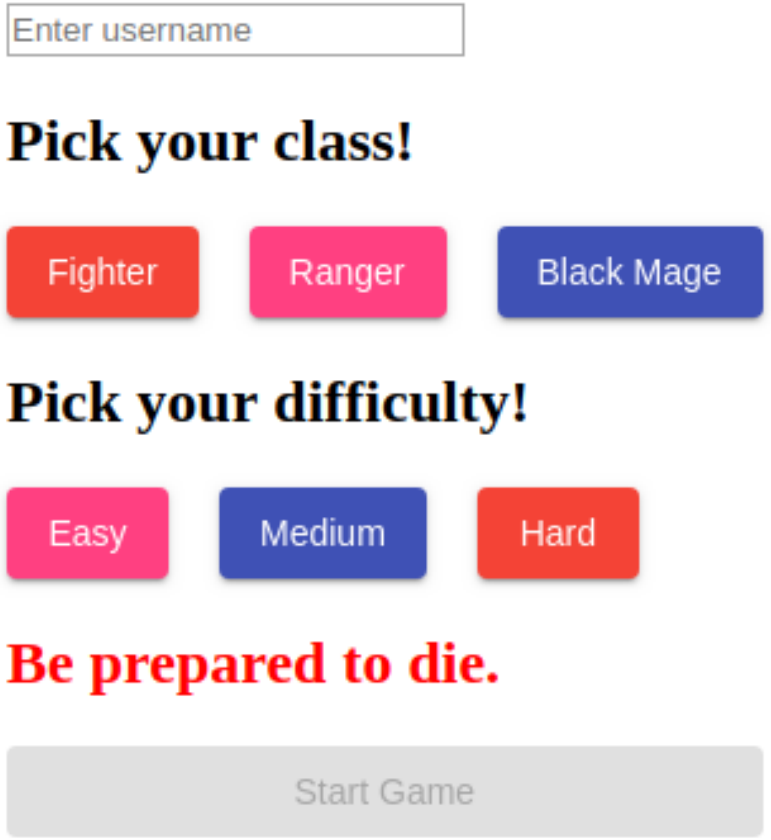
2. Now let's checkout to "feature/homePageComponentUnfinishedTDD", this branch has no TDD involving the Home Page. The final version is on "feature/homePageComponentFinishedTDD".

```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/homePageComponentUnfinishedTDD
Switched to branch 'feature/homePageComponentUnfinishedTDD'
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$
```

HOMEPAGE WIREFRAME

1. For our project, our "client" wants to create a game. They want us to create a dungeon crawler. For this game, they want to offer each player a choice between classes; "Fighter", "Ranger", and a "Black Mage". They also want to offer each player a choice of difficulty, offering "Easy", "Medium", and "Hard". Furthermore, they want the game to be a dice based game and have a scoreboard so it's competitive.

2. For the home page, they have provided us with the following wireframe.



SHALLOW SPEC SETUP

1. For shallow testing, we separate our HTML, CSS, TS testing (which uses testbed and fixture), from our pure Unit Tests. To do this, we created a home.component.shallow.spec.ts file and copied all the files from our original home.component.spec.ts file.

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { HomeComponent } from './home.component';

fdescribe('HomeComponent shallow', () => {
  let component: HomeComponent;
  let fixture: ComponentFixture<HomeComponent>;

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

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

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

2. From there we can clean our original spec file and set it up for pure Unit Testing.

```
import { HomeComponent } from './home.component';

fdescribe('HomeComponent', () => {
  let component: HomeComponent;


  beforeEach(() => {
    component = new HomeComponent();
  });

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

HOMEPAGE COMPONENT TDD

1. So first things first, in our HTML, let's go ahead and just directly bind our username input box to a username variable. Our HTML is already defined with a [(ngModel)]="username". This well automatically picks up input from the user and stores it into a variable called username.

```
<input placeholder="Enter username" [(ngModel)]="username" (change)="checkStartButton()"/>
```

2. So let's create a variable named username. If we don't set it to an empty string our tests will complain saying username is undefined.

```
export class HomeComponent implements OnInit {

  username: string = "";

  constructor() { }

  ngOnInit() {
  }
```

3. Now run our tests. The tests will fail saying "ngModel isn't a known property of 'input'". Since this is testing HTML, and since we don't test for HTML or CSS, simply go into the shallow.spec.ts file and add the following imports.

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

4. This allows the component to be built which is why shallow is important still but seperates it from testing our TS methods.

HOMEPAGE COMPONENT TDD

5. Now for our TS, lets add a global "classSelected: string" variable so we can use this later. This will be used later when checking conditions to see if the play button should be clickable or not.

```
classSelected : string = "";
username: string = "";

constructor() { }

ngOnInit() {
}
```

6. Now let's switch over to our spec file, "home.component.spec.ts", and add a Unit Test for a method called "playerClass".
If we look at the HTML,

```
<div class = "dice">
  <button mat-raised-button color="warn" (click)="playerClass('Fighter')">Fighter</button>
  <button mat-raised-button color="accent" (click)="playerClass('Ranger')">Ranger</button>
  <button mat-raised-button color="primary" (click)="playerClass('BlackMage')">Black Mage</button>
</div>
```

We're passing in a string to our method on a click event. So the final signature of our method should look like "playerClass(currClass: string){ }".

HOMEPAGE COMPONENT TDD

7. Furthermore, let's create a signature of a method to check our initial validation (checking to see if the user selected a class, selected a difficulty, and typed in a username).

```
checkStartButton() {
}
```

8. If the user has filled the entire page out, then we enable the button, as we can see in the HTML.

```
<button [disabled]="isDisabled" mat-raised-button color="primary" (click)="startGame()">Start Game</button>
```

9. We directly binded the disabled value to a boolean inside our TS, so let's create another global variable "isDisabled: boolean" and set it to true by default so the button Start Game is disabled until we till it otherwise.

```
classSelected : string = "";
isDisabled: boolean = true;
username: string = "";

constructor() { }

ngOnInit() {
}
```

HOMEPAGE COMPONENT TDD

10. Now that our HTML is correctly binded to our TS file, let's create a test for our method "playerClass(currClass:string) { }" inside our "home.component.spec.ts" file.

```
fdescribe('HomeComponent', () => {
  let component: HomeComponent;

  beforeEach(() => {
    component = new HomeComponent();
  });

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

  it('should get the players class and save it in sessionStorage', () => {
    spyOn(component, 'checkStartButton')

    component.playerClass("Ranger");

    expect(component.classSelected).toEqual("Ranger selected");

    expect(sessionStorage.getItem('playersClass')).toEqual("Ranger");

    expect(component.checkStartButton).toHaveBeenCalledTimes(1);
  });
});
```

First thing we should notice, we are no longer using TestBed and fixtures, instead for Unit Testing, we create a component as usual, by calling it's constructor. By doing it this way, we have complete control over our injects rather than letting TestBed do it. Next, we "spyOn" our checkStartButton method which behaves in the same manner as a mock from Java. Since Unit testing is only testing one method at a time, we simply use a fake call to our other method to check if it's called, and it should only be called 1 time. Futhermore, after we call our empty signature, we're checking to make sure our global variable (classSelected: string) was set, and that we also saved the class into sessionStorage.

HOMEPAGE COMPONENT TDD

11. SessionStorage allows us to save data to a particular browser. By doing so, even once we navigate to other components using routing, we can still access this data.

12. If we run out test now, it fails. Now implement our actual code to make the test pass.

```
playerClass(currClass: string) {
  this.classSelected = currClass + " selected";
  sessionStorage.setItem('playersClass', currClass);
  this.checkStartButton();
}
```

First thing we do, we change our global variable for our validation, we then create our sessionStorage variable "playersClass" and set it to "currClass" which is our input variable. We then make a call to our "checkStartButton()" method, to see if all validations pass now which will enable the Start Game button.

HOMEPAGE CHECKPOINT 1

1. Now that we have our "playerClass(currClass: string)" method, it's test method, and the following HTML code which is already implemented for you. Your task is to create a similar method using TDD to get the difficulty.

```
<div class = "dice">
  <button mat-raised-button color="accent" (click)="difficulty('Easy')">Easy</button>
  <button mat-raised-button color="primary" (click)="difficulty('Medium')">Medium</button>
  <button mat-raised-button color="warn" (click)="difficulty('Hard')">Hard</button>
</div>
```

HOMEPAGE CHECKPOINT 1 SOLUTION

1. Following the same way we implemented "playerClass(currClass: string)" method, the Unit test should look like,

```
it('should get the difficulty selected, and save it in sessionStorage', () => {
  spyOn(component, 'checkStartButton');

  component.difficulty("Easy");

  expect(component.difficultySelected).toEqual("Easy selected");

  expect(sessionStorage.getItem('difficulty')).toEqual("Easy");

  expect(component.checkStartButton).toHaveBeenCalledTimes(1);
});
```

2. The actual code implementation should look like,

```
difficulty(difficulty: string) {
  this.difficultySelected = difficulty + " selected";
  sessionStorage.setItem('difficulty', difficulty);
  this.checkStartButton();
}
```

With a new global variable "difficultySelected: string".

```
classSelected : string = "";
difficultySelected : string = "";
isDisabled: boolean = true;
username: string = "";
```

HOMEPAGE COMPONENT TDD

13. We are now going to implement a method to launch our game. Once our button is clickable, we want the button to call a method that will navigate the user into the game, in this case, the dungion-crawler component.

14. Since we handled the logic of turning off and on the clickable portion of the button with a boolean, all this method needs to do is save the username into sessionStorage and then navigate to the other component. So let's create a test for this.

15. First thing we need to do; in our actual code, inject a private Router into our constructor. This will allow us to navigate to another component.

```
export class HomeComponent implements OnInit {

  classSelected : string = "";
  difficultySelected : string = "";
  isDisabled: boolean = true;
  username: string = "";

  constructor(private _router: Router) { }

  ngOnInit() {
  }
```

HOMEPAGE COMPONENT TDD

16. With our router now injected, we need to create our own Router in our Unit Testing so that way we have complete control over our injects.

```
let component: HomeComponent;
let mockRouter: any;

beforeEach(() => {
  mockRouter = jasmine.createSpyObj(['navigate']);
  component = new HomeComponent(mockRouter);
});
```

17. We begin by declaring a mockRouter and setting it's type to any. If we set its type to "mockRouter:Router", that would be using the actual Router taking away our control. So we defined it as type any so we can pass it into the constructor without any problems and giving is complete control.

18. With this control, we set the mockRouter to a spy object with a method named "navigate". So now we have a fake method inside of our fake "mocked" Router. All we need to do now, is to inject this into our component.

19. When it comes to shallow testing, the injection into the constructor is the most confusing part, but can be very rewarding.

HOMEPAGE COMPONENT TDD

20. Now that we have our mocked Router setup and injected into our actual code, we can make a test now to check this navigation and to check if the username is actually stored.

```
it('should save the username to sessionStorage and route to the dungion component', () => {
  component.username = "Michael";

  component.startGame();

  expect(sessionStorage.getItem('username')).toEqual("Michael");

  expect(mockRouter.navigate).toHaveBeenCalledTimes(1);

  expect(mockRouter.navigate).toHaveBeenCalledWith(['dungion']);
});
```

21. So we set the username ourselves, since it's set in the HTML code, then run our method. After our method is called, it should save into the sessionStorage variable "username", and then it should call our mocked Router's navigate method, and it should call that method with this parameter.

HOMEPAGE CHECKPOINT 2

1. With our test written for our "startGame() { }" method, we have a compiler error stating that "component.startGame()" doesn't exist.

2. Your task is to create the actual code implementation for this method to make the test pass. Furthermore, once the signature is created, the compiler error will disappear which means the test went from red to green. Run your tests again with a blank signature to watch them fail.

3. Once your tests fail without a compiler error, implement the actual code to make them pass.

HOMEPAGE CHECKPOINT 2 SOLUTION

1. The solution for this can be accomplished in two lines of code. We first save the username into sessionStorage, and then route to our other component.

```
startGame() {
  sessionStorage.setItem('username', this.username);
  this._router.navigate(['dungion']);
}
```

2. If we run our test now, you can see that it passes.

HOMEPAGE COMPONENT TDD

22. We now have our "playerClass(currClass: string)", "difficulty(difficulty: string)", and "startGame()" methods implemented. From here, the only thing we're missing is to create a method that checks all the critera and determines if the "startGame()" button should be clickable to launch the game.

23. Previously we created a "checkStartButton() { }" empty signature so we could "spyOn" the method call. Now let's create a test to check this method.

```
it('should set isDisabled to false if everything has been filled in', () => {
  expect(component.isDisabled).toBe(true);

  component.classSelected = "Ranger selected";

  component.difficultySelected = "Easy selected";

  component.username = "Michael";

  component.checkStartButton();

  expect(component.isDisabled).toBe(false);
});
```

24. We first test to make sure the initial value is true, making the button un-clickable. We then set the class, difficulty, and username global variables; exactly as a user would fill out the information. We then run our component's method and test to make sure the value is now false, making the button clickable.

HOMEPAGE COMPONENT TDD

25. If we run our test, it fails. Now let's create the actual code implementation to make our test pass.

```
checkStartButton() {
  this.isDisabled = false;
}
```

26. However, as you can see, the actual code implementation doesn't check the global variables in any way and yet the test passes. In this case, the test is passing but for the wrong reasons. We should now break this test up into multiple tests and check each piece of our validation to make sure it passes for the correct reasons.

HOMEPAGE COMPONENT TDD

27. The way we're going to fix this, is by creating three more tests. The first test would still pass for the wrong reasons, but the three other tests would check all 3 global variables to make sure the button is un-clickable (isDisabled = true) if only 1 global variable is set.

```
it('should keep isDisabled to true if only username is input', () => {
  expect(component.isDisabled).toBe(true);

  component.username = "Michael";

  component.checkStartButton();

  expect(component.isDisabled).toBe(true);
});

it('should keep isDisabled to true if only username is input', () => {
  expect(component.isDisabled).toBe(true);

  component.classSelected = "Ranger selected";

  component.checkStartButton();

  expect(component.isDisabled).toBe(true);
});

it('should keep isDisabled to true if only username is input', () => {
  expect(component.isDisabled).toBe(true);

  component.difficultySelected = "Easy selected";

  component.checkStartButton();

  expect(component.isDisabled).toBe(true);
});
```

28. In these tests, the previous code, Would make these fail, but with a Modification to our implementation These tests will now pass.

```
checkStartButton() {
  this.isDisabled = false;
}
```

```
checkStartButton() {
  if(this.classSelected != "" && this.difficultySelected != "" && this.username != ""){
    this.isDisabled = false;
  }
}
```

ANGULAR ATTACKBUTTONS COMPONENT TDD

1. AttackButtons Setup Pg 29-32
2. NgOnInit method TDD Pg 33-35
3. Delay method TDD Pg 35-36
4. ButtonClicked TDD (With a "@Output()" Variable and Delay) Pg 37-39
5. Checkpoint 1 (Implementing ButtonClicked method) Pg 40-41

ANGULAR ATTACK BUTTONS SETUP

1. Let's start with a CD into our enablementorium and check our branch.

```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git branch
* develop
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

2. Next, let's switch to our unfinished TDD branch which is equivalent to doing a "ng g c attackButtons".

```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/attackButtonsUnfinishedTDD
Branch 'feature/attackButtonsUnfinishedTDD' set up to track remote branch 'feature/attackButtonsUnfinishedTDD' from 'origin'.
Switched to a new branch 'feature/attackButtonsUnfinishedTDD'
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$
```

3. Since we don't TDD html, we can ignore our mat-table currently in "attack-buttons.component.html".

ANGULAR ATTACK BUTTONS SHALLOW SETUP

1. If we run our test from the beginning, you'll notice we have a test failing.

```
HeadlessChrome 0.0.0 (Linux 0.0.0) AttackButtonsComponent should create FAILED
        Can't bind to 'dataSource' since it isn't a known property of 'table'. ("<table mat-table [ERROR ->][dataSource]="dataSource" class="mat-elevation-z8">
          <ng-container matColumnDef="attack">
            <th ma"): ng:///DynamicTestModule/AttackButtonsComponent.html@0:17
        Can't bind to 'matHeaderRowDef' since it isn't a known property of 'tr'. ("
          </ng-container>
          <tr mat-header-row [ERROR ->]*matHeaderRowDef="displayedColumns"></tr>
          <tr mat-row *matRowDef="let row; columns: displayedColumn"): ng:///DynamicTestModule/AttackButtonsComponent.html@18:21
```

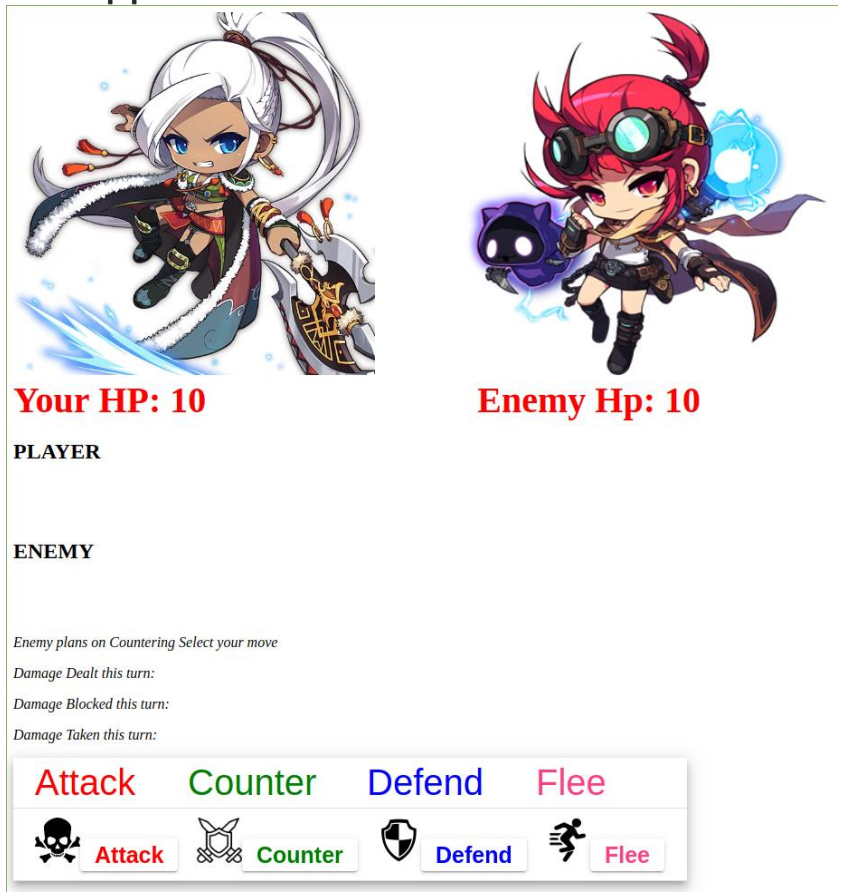
2. Our Unit Tests don't have a problem, this test is failing from our "attack-buttons.component.shallow.spec.ts" file because it still accounts for our HTML. Inside of this file, let's just simply add an imports with a "MatTableModule" to make this test pass.

```
fdescribe('AttackButtonsComponent', () => {
  let component: AttackButtonsComponent;
  let fixture: ComponentFixture<AttackButtonsComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ AttackButtonsComponent ],
      imports: [MatTableModule]
    })
    .compileComponents();
  }));

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

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

ANGULAR ATTACK BUTTONS WIREFRAME

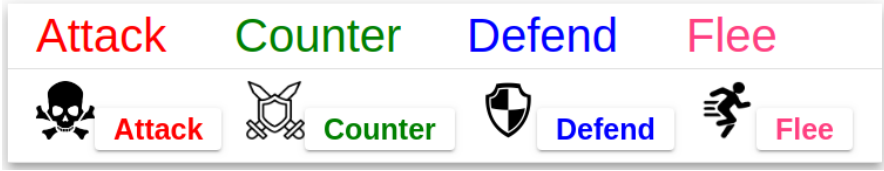
1. Now that we have our home page component setup and fully Unit Tested, let's create our next component that is going to be navigated to once the user has entered all the information.

2. For this wireframe, we need a page that allows us to walk through a Dungion. We need to have our character as well as the opponent, a set of moves for us to interact with, some notifications to show damage and other information. Furthermore, we also need to show our dice rolls and the opponents dice rolls.



ANGULAR ATTACK BUTTONS WIREFRAME

3. For now we're going to focus on just the bottom piece for interacting with the user and what action they want to perform.



ANGULAR ATTACK BUTTONS COMPONENT TDD

1. So, if we take a look at our HTML for this component, we have a Mat-Table which is using a displayedColumns, and a dataSource. Each column also contains a single button that we'll have to hook up later.

2. For now, let's go ahead and create our variables for the Mat-Table inside our "attack-buttons.component.ts". Since our Mat-Table is not being populated with any actual information, since we're using it just to display buttons in a beautiful way; it's left blank and is initialized to just an empty object.

```
displayedColumns : string[] = ['attack', 'counterattack', 'defend', 'flee'];
dataSource = new MatTableDataSource();
```

3. Now our file should look like,

```
@Component({
  selector: 'app-attack-buttons',
  templateUrl: './attack-buttons.component.html',
  styleUrls: ['./attack-buttons.component.css']
})
export class AttackButtonsComponent implements OnInit {

  displayedColumns : string[] = ['attack', 'counterattack', 'defend', 'flee'];
  dataSource = new MatTableDataSource();

  constructor() { }

  ngOnInit() {
  }
}
```

ANGULAR ATTACK BUTTONS COMPONENT TDD

4. So now we can begin writing Unit Tests before we write actual code. With our testing file "attack-buttons.component.spec.ts" starting with,

```
fdescribe('AttackButtonsComponent', () => {
  let component: AttackButtonsComponent;

  beforeEach(() => {
    component = new AttackButtonsComponent();
  });

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

5. So with every dataSource it needs to be set for anything to be displayed. So we can begin by setting our dataSource.data to an empty object. Let's write a test for that.

```
it('should set the dataSource to empty on startup', () => {
  let mockMatTableDataSource: any = {
    data: 'hello'
  }
  component.dataSource = mockMatTableDataSource;
  component.ngOnInit();
  expect(component.dataSource.data).toEqual([{}]);
});
```

6. We initialize the data to 'hello', and then make sure when we run this page "ngOnInit();", it gets set to empty.

ANGULAR ATTACK BUTTONS COMPONENT TDD

7. Now with our test in place, we can write our actual code.

```
ngOnInit() {
  let dataaa = [{}];
  this.dataSource.data = dataaa;
}
```

8. Next, we should implement a delay method that simply just makes our buttons un-clickable for some period of time to prevent the user spamming buttons. Starting with our test case,

```
it('should delay for 2 seconds', fakeAsync(() => {
  let delayedValue: boolean = false;
  component.delay(2000).then( () => {
    delayedValue = true;
  });
  expect(delayedValue).toBeFalsy();
  tick(50);
  expect(delayedValue).toBeFalsy();
  tick(1950);
  expect(delayedValue).toBeTruthy();
}));
```

9. The way we test for this is by using fakeAsync. This allows us to use methods like "tick(milliseconds)". Tick passes time by waiting for a response to happen, so we set some boolean to false, and wait for a total of 2 seconds to pass. Once the time has passed, we set the boolean to true and then check to see if the value has been changed from false to true.

ANGULAR ATTACK BUTTONS COMPONENT TDD

10. Now with our test written, let's go ahead and implement our delay method.

```
delay(ms: number) {
  return new Promise( resolve => setTimeout( () => {
    resolve(ms);
  }, ms));
}
```

11. All this method is doing, is waiting for the number of milliseconds specified.

12. So next, what we want to do is create a method that takes in a string and sends that string up to it's parent component since our "attack-buttons.component" is being injected into our "dungion-crawler.component.html". So in this case, our injected component is the child, and the component it's injected into is our parent. By doing it this way, we separate concerns and have only particular components take care of some piece of logic while our parent "dungion-crawler.component", uses these results to do comparisons and spit out information to the user. Furthermore, our method needs to call our delay method so the buttons are un-clickable for some period of time.

ANGULAR ATTACK BUTTONS COMPONENT TDD

13. Before we implement our method, if we take a quick look at our HTML we have,

```
<button class="button-design flee-design" id="fleeButton"
 mat-raised-button [disabled]="isDisabled" (click)="buttonClicked('Flee')">Flee</button></td>
```

Along with 3 other buttons, all passing in strings to a method called "buttonClicked(string)". The buttons also have their disabled binded to a boolean, this boolean we need to change depending on our delay method.

14. So, let's go ahead and create a global variable to match our HTML's disabled variable.

```
isDisabled = false;
```

15. Now our component.ts should look like this

```
@Component({
  selector: 'app-attack-buttons',
  templateUrl: './attack-buttons.component.html',
  styleUrls: ['./attack-buttons.component.css']
})
export class AttackButtonsComponent implements OnInit {

  displayedColumns : string[] = ['attack', 'counterattack', 'defend', 'flee'];
  dataSource = new MatTableDataSource();
  isDisabled = false;

  constructor() { }

  ngOnInit() {
    let dataaa = [{}];
    this.dataSource.data = dataaa;
  }

  delay(ms: number) {
    return new Promise( resolve => setTimeout( () => {
      resolve(ms);
    }, ms));
  }
}
```

ANGULAR ATTACK BUTTONS COMPONENT TDD

16. So with this all in mind, let's create a test for this method.

```
it('should send the button clicked back up to the parent component', fakeAsync(() => {
  component.isDisabled = true;
  let mockEventEmitter: any = {
    emit: jasmine.createSpy('emit')
  };
  let mockThen: any = {
    then: fn => fn()
  }
  component.currentPlayerCommand = mockEventEmitter;
  spyOn(component, 'delay').and.returnValue(mockThen);
  component.buttonClicked("Attack");
  expect(component.currentPlayerCommand.emit).toHaveBeenCalledTimes(1);
  expect(component.currentPlayerCommand.emit).toHaveBeenCalledWith("Attack");
  expect(component.isDisabled).toBeFalsy();

}));
```

17. The way we go about writing this test, is by setting our isDisabled global to true, so that when we click a button we disable the buttons. We're also going to need to create an "@Output()" variable to pass the clicked button command to our parent component, and this variable type "EventEmitter" has an emit method that we need to mock. Next what we need to do is to mock out a "then" method because we're going to have to call our implemented delay method which returns a promise and promises have a then method to handle a block of code when the promise finally returns a value. Next we have to inject our fake EventEmitter into our real one. Furthermore, since we're calling our delay method, we need to "spyOn(component,'delay')" and have it return our mocked "then" method which simply just returns a function "fn" which is the same as instantly returning from our delay method. However, since we're still using our delay method, we need to make this test a "fakeAsync()". After all the setup, we just make sure our mocks are called, and that our "isDisabled" is changed to false, making sure our buttons are clickable.

ANGULAR ATTACK BUTTONS COMPONENT TDD

18. Now that our test is written, let's create a global variable to send back to our parent component.

```
@Output() currentPlayerCommand = new EventEmitter<string>();
```

19. Now our component should look like.

```
@Component({
  selector: 'app-attack-buttons',
  templateUrl: './attack-buttons.component.html',
  styleUrls: ['./attack-buttons.component.css']
})
export class AttackButtonsComponent implements OnInit {

  displayedColumns : string[] = ['attack', 'counterattack', 'defend', 'flee'];
  dataSource = new MatTableDataSource();
  isDisabled = false;
  @Output() currentPlayerCommand = new EventEmitter<string>();

  constructor() { }

  ngOnInit() {
    let dataaa = [{}];
    this.dataSource.data = dataaa;
  }

  delay(ms: number) {
    return new Promise( resolve => setTimeout( () => {
      resolve(ms);
    }, ms));
  }
}
```

ANGULAR ATTACK BUTTONS CHECKPOINT 1

1. Now with our global variables, and test method written, your goal is to write the implementation code for our method "buttonClicked(string){ }". Make sure this method is inside our "attack-buttons.component.ts".

ANGULAR ATTACK BUTTONS CHECKPOINT 1 SOLUTION

1. For our solution, we begin by emitting the command back up to the parent. After we send that, we then change our buttons to un-clickable by setting our global binded variable "isDisabled" to true. After that we call our delay method, and only once 2 seconds have passed, we make our buttons clickable again by changing the variable to false.

```
buttonClicked(currentCommand: string) {
  this.currentPlayerCommand.emit(currentCommand);
  this.isDisabled = true;
  this.delay(2000).then( () => {
    this.isDisabled = false;
  });
}
```

2. Now our final solution should look like,

```
@Component({
  selector: 'app-attack-buttons',
  templateUrl: './attack-buttons.component.html',
  styleUrls: ['./attack-buttons.component.css']
})
export class AttackButtonsComponent implements OnInit {

  displayedColumns : string[] = ['attack', 'counterattack', 'defend', 'flee'];
  dataSource = new MatTableDataSource();
  isDisabled = false;
  @Output() currentPlayerCommand = new EventEmitter<string>();

  constructor() { }

  ngOnInit() {
    let dataaa = [{}];
    this.dataSource.data = dataaa;
  }

  buttonClicked(currentCommand: string) {
    this.currentPlayerCommand.emit(currentCommand);
    this.isDisabled = true;
    this.delay(2000).then( () => {
      this.isDisabled = false;
    });
  }

  delay(ms: number) {
    return new Promise( resolve => setTimeout( () => {
      resolve(ms);
    }, ms));
  }
}
```

ANGULAR ENEMYUI COMPONENT TDD

1. EnemyUI setup Pg 43-44
2. NgOnInit method TDD Pg 45-51
3. Checkpoint 1 (Implementing switch cases) Pg 52-53
4. getNextMob method TDD Pg 54-67
5. getEnemyAttackType method TDD Pg 68-84
6. Checkpoint 2 (Mocking and Injecting) Pg 74-75
7. triggerEnemyDiceRoll method TDD Pg 85-93
8. Checkpoint 3 (Mocking and Injecting) Pg 86-88

ENEMYUI COMPONENT TDD

1. Our next step is to create a component that will handle the enemy's decision making based on a random number generator. First thing, let's go ahead and cd into our enablementorium folder and check our branch.

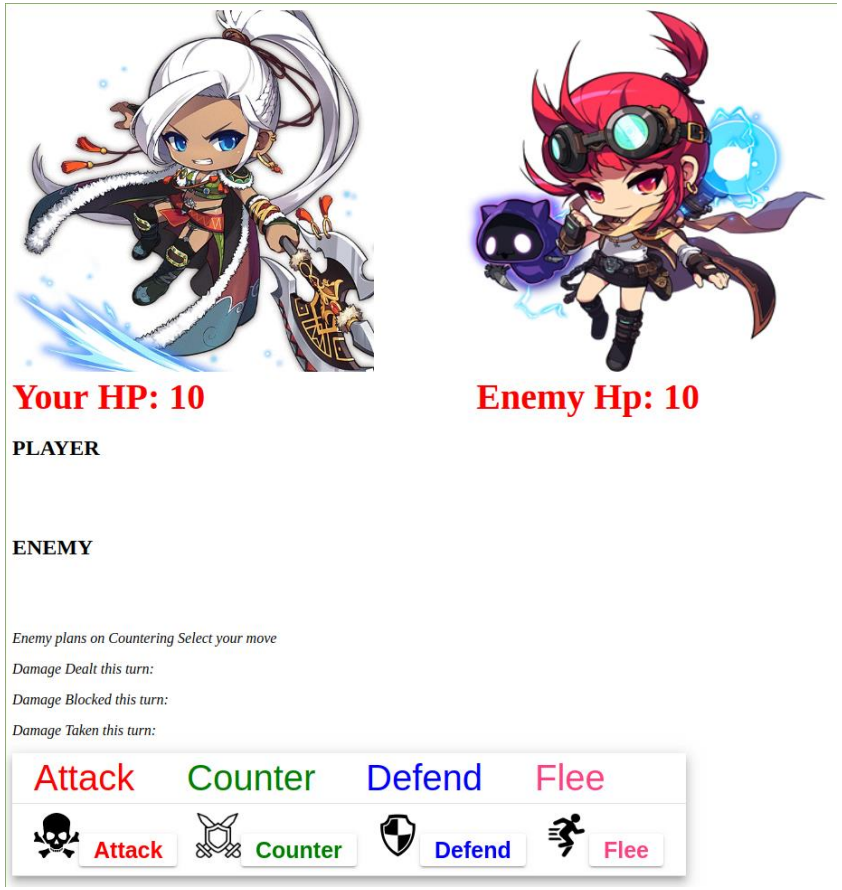
```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git branch
* develop
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

2. Now let's checkout our un-finished TDD branch

```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/enemyUiComponentUnfinishedTDD
Branch 'feature/enemyUiComponentUnfinishedTDD' set up to track remote branch 'feature/enemyUiComponentUnfinishedTDD' from 'origin'.
Switched to a new branch 'feature/enemyUiComponentUnfinishedTDD'
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$
```

ENEMYUI COMPONENT TDD

3. So if we take a second and think, what's the first thing we need in our enemy's UI? If we take a look at the wireframe provided,



We need the enemies Ui to keep track of the enemies and which enemy we're currently fighting. With this, we need to keep track of which image we need to display, we also need to calculate what move the enemy is going to use so we can inform the player, notice how it says "Enemy plans on Countering Select your move". Lastly, we need this component to roll dice for the enemy depending on the move they selected.

ENEMYUI COMPONENT TDD

4. The first thing we need to do is fetch a mobOrder (list of enemies to fight), from our "mobOrderService". So we need to import this service into our constuctor, but first let's add it to our tests. Whenever we're injecting into our constructor, we need to add it to both our IT Testing "shallow.spec.ts" and to to our Unit Testing "spec.ts" as a mocked value/class. Furthermore, we do this because we're only testing this component, so we mock everything else external to our component.

5. So our first step is to add it to our "shallow.spec.ts" file.

```
fdescribe('EnemyUIComponent shallow', () => {
  let component: EnemyUIComponent;
  let fixture: ComponentFixture<EnemyUIComponent>;
  let mobOrderService: MobOrderService;

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

  beforeEach(() => {
    fixture = TestBed.createComponent(EnemyUIComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
    mobOrderService = TestBed.get(MobOrderService);
  });

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

6. As a first step, we declare it as the actual service, and then tell TestBed to initialize the real object as well.

ENEMYUI COMPONENT TDD

7. So our next step, is to replace the real one with a mocked version;

```
const mockMobOrderService = {
  createMobOrder: jasmine.createSpy('createMobOrder').and.returnValue(of(mockMobOrderEasy))
}
describe('EnemyUIComponent shallow', () => {
  let component: EnemyUIComponent;
  let fixture: ComponentFixture<EnemyUIComponent>;
  let mobOrderService: MobOrderService;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ EnemyUIComponent ],
      providers: [{provide: MobOrderService, useValue: mockMobOrderService}]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(EnemyUIComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
    mobOrderService = TestBed.get(MobOrderService);
  });

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

The way we set this up is by creating a object"mockMobOrderService",and giving it a variable "createMobOrder",which has the same method name as the real service.This way when we call this method it will call our spy rather than the real one and return our value "mockMobOrderEasy".The "of", "of(mockMobOrderEasy)" is needed if the method we're expecting returns an observable. Lastly, we need to put it inside of our providers, by doing this, we provide the real one but then "useValue:mockMobOrderService". Furthermore, when this happens, TestBed tries to get the real service, but instead it uses our mockedService.

ENEMYUI COMPONENT TDD

8. With our "shallow.spec.ts" file setup, now we can setup our "spec.ts" Unit Testing file; very similar to our shallow file.

```
const mockMobOrderService = {
  createMobOrder: jasmine.createSpy('createMobOrder').and.returnValues(of(mockMobOrderEasy), of(mockMobOrderMedium), of(mockMobOrderHard))
}
fdescribe('EnemyUIComponent', () => {
  let component: EnemyUIComponent;
  let mobOrderService: any;

  beforeEach(() => {
    mobOrderService = mockMobOrderService;
    component = new EnemyUIComponent(mobOrderService);
  });

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

9. In this case, our mock is the same as the one in our "shallow.spec.ts", but the way we inject it is different. Rather than using TestBed to get it from our providers, we can directly declare it as an "any" object type and then initialize it to our mocked service. Therefore we can now inject it directly into our constructor. In our constructor it's going to be of type "MobOrderService", and this is why we need to declare it as an "any" type before hand. In this case "any" is taking the place of the real "MobOrderService".

ENEMYUI COMPONENT TDD

10. Now let's go ahead and do our implementation of injecting it.

```
@Component({
  selector: 'app-enemy-ui',
  templateUrl: './enemy-ui.component.html',
  styleUrls: ['./enemy-ui.component.css']
})
export class EnemyUIComponent implements OnInit {

  constructor(private _mobOrderService: MobOrderService) { }

  ngOnInit() {

  }

}
```

11. Now that our service is injected we can get a list of enemies. This list is generated on the java backend taking in a difficulty; reminder, we saved the difficulty in sessionStorage("difficulty"). We're also going to need some way to keep track of which enemy we're currently on.

ENEMYUI COMPONENT TDD

12. So let's add a global variable, `mobCounter: number;`

With that being declared, our "ngOnInit" method will need to set that number, and get a list of enemies. So we'll also need some way to store this list of mobs, hence create another global variable.

```
mobOrder: MobOrder = new MobOrder;
```

13. The last thing we'll need to do is to change the image on our enemy depending on the first enemy generated. So let's take a look at our HTML,

```
<img id="enemyClass" [src]="currentImage"/>
```

Our "src" is bounded to a global variable "currentImage", so we need to create one more global variable. `currentImage: string = "";`

We need to initialize this string to empty, otherwise it'll be undefined for our tests.

ENEMYUI COMPONENT TDD

14. So with this all in mind, let's now go ahead and write our first test for our "ngOnInit" method.

```
it('should send a get a mobOrder from the service with mob1 = Ranger', () => {
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Easy');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(1);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Easy');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
});
```

15. We hardcode mobCounter to be 0, so when we call "ngOnInit" we make sure it gets set to 1 by default. We also set our sessionStorage difficulty to easy. Furthermore, if you remember from the setup,

```
const mockMobOrderService = {
  createMobOrder: jasmine.createSpy('createMobOrder').and.returnValues(of(mockMobOrderEasy),
                                                                        of(mockMobOrderMedium),
                                                                        of(mockMobOrderHard))
}
```

MockMobOrderEasy has a "Ranger" as it's first enemy so we return that value the first time our service method is called. The way "returnValues" works, the first time our spy is called it will return mockMobOrderEasy, second time our spy is called, it returns mockMobOrderMedium, and if it's called a third time we'll return mockMobOrderHard.

ENEMYUI COMPONENT TDD

16. Now with our first test implemented, let's go ahead and create just enough code for this to pass.

```
ngOnInit() {
  this.mobCounter = 1;
  this._mobOrderService.createMobOrder(sessionStorage.getItem("difficulty")).subscribe( response => {
    this.mobOrder = response;
    switch(this.mobOrder.mob1){
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
  });
}
```

Now that we have our first test passing, let's go ahead and add 2 more tests to check if the first enemy is a "Black_Mage" or a "Fighter".

```
it('should send a get a mobOrder from the service with mob1 = Fighter', () => {
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Medium');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(2);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Medium');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyFighter.png');
});

it('should send a get a mobOrder from the service with mob1 = Black Mage', () => {
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Hard');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(3);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Hard');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyBlackMage.png');
});
```

ENEMYUI CHECKPOINT 1

1. With our 2 new tests and given our current implementation,

```
@Component({
  selector: 'app-enemy-ui',
  templateUrl: './enemy-ui.component.html',
  styleUrls: ['./enemy-ui.component.css']
})
export class EnemyUIComponent implements OnInit {

  mobOrder: MobOrder = new MobOrder;
  mobCounter: number;
  currentImage: string = "";

  constructor(private _mobOrderService: MobOrderService) { }

  ngOnInit() {
    this.mobCounter = 1;
    this._mobOrderService.createMobOrder(sessionStorage.getItem("difficulty")).subscribe( response => {
      this.mobOrder = response;
      switch(this.mobOrder.mob1){
        case ClassListEnum.Ranger:
          this.currentImage = "assets/images/EnemyRanger.png";
      }
    });
  }
}
```

2. Get the new tests to pass.

ENEMYUI CHECKPOINT 1 SOLUTION

1. The answer to this checkpoint is just adding 2 more cases to our switch statement,

```
ngOnInit() {
  this.mobCounter = 1;
  this._mobOrderService.createMobOrder(sessionStorage.getItem("difficulty")).subscribe( response => {
    this.mobOrder = response;
    switch(this.mobOrder.mob1){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
  });
}
```

Now our tests pass

ENEMYUI COMPONENT TDD

17. Our next step is to implement a new method that will increase our global variable "mobCounter", and grab the next enemy in our list when the previous enemy is defeated. Since we're handling that logic in our parent component "dungion-crawler.component.ts", all we need to do is display a different image on our enemyUI HTML page. The way we can create this is by creating an "@Input()" variable and setting it's type to "Observable<void>". By doing this, whenever we send anything to this variable, the variables ".subscribe()" method will trigger so we can grab the next enemy. First let's create another "describe" inside our "spec.ts" file.

```
describe('should call the get the next Mob in the order and grab the correct image', () => {

});
```

ENEMYUI COMPONENT TDD

18. The logic in our next method is going to be very similar to our "ngOnInit" method. Everytime it gets called, we increment mobCounter by 1, and change the image to the next image depending on the next enemy. Knowing that we need 3 tests ahead of time to check each piece of our switch statement, we end up with.

```
it('should be Fighter on mobCounter = 2', () => {
  component.mobCounter = 1;
  component.mobOrder = mockMobOrderEasy;
  component.currentImage = '';
  component.getNextMob();

  expect(component.nextMob.subscribe).toHaveBeenCalled();
  expect(component.mobCounter).toEqual(2);
  expect(component.currentImage).toEqual('assets/images/EnemyFighter.png');
});

it('should be BlackMage on mobCounter = 2', () => {
  component.mobCounter = 1;
  component.mobOrder = mockMobOrderMedium;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(2);
  expect(component.currentImage).toEqual('assets/images/EnemyBlackMage.png');
});

it('should be Ranger on mobCounter = 2', () => {
  component.mobCounter = 1;
  component.mobOrder = mockMobOrderHard;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(2);
  expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
});
```

Notice, we have "expect(component.nextMob.subscribe).toHaveBeenCalled()" in only 1 of these tests. This is fine because if someone changes that piece, a test will still fail, rather than having that in all 3 tests.

ENEMYUI COMPONENT TDD

19. Looking at our new tests, we need a global variable "nextMob" that is of type "Observable". However, to make sure it's subscribe method was called, we need to inject a fake one into the real one, otherwise we cannot spy on it. So at the top of our testing file let's create one.

```
const mockMobOrderService = {
  createMobOrder: jasmine.createSpy('createMobOrder').and.returnValues(of(mockMobOrderEasy),
                                                                        of(mockMobOrderMedium),
                                                                        of(mockMobOrderHard))
}
const mockNextMob: any = {
  subscribe: jasmine.createSpy('subscribe').and.callFake(fn=>fn())
}
describe('EnemyUIComponent', () => {
  let component: EnemyUIComponent;
  let mobOrderService: any;

  beforeEach(() => {
    mobOrderService = mockMobOrderService;
    component = new EnemyUIComponent(mobOrderService);
    component.nextMob = mockNextMob;
  });

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

20. We create a "mockNextMob", and make our own "subscribe" method using a spy. This spy needs to have a ".and.callFake(fn=>fn())" because when we call this spy it simply just checks if the subscribe was called or not, but if we want to actually run the body of our subscribe, we need to make a fake call to a function we define "fn=>fn()" to trigger our Observable. The function we defined has no logic, but it passes a variable through our stream which triggers our subscribe body to run. This will make more sense when we take a look at the implementation. Once our spy is made, in our "beforeEach", we need to inject it into our components.nextMob variable after we initialize our "component = new EnemyUIComponent(mobOrderService);"

ENEMYUI COMPONENT TDD

21. In our actual implementation, we need a global "nextMob" variable. In this case when the first enemy is defeated in our parent component, we need to send a signal to our child component "enemy-ui.component" to switch to the next enemy. To achieve this, first we need to make our global variable a "@Input()" which means it's taking some signal or input from it's parent component. Next, we can make it an Observable<void>. By making it an Observable with a return type of void, we can just subscribe to this variable, and whenever we send a signal (switching enemies), the body of this subscribe will run.

22. Now let's go ahead and add our global variable.

```
@Input() nextMob: Observable<void>;
```

ENEMYUI COMPONENT TDD

23. Since we're going to add another method to our "ngOnInit()" we need to refactor those 3 tests to include checking for this method call,

```
it('should send a get a mobOrder from the service with mob1 = Ranger', () => {
  spyOn(component, 'getNextMob');
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Easy');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(1);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Easy');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
  expect(component.getNextMob).toHaveBeenCalledTimes(1);
  expect(component.getNextMob).toHaveBeenCalledWith();
});

it('should send a get a mobOrder from the service with mob1 = Fighter', () => {
  spyOn(component, 'getNextMob');
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Medium');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(2);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Medium');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyFighter.png');
  expect(component.getNextMob).toHaveBeenCalledTimes(1);
  expect(component.getNextMob).toHaveBeenCalledWith();
});

it('should send a get a mobOrder from the service with mob1 = Black Mage', () => {
  spyOn(component, 'getNextMob');
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Hard');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(3);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Hard');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyBlackMage.png');
  expect(component.getNextMob).toHaveBeenCalledTimes(1);
  expect(component.getNextMob).toHaveBeenCalledWith();
});
```

So we "spyOn(component,'getNextMob');",by doing this step, we don't actually call our new method, since we're only testing "ngOnInit()" in these tests. Then we make sure it was only called 1 time and called with no parameters.

ENEMYUI COMPONENT TDD

24. Now let's add a "getNextMob" method, our current code should look like this,

```
@Component({
  selector: 'app-enemy-ui',
  templateUrl: './enemy-ui.component.html',
  styleUrls: ['./enemy-ui.component.css']
})
export class EnemyUIComponent implements OnInit {

  @Input() nextMob: Observable<void>;
  mobOrder: MobOrder = new MobOrder;
  mobCounter: number;
  currentImage: string = "";

  constructor(private _mobOrderService: MobOrderService) { }

  ngOnInit() {
    this.mobCounter = 1;
    this._mobOrderService.createMobOrder(sessionStorage.getItem("difficulty")).subscribe( response => {
      this.mobOrder = response;
      switch(this.mobOrder.mob1){
        case ClassListEnum.Black_Mage:
          this.currentImage = "assets/images/EnemyBlackMage.png";
          break;
        case ClassListEnum.Fighter:
          this.currentImage = "assets/images/EnemyFighter.png";
          break;
        case ClassListEnum.Ranger:
          this.currentImage = "assets/images/EnemyRanger.png";
      }
    });
    this.getNextMob();
  }

  getNextMob() {}
}
```

Since observables will only run when we subscribe to them, we need to setup our subscribe as soon as this component is loaded, hence "ngOnInit()".

ENEMYUI COMPONENT TDD

25. Before going back to our "spec.ts" file, let's add "nextMob" to our "shallow.spec.ts" file. If we skip this step, our shallow file will fail because it won't know what "nextMob.subscribe" is. So first thing we need to do is create a mocked global variable "nextMob" with a spy for our "subscribe" method. The reason we need it to return an empty function, is to make sure our subscribe runs.

```
const mockNextMob: any = {
  subscribe: jasmine.createSpy('subscribe').and.returnValue(fn => fn())
}
```

From there we can inject it into our components real "nextMob" global variable.

```
fixture = TestBed.createComponent(EnemyUIComponent);
component = fixture.componentInstance;
component.nextMob = mockNextMob;
```

Our shallow file should now look like,

```
const mockMobOrderService = {
  createMobOrder: jasmine.createSpy('createMobOrder').and.returnValue(of(mockMobOrderEasy))
}
const mockNextMob: any = {
  subscribe: jasmine.createSpy('subscribe').and.returnValue(fn => fn())
}
describe('EnemyUIComponent shallow', () => {
  let component: EnemyUIComponent;
  let fixture: ComponentFixture<EnemyUIComponent>;
  let mobOrderService: MobOrderService;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ EnemyUIComponent ],
      providers: [{provide: MobOrderService, useValue: mockMobOrderService}]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(EnemyUIComponent);
    component = fixture.componentInstance;
    component.nextMob = mockNextMob;
    fixture.detectChanges();
    mobOrderService = TestBed.get(MobOrderService);
  });

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

ENEMYUI COMPONENT TDD

26. Now that it's mocked in our shallow file, we need to mock it in our "spec.ts" file as well. We mock it almost the same way,

```
const mockNextMob: any = {
  subscribe: jasmine.createSpy('subscribe').and.callFake(fn=>fn())
}
```

In this case, we change ".and.returnValue(fn => fn())" to ".and.callFake(fn=>fn())". We do this because our "mockNextMob" has to be of type "any" for us to inject it into our component. So when we call our subscribe method, we need it to be recognized as a method so we call a "fake method" which just returns nothing but will run our subscribes body of code.

27. Next all we need to do is inject it into the real one after the component has been initialized inside our "beforeEach";

```
component = new EnemyUIComponent(mobOrderService);
component.nextMob = mockNextMob;
```

ENEMYUI COMPONENT TDD

28. Now with our signature setup, we need to subscribe to our observable and keep track of which enemy we're on.

```
getNextMob() {
  this.nextMob.subscribe( () => {
    this.mobCounter += 1;
  });
}
```

29. With this code, we can now add anything else to our body and it will run everytime we send a trigger to our nextMob variable. Furthermore, we now need to switch the image depending if it's enemy 2, 3 or 4 since our "ngOnInit" covers enemy 1. So let's follow the same logic for enemy 2, starting with our tests.

```
describe('should call the get the next Mob in the order and grab the correct image', () => {
  it('should be Fighter on mobCounter = 2', () => {
    component.mobCounter = 1;
    component.mobOrder = mockMobOrderEasy;
    component.currentImage = '';
    component.getNextMob();

    expect(component.nextMob.subscribe).toHaveBeenCalled();
    expect(component.mobCounter).toEqual(2);
    expect(component.currentImage).toEqual('assets/images/EnemyFighter.png');
  });

  it('should be BlackMage on mobCounter = 2', () => {
    component.mobCounter = 1;
    component.mobOrder = mockMobOrderMedium;
    component.currentImage = '';
    component.getNextMob();

    expect(component.mobCounter).toEqual(2);
    expect(component.currentImage).toEqual('assets/images/EnemyBlackMage.png');
  });

  it('should be Ranger on mobCounter = 2', () => {
    component.mobCounter = 1;
    component.mobOrder = mockMobOrderHard;
    component.currentImage = '';
    component.getNextMob();

    expect(component.mobCounter).toEqual(2);
    expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
  });
}
```

ENEMYUI COMPONENT TDD

30. Now let's implement the code to our subscribe body we need to pass these,

```
switch(this.mobOrder.mob2){
  case ClassListEnum.Black_Mage:
    this.currentImage = "assets/images/EnemyBlackMage.png";
    break;
  case ClassListEnum.Fighter:
    this.currentImage = "assets/images/EnemyFighter.png";
    break;
  case ClassListEnum.Ranger:
    this.currentImage = "assets/images/EnemyRanger.png";
}
```

31. However, this code will only apply for enemy 2, so we'll need to add some way of switching which enemy we want. So for example, if we kill the second enemy, we want to get the image associated with enemy 3, so we can put another switch and encapsulate the other switch inside of it.

```
switch(this.mobCounter){
  case 2:
    switch(this.mobOrder.mob2){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
    break;
```

ENEMYUI COMPONENT TDD

32. We can now create more tests inside of our new describe block to account for enemy 3,

```
it('should be Fighter on mobCounter = 3', () => {
  component.mobCounter = 2;
  component.mobOrder = mockMobOrderHard;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(3);
  expect(component.currentImage).toEqual('assets/images/EnemyFighter.png');
});

it('should be BlackMage on mobCounter = 3', () => {
  component.mobCounter = 2;
  component.mobOrder = mockMobOrderMob3;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(3);
  expect(component.currentImage).toEqual('assets/images/EnemyBlackMage.png');
});

it('should be Ranger on mobCounter = 3', () => {
  component.mobCounter = 2;
  component.mobOrder = mockMobOrderMedium;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(3);
  expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
});
```

It's nearly identical to when we checked for enemy 2, but for 100% test coverage we need to test every case in our switch statement.

ENEMYUI COMPONENT TDD

33. Our next step is to implement our case 3 inside of our switch statement,

```
switch(this.mobCounter){
  case 2:
    switch(this.mobOrder.mob2){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
    break;
  case 3:
    switch(this.mobOrder.mob3){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
    break;
```

34. Lastly, we need to implement 1 more to check for if it's enemy 4.

ENEMYUI COMPONENT TDD

35. Our next step is to implement our case 3 inside of our switch statement,

```
switch(this.mobCounter){
  case 2:
    switch(this.mobOrder.mob2){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
    break;
  case 3:
    switch(this.mobOrder.mob3){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
    break;
```

Lastly, we need to implement 1 more to check for if it's enemy 4.

ENEMYUI COMPONENT TDD

36. To implement our tests, follow what we already have and you'll end up with,

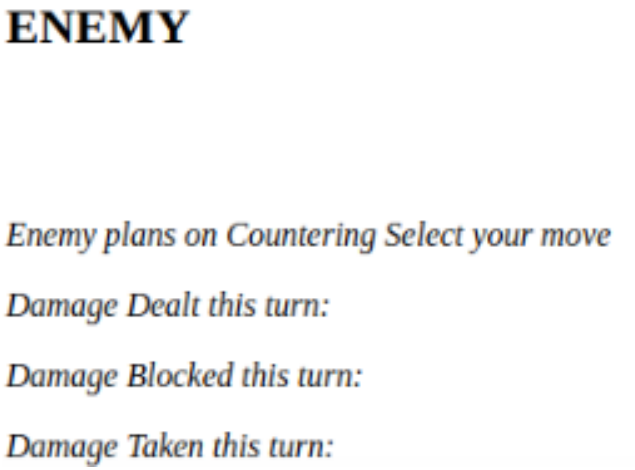
```
it('should be Fighter on mobCounter = 4', () => {
  component.mobCounter = 3;
  component.mobOrder = mockMobOrderMob4Fighter;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(4);
  expect(component.currentImage).toEqual('assets/images/EnemyFighter.png');
});

it('should be BlackMage on mobCounter = 4', () => {
  component.mobCounter = 3;
  component.mobOrder = mockMobOrderHard;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(4);
  expect(component.currentImage).toEqual('assets/images/EnemyBlackMage.png');
});

it('should be Ranger on mobCounter = 4', () => {
  component.mobCounter = 3;
  component.mobOrder = mockMobOrderMob4Ranger;
  component.currentImage = '';
  component.getNextMob();

  expect(component.mobCounter).toEqual(4);
  expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
});
```

Then add our last case to our switch,

```
case 4:
  switch(this.mobOrder.mob4){
    case ClassListEnum.Black_Mage:
      this.currentImage = "assets/images/EnemyBlackMage.png";
      break;
    case ClassListEnum.Fighter:
      this.currentImage = "assets/images/EnemyFighter.png";
      break;
    case ClassListEnum.Ranger:
      this.currentImage = "assets/images/EnemyRanger.png";
  }
```

ENEMYUI COMPONENT TDD

37. So if we look at what we currently have, we have retrieved an enemy list from our Java backend and implemented a way to display the enemies images based on which enemy we're currently on. Therefore, let's take a look back at our wireframe we have,



We're still missing a way to roll the enemies dice, and some way to tell the user what the enemy is planning on doing before the dice are even rolled. Our next step will be to notify our parent of what the enemy plans on doing.

38. To notify our parent we're going to need a few global variables. First, we need some way of triggering this when the parent wants the next move, which is similar to our "nextMob" variable. We then need some way to roll dice (injecting our dice service), and lastly we need to notify our parent the attack the enemy plans on performing.

ENEMYUI COMPONENT TDD

39. Let's start with the global variable we already know that will trigger when the parent notifies us,

```
@Input() enemyAttackTypeEvent: Observable<void>;
```

Next, let's add a global variable to notify our parent. The way we perform this, is by adding an "@Output()" rather than an "@Input()". Furthermore, this output in this case needs to be an "EventEmitter<string>". By setting it to an "EventEmitter<string>", we can pass it directly into our parents methods using the "$event" in the parameter and that value is just a string. With that string our parent component can just display it to the HTML. In HTML, this parameter passing looks like,

```
(attackType)="getEnemyAttack($event)"
```

Which will call our method "getEnemyAttack($event)" inside of our ts file. Our global variable looks like,

```
@Output() attackType = new EventEmitter<string>();
```

Lastly, we need a variable from our constructor for our "diceService.ts", so we can use it's roll method for our dice.

ENEMYUI COMPONENT TDD

40. So now we need to add it into our testing constructor's before we add it to our implementation. Let's start with our "shallow.spec.ts" file. First we need to mock our service, and more specifically the roll method from our diceService.

```
const mockDiceService = {
  roll: jasmine.createSpy('roll').and.returnValues(1,2,3,4,5,6)
}
```

By doing "returnValues", it goes in order of it being called. For example, if we look at our roll method, the first time it's called we'll return 1, and if our spy is called 6 times the last time returns 6.

41. After our mock has been declared and initialized we need to declare a variable for us to reference,

```
let component: EnemyUIComponent;
let fixture: ComponentFixture<EnemyUIComponent>;
let mobOrderService: MobOrderService;
let diceService: DiceService;
```

After that we need to initialize this variable using TestBed since this is shallow testing.

```
beforeEach(() => {
  fixture = TestBed.createComponent(EnemyUIComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
  mobOrderService = TestBed.get(MobOrderService);
  diceService = TestBed.get(DiceService);
});
```

ENEMYUI COMPONENT TDD

42. Our last step to setup our fake diceService to provide our real DiceService in our providers, but then use our mocked value instead.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ EnemyUIComponent ],
    providers: [{provide: DiceService, useValue: mockDiceService},
                {provide: MobOrderService, useValue: mockMobOrderService}]
  })
  .compileComponents();
}));
```

Now with our shallow spec setup to incorporate our diceService, we need to setup our unit testing "spec.ts" file.

43. For teaching purposes we're going to setup our "mockedDiceService" in a different way. Instead we're going to create our own mocked class with our own methods.

```
class mockDiceService {
  constructor() {}

  roll(): Observable<number> {
    return of();
  }

  addDice(response: number, attackType: string): string {
    return 'Does not matter';
  }
}
```

We can make this class directly inside our testing file and with this approach we can "spyOn" rather than creating our own "jasmine.createSpy" and injecting that spy into our real one. This way we inject our own fake class instead.

ENEMYUI COMPONENT TDD

44. So now that we have our fake class setup we can inject it into our components constructor. First we need to setup our variable "diceService" and set it's type to "any" so we can inject it into our real DiceService,

```
fdescribe('EnemyUIComponent', () => {
  let component: EnemyUIComponent;
  let mobOrderService: any;
  let diceService: any;
```

Next we need to set our "diceService" in our "beforeEach" and inject it.

```
beforeEach(() => {
  mobOrderService = mockMobOrderService;
  diceService = new mockDiceService();
  component = new EnemyUIComponent(mobOrderService, diceService);
  component.nextMob = mockNextMob;
});
```

We need to initialize our variable "diceService" to a "new mockDiceService()" because it's a regular class. For spy's we can just set them directly equal to them.

ENEMYUI COMPONENT TDD

45. With our "diceService" faked in both our "shallow.spec.ts" and our unit testing "spec.ts" files, we can go ahead and add it to our implementation constructor. Our file should now look like this.

```
@Output() attackType = new EventEmitter<string>();
@Input() enemyAttackTypeEvent: Observable<void>;
@Input() nextMob: Observable<void>;
mobOrder: MobOrder = new MobOrder;
mobCounter: number;
currentImage: string = "";

constructor(private _mobOrderService: MobOrderService, private _rollDiceService: DiceService) { }
```

So now we have a way to roll dice. We also have a way to return our selected attack to our parent to notify the user and a way for us to pick another attack by taking in input from the parent.

46. All we need now is one more variable so our child component knows which attack it's performing. We need this because we notify our parent of our attackType but we don't store it for our childs "enemy-ui.component.ts" use. If we look ahead and look at our "DiceService", we also have an "addDice" method, this method takes in a dice roll "number" and a attackType "string". Since we're calling this method from our enemyUI, we need a local attackType to be stored. So let's implement it in our variable "enemyAttackType".

```
@Output() attackType = new EventEmitter<string>();
@Input() enemyAttackTypeEvent: Observable<void>;
@Input() nextMob: Observable<void>;
mobOrder: MobOrder = new MobOrder;
mobCounter: number;
currentImage: string = "";
enemyAttackType: string = "";

constructor(private _mobOrderService: MobOrderService, private _rollDiceService: DiceService) { }
```

ENEMYUI CHECKPOINT 2

1. Now that we have our variables setup, let's go ahead and make our tests. First thing, if you recall from setting up our variable "nextMob", our new variable "enemyAttackTypeEvent" is the same thing. Our goal in this checkpoint is to mock and inject the variables the same way we have before.

2. So we can mock it the same way. In our "shallow.spec.ts" file, we just need to create a spy, then inject it into our real one.

3. Next, let's mock it in our unit tests "spec.ts", then inject it into the real one.

ENEMYUI CHECKPOINT 2 SOLUTION

1. Now that we have our variables setup, let's go ahead and make our tests. First thing, if you recall from setting up our variable "nextMob", our new variable "enemyAttackTypeEvent" is the same thing. So we can mock it the same way. In our "shallow.spec.ts" file, we just need to create a spy,

```
const mockEnemyAttackTypeEvent: any = {
  subscribe: jasmine.createSpy('subscribe').and.returnValue(fn => fn())
}
```

And then inject it into our real one,

```
beforeEach(() => {
  fixture = TestBed.createComponent(EnemyUIComponent);
  component = fixture.componentInstance;
  component.nextMob = mockNextMob;
  component.enemyAttackTypeEvent = mockEnemyAttackTypeEvent;
```

Next, let's mock it in our unit tests "spec.ts",

```
const mockEnemyAttackTypeEvent: any = {
  subscribe: jasmine.createSpy('subscribe').and.callFake(fn=>fn())
}
```

And then inject it into the real one.

```
beforeEach(() => {
  mobOrderService = mockMobOrderService;
  diceService = new mockDiceService();
  router =  mockRouter;
  component = new EnemyUIComponent(mobOrderService, diceService);
  component.nextMob = mockNextMob;
  component.enemyAttackTypeEvent = mockEnemyAttackTypeEvent;
```

ENEMYUI COMPONENT TDD

47. Now with our new variable setup we need to setup our "enemyAttackTypeEvent.subscribe", so when our parent triggers it, it's body of code will trigger. Very similar to our "getNextMob" method previously. First thing we need to do is refactor our first 3 tests to spyOn a new method we're going to create called "getEnemyAttackType()".

```
it('should send a get a mobOrder from the service with mob1 = Ranger', () => {
  spyOn(component, 'getNextMob');
  spyOn(component, 'getEnemyAttackType');
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Easy');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(1);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Easy');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
  expect(component.getNextMob).toHaveBeenCalledTimes(1);
  expect(component.getNextMob).toHaveBeenCalledWith();
  expect(component.getEnemyAttackType).toHaveBeenCalledTimes(1);
  expect(component.getEnemyAttackType).toHaveBeenCalledWith();
});
it('should send a get a mobOrder from the service with mob1 = Fighter', () => {
  spyOn(component, 'getNextMob');
  spyOn(component, 'triggerEnemyDiceRoll');
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Medium');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(2);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Medium');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyFighter.png');
  expect(component.getNextMob).toHaveBeenCalledTimes(1);
  expect(component.getNextMob).toHaveBeenCalledWith();
  expect(component.getEnemyAttackType).toHaveBeenCalledTimes(1);
  expect(component.getEnemyAttackType).toHaveBeenCalledWith();
});
it('should send a get a mobOrder from the service with mob1 = Black Mage', () => {
  spyOn(component, 'getNextMob');
  spyOn(component, 'getEnemyAttackType');
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Hard');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(3);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Hard');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyBlackMage.png');
  expect(component.getNextMob).toHaveBeenCalledTimes(1);
  expect(component.getNextMob).toHaveBeenCalledWith();
  expect(component.getEnemyAttackType).toHaveBeenCalledTimes(1);
  expect(component.getEnemyAttackType).toHaveBeenCalledWith();
});
```

ENEMYUI COMPONENT TDD

48. Now let's implement a blank method and verify our tests still pass,

Then add it to our "ngOnInit()" method,

```
getEnemyAttackType() {
}
```

```
ngOnInit() {
  this.mobCounter = 1;
  this._mobOrderService.createMobOrder(sessionStorage.getItem("difficulty")).subscribe( response => {
    this.mobOrder = response;
    switch(this.mobOrder.mob1){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
  });
  this.getNextMob();
  this.getEnemyAttackType();
}
```

Now let's create a new describe for our new tests in our unit testing "spec.ts" file.

```
describe('get the enemys next intended attack type so the player can respond', () => {
```

Our next step is to think about the tests we're going to need. Since the enemy can perform the same actions as the player except "flee", we could simply implement a die roll to determine what our UI will perform. So let's say if the die roll is 1 or 2, the enemy will "Attack". If the die roll is 3 or 4 the enemy will "Counter" and if the die roll is 5 or 6 our UI will simply "Defend".

ENEMYUI COMPONENT TDD

49. Now let's create a test slowly, first thing we need is to have our "enemyAttackTypeEvent.subscribe", so setup our body to be ran when our parent tills us too.

```
describe('get the enemys next intended attack type so the player can respond', () => {
  it('should emit an attack back up to the parent and set the local attackType', () => {

    component.getEnemyAttackType();
    expect(component.enemyAttackTypeEvent.subscribe).toHaveBeenCalled();
```

Now let's implement just enough code to make the test pass,

```
getEnemyAttackType() {
  this.enemyAttackTypeEvent.subscribe( () => {
  });
}
```

Next, we need to roll a die to determine what the UI is going to do, so let's refractor our test to check for it.

```
describe('get the enemys next intended attack type so the player can respond', () => {
  it('should emit an attack back up to the parent and set the local attackType', () => {

    spyOn(diceService, 'roll').and.returnValues(of(1), of(2));
    component.getEnemyAttackType();
    expect(component.enemyAttackTypeEvent.subscribe).toHaveBeenCalled();
    expect(diceService.roll).toHaveBeenCalledTimes(1);
    expect(diceService.roll).toHaveBeenCalledWith();
```

Reminder, our diceService is an actual class in this case and not a spy, so we have to spyOn our roll method. Then from there what we can do is "returnValues(of(1), of(2))". Furthermore, we can check to make sure our method was only called 1 time and it was called with no parameters. Lastly, we need "of()" because our method returns an Observable, so we're returning a body of 1 and 2. The first time we call our method we get 1 and the second time we get 2.

ENEMYUI COMPONENT TDD

50. With our test now including this, let's implement the code to make it pass.

```
getEnemyAttackType() {
  this.enemyAttackTypeEvent.subscribe( () => {
    this._rollDiceService.roll().subscribe(response => {
    })
  });
}
```

Now looking at our current code, we're not using our response yet. This response is a "number" data type from 1-6. So from here we can finish our test if our enemy UI is attacking, which means if the "response" is 1 or 2, we attack. Furthermore, there's 2 things we need to do. If the die is 1 or 2, we need to save it in our global variable "enemyAttackType" as an "Attack", and we need to send an event to our parent using our variable "attackType" which is an "EventEmitter". EventEmitter's have a method called "emit" which will send a message to our parent as a "$event" which was covered on a previous page. Since this new method "emit" needs to be mocked but it's only used in a few tests, let's go ahead and create a mock local to our test that we can inject.

```
describe('get the enemys next intended attack type so the player can respond', () => {
  it('should emit an attack back up to the parent and set the local attackType', () => {
    const mockEventEmitter:any = {
      emit: jasmine.createSpy('emit')
    }
    spyOn(diceService, 'roll').and.returnValues(of(1), of(2));
    component.attackType = mockEventEmitter;

    component.getEnemyAttackType();
```

Directly from our test, we can create a spy, and inject this into our global variable "attackType".

ENEMYUI COMPONENT TDD

51. Now let's finish this test with everything we need,

```
describe('get the enemys next intended attack type so the player can respond', () => {
  it('should emit an attack back up to the parent and set the local attackType', () => {
    const mockEventEmitter:any = {
      emit: jasmine.createSpy('emit')
    }
    spyOn(diceService, 'roll').and.returnValues(of(1), of(2));
    component.enemyAttackType = '';
    component.attackType = mockEventEmitter;

    component.getEnemyAttackType();
    expect(component.enemyAttackTypeEvent.subscribe).toHaveBeenCalled();
    expect(diceService.roll).toHaveBeenCalledTimes(1);
    expect(diceService.roll).toHaveBeenCalledWith();
    expect(component.enemyAttackType).toEqual('Attack');
    expect(mockEventEmitter.emit).toHaveBeenCalledTimes(1);
    expect(mockEventEmitter.emit).toHaveBeenCalledWith('Attack');
```

Adding to what we previously had, we just make sure our new mock "emit" is being called and that it's actually passing an "$event" with a message of "Attack". We also make sure our global variable "enemyAttackType" is being set, we do this by setting it to empty first then checking it after we call our method to see if it's been set to "Attack". Now let's implement the code to make this pass.

```
getEnemyAttackType() {
  this.enemyAttackTypeEvent.subscribe( () => {
    this._rollDiceService.roll().subscribe(response => {
      switch(response){
        case 1:
          this.enemyAttackType = "Attack";
          this.attackType.emit("Attack");
          break;
      }
```

ENEMYUI COMPONENT TDD

52. Next, we can either create a new test to check if the die rolled is a 2 which also results in "Attack" which means we'll have to create a new mock, spyOn, etc... or we can just add it to our current test by refactoring our test again to check if it's 2 since we already have our "spyOn" returning 2 when its called a second time. Our final test could look like this.

```
describe('get the enemys next intended attack type so the player can respond', () => {
  it('should emit an attack back up to the parent and set the local attackType', () => {
    const mockEventEmitter:any = {
      emit: jasmine.createSpy('emit')
    }
    spyOn(diceService, 'roll').and.returnValues(of(1), of(2));
    component.enemyAttackType = '';
    component.attackType = mockEventEmitter;

    component.getEnemyAttackType();
    expect(component.enemyAttackTypeEvent.subscribe).toHaveBeenCalled();
    expect(diceService.roll).toHaveBeenCalledTimes(1);
    expect(diceService.roll).toHaveBeenCalledWith();
    expect(component.enemyAttackType).toEqual('Attack');
    expect(mockEventEmitter.emit).toHaveBeenCalledTimes(1);
    expect(mockEventEmitter.emit).toHaveBeenCalledWith('Attack');

    component.enemyAttackType = '';
    component.getEnemyAttackType();
    expect(diceService.roll).toHaveBeenCalledTimes(2);
    expect(component.enemyAttackType).toEqual('Attack');
    expect(mockEventEmitter.emit).toHaveBeenCalledTimes(2);
    expect(mockEventEmitter.emit).toHaveBeenCalledWith('Attack');

  });
```

We don't need to check to see if our roll and "enemyAttackTypeEvent.subscribe" have been called with no parameters again since it's already covered. Now let's refactor our implementation to make this pass.

ENEMYUI COMPONENT TDD

53. To make our new test pass we need to add a " case 2:" to our switch.

```
getEnemyAttackType() {
  this.enemyAttackTypeEvent.subscribe( () => {
    this._rollDiceService.roll().subscribe(response => {
      switch(response){
        case 1:
        case 2:
          this.enemyAttackType = "Attack";
          this.attackType.emit("Attack");
          break;
      }
    })
  });
}
```

Now all we need to do is implement 2 more tests following our exact same test to check if our UI is performing a "Counter" or performing a "Defend".

54. Inside our same describe let's add a second test to check for "Counter".

```
it('should emit an counter back up to the parent and set the local attackType', () => {
  const mockEventEmitter:any = {
    emit: jasmine.createSpy('emit')
  }
  spyOn(diceService, 'roll').and.returnValues(of(3), of(4));
  component.enemyAttackType = '';
  component.attackType = mockEventEmitter;

  component.getEnemyAttackType();
  expect(diceService.roll).toHaveBeenCalledTimes(1);
  expect(diceService.roll).toHaveBeenCalledWith();
  expect(component.enemyAttackType).toEqual('Counter');
  expect(mockEventEmitter.emit).toHaveBeenCalledTimes(1);
  expect(mockEventEmitter.emit).toHaveBeenCalledWith('Counter');

  component.enemyAttackType = '';
  component.getEnemyAttackType();
  expect(diceService.roll).toHaveBeenCalledTimes(2);
  expect(component.enemyAttackType).toEqual('Counter');
  expect(mockEventEmitter.emit).toHaveBeenCalledTimes(2);
  expect(mockEventEmitter.emit).toHaveBeenCalledWith('Counter');
});
```

The only main difference is that here we have our roll method return a body of 3 and 4 for our Observable.

ENEMYUI COMPONENT TDD

55. Now let's implement the code to make this test pass,

```
getEnemyAttackType() {
  this.enemyAttackTypeEvent.subscribe( () => {
    this._rollDiceService.roll().subscribe(response => {
      switch(response){
        case 1:
        case 2:
          this.enemyAttackType = "Attack";
          this.attackType.emit("Attack");
          break;
        case 3:
        case 4:
          this.enemyAttackType = "Counter";
          this.attackType.emit("Counter");
          break;
```

56. Lastly, let's implement 1 more test to check for "Defend" with a roll of 5 and 6.

```
it('should emit an defend back up to the parent and set the local attackType', () => {
  const mockEventEmitter:any = {
    emit: jasmine.createSpy('emit')
  }
  spyOn(diceService, 'roll').and.returnValues(of(5), of(6));
  component.enemyAttackType = '';
  component.attackType = mockEventEmitter;

  component.getEnemyAttackType();
  expect(diceService.roll).toHaveBeenCalledTimes(1);
  expect(diceService.roll).toHaveBeenCalledWith();
  expect(component.enemyAttackType).toEqual('Defend');
  expect(mockEventEmitter.emit).toHaveBeenCalledTimes(1);
  expect(mockEventEmitter.emit).toHaveBeenCalledWith('Defend');

  component.enemyAttackType = '';
  component.getEnemyAttackType();
  expect(diceService.roll).toHaveBeenCalledTimes(2);
  expect(component.enemyAttackType).toEqual('Defend');
  expect(mockEventEmitter.emit).toHaveBeenCalledTimes(2);
  expect(mockEventEmitter.emit).toHaveBeenCalledWith('Defend');
});
```

ENEMYUI COMPONENT TDD

57. To make this test pass, let's finish implementing our method.

```
getEnemyAttackType() {
  this.enemyAttackTypeEvent.subscribe( () => {
    this._rollDiceService.roll().subscribe(response => {
      switch(response){
        case 1:
        case 2:
          this.enemyAttackType = "Attack";
          this.attackType.emit("Attack");
          break;
        case 3:
        case 4:
          this.enemyAttackType = "Counter";
          this.attackType.emit("Counter");
          break;
        case 5:
        case 6:
          this.enemyAttackType = "Defend";
          this.attackType.emit("Defend");
      }
    })
  });
}
```

If we run all of our tests, you'll notice they still pass.

ENEMYUI COMPONENT TDD

58. Moving forward, we now have a way to update our enemies image when switching enemies, we also have a way to notify the parent/user of which attack our UI is planning on doing. The last step for our UI is to make sure they can roll dice when our player rolls dice, so when our parent sends a "signal" then we can roll dice. We also need to send the dice rolled back to our parent. This should all sound familiar as we'll be following all of our previous steps to implement our next method. So once again, let's create our new global variables for dice rolling. We're need a "@Input()" and a "@Output()". However, this time we're going to be returning a "string[ ]" in our "@Output()" as it will be an array of dice. After these 2 new global variables, we should end with all the following global variables.

```
@Output() enemyDiceRolls = new EventEmitter<string[]>();
@Output() attackType = new EventEmitter<string>();
@Input() enemyDiceRollEvent: Observable<void>;
@Input() enemyAttackTypeEvent: Observable<void>;
@Input() nextMob: Observable<void>;
mobOrder: MobOrder = new MobOrder;
mobCounter: number;
currentImage: string = "";
enemyAttackType: string = "";
```

ENEMYUI CHECKPOINT 3

1. Now that we have 2 more global variables, we need to make sure we mock and inject them from our tests before we write methods that use them. The goal in this checkpoint is to mock and inject them from our testing file.

2. First, create a mock and inject it for our global variable inside of our "shallow.spec.ts" file.;

```
@Input() enemyDiceRollEvent: Observable<void>;
```

3. Secondly, create a mock and inject it for our global variable for our unit tests inside of our "spec.ts" file.

ENEMYUI CHECKPOINT 3 SOLUTION

1. If we follow what we previously had, then the way we implement this for our "shallow.spec.ts" file should consist of a mock,

```
const mockEnemyDiceRollEvent: any = {
  subscribe: jasmine.createSpy('subscribe').and.returnValue(fn => fn())
}
```

Then we inject that mock into our real one.

```
beforeEach(() => {
  fixture = TestBed.createComponent(EnemyUIComponent);
  component = fixture.componentInstance;
  component.nextMob = mockNextMob;
  component.enemyAttackTypeEvent = mockEnemyAttackTypeEvent;
  component.enemyDiceRollEvent = mockEnemyDiceRollEvent;
  fixture.detectChanges();
  mobOrderService = TestBed.get(MobOrderService);
  diceService = TestBed.get(DiceService);
});
```

ENEMYUI CHECKPOINT 3 SOLUTION

2. If we follow what we previously had, then the way we implement this for our "spec.ts" file should consist of a mock,

```
const mockEnemyAttackTypeEvent: any = {
  subscribe: jasmine.createSpy('subscribe').and.callFake(fn=>fn())
}
```

Then we inject that mock into our real one.

```
beforeEach(() => {
  mobOrderService = mockMobOrderService;
  diceService = new mockDiceService();
  component = new EnemyUIComponent(mobOrderService, diceService);
  component.nextMob = mockNextMob;
  component.enemyAttackTypeEvent = mockEnemyAttackTypeEvent;
  component.enemyDiceRollEvent = mockEnemyDiceRollEvent;
});
```

ENEMYUI COMPONENT TDD

59. Now with our mocks created and injected, we need to refactor our first 3 tests yet again. We need to keep doing this because we need our "@Input()" to be setup from our "ngOnInit()". This way we setup our "subscribes" from immediately so they're listening to calls from our parent. So if we refactor our first test again, it should look like;

```
it('should send a get a mobOrder from the service with mob1 = Ranger', () => {
  spyOn(component, 'getNextMob');
  spyOn(component, 'getEnemyAttackType');
  spyOn(component, 'triggerEnemyDiceRoll');
  component.mobCounter = 0;
  sessionStorage.setItem('difficulty', 'Easy');
  component.ngOnInit();
  expect(mobOrderService.createMobOrder).toHaveBeenCalledTimes(1);
  expect(mobOrderService.createMobOrder).toHaveBeenCalledWith('Easy');
  expect(component.mobCounter).toEqual(1);
  expect(component.currentImage).toEqual('assets/images/EnemyRanger.png');
  expect(component.getNextMob).toHaveBeenCalledTimes(1);
  expect(component.getNextMob).toHaveBeenCalledWith();
  expect(component.getEnemyAttackType).toHaveBeenCalledTimes(1);
  expect(component.getEnemyAttackType).toHaveBeenCalledWith();
  expect(component.triggerEnemyDiceRoll).toHaveBeenCalledTimes(1);
  expect(component.triggerEnemyDiceRoll).toHaveBeenCalledWith();
});
```

Follow this same logic and refactor these 2 tests in the same way,

```
it('should send a get a mobOrder from the service with mob1 = Fighter', () => {
it('should send a get a mobOrder from the service with mob1 = Black Mage', () => {
```

ENEMYUI COMPONENT TDD

60. With those 3 tests refactored let's go ahead and add a new empty method to our implementation code "component.ts" and call it from our "ngOnInit" method.

```
triggerEnemyDiceRoll() {
}
```

Now our final implementation of "ngOnInit" should look like,

```
ngOnInit() {
  this.mobCounter = 1;
  this._mobOrderService.createMobOrder(sessionStorage.getItem("difficulty")).subscribe( response => {
    this.mobOrder = response;
    switch(this.mobOrder.mob1){
      case ClassListEnum.Black_Mage:
        this.currentImage = "assets/images/EnemyBlackMage.png";
        break;
      case ClassListEnum.Fighter:
        this.currentImage = "assets/images/EnemyFighter.png";
        break;
      case ClassListEnum.Ranger:
        this.currentImage = "assets/images/EnemyRanger.png";
    }
  });
  this.getNextMob();
  this.getEnemyAttackType();
  this.triggerEnemyDiceRoll();
}
```

ENEMYUI COMPONENT TDD

61. Our next step is to create a new test now for our new method.

```
it('should trigger enemy Dice roll', () => {
  component.triggerEnemyDiceRoll();
  expect(mockEnemyDiceRollEvent.subscribe).toHaveBeenCalledTimes(1);
});
```

We are starting with the simplest test and adding code over time, as this is how TDD is conducted. Now if we add our implementation to make this test pass.

```
triggerEnemyDiceRoll() {
  this.enemyDiceRollEvent.subscribe( () => {
  });
}
```

62. Our next step is to make sure we create a temp variable to keep track of this array of dice rolls. Once this array has been populated we can "emit" this array to our parent using our global variable "enemyDiceRolls" because it's of the data type "EventEmitter". We're also going to need to roll 6 dice. Once these 6 dice are rolled we then need to call our "diceService.addDice(number, string)" method to get the images associated with these dice. So we're going to need to "spyOn" our mocked class we have above in our unit testing "spec.ts" file. To be more exact, we need to "spyOn" both our mocked "roll" method and our mocked "addDice" method.

ENEMYUI COMPONENT TDD

63. So the way our service is setup, if an "Attack" command is sent we roll a die. If that roll is 1, 2, or 3 it's a successful "Attack" die. However, if it's 4, 5, or 6 it does nothing, so this exact behavior is what we can mock and check for. Furthermore, let's finish writing our test, it should look like,

```
it('should trigger enemy Dice roll', () => {
  const mockEnemyDiceRolls: any = {
    emit: jasmine.createSpy('emit')
  }
  spyOn(diceService, 'roll').and.returnValues(of(1), of(4), of(2), of(5), of(3), of(6));
  spyOn(diceService, 'addDice').and.returnValues('Attack', 'Nothing',
                                                 'Attack', 'Nothing',
                                                 'Attack', 'Nothing');
  component.enemyAttackType = 'Attack';
  component.enemyDiceRolls = mockEnemyDiceRolls;
  component.triggerEnemyDiceRoll();
  expect(mockEnemyDiceRollEvent.subscribe).toHaveBeenCalledTimes(1);
  expect(mockEnemyDiceRolls.emit).toHaveBeenCalledTimes(1);
  expect(mockEnemyDiceRolls.emit).toHaveBeenCalledWith(['Attack', 'Nothing',
                                                        'Attack', 'Nothing',
                                                        'Attack', 'Nothing']);
  expect(diceService.roll).toHaveBeenCalledTimes(6);
  expect(diceService.addDice).toHaveBeenCalledTimes(6);
});
```

There's a lot going on here so let's walk through it. First we need to mock our "emit" method from our global variable.

```
@Output() enemyDiceRolls = new EventEmitter<string[]>();
```

Next we need to roll our dice 6 times so we return all 6 possible values. Next, we make sure our "addDice" method returns the strings associated with our 6 possible values. For example, if we return "of(4)" then 'Nothing' happens. Then we need to set our "enemyAttackType" to "Attack" and inject our mocked "emit" into our real one. Then we just check to make sure our emit and our two "spyOn" have been called 6 times to make sure 6 dice are rolled.

ENEMYUI COMPONENT TDD

64. Now to implement our actual code, we could do a loop 6 times, but instead let's do a forkJoin to roll 6 dice then add them to our temp array and then "emit" them back to our parent component.

```
triggerEnemyDiceRoll() {
  this.enemyDiceRollEvent.subscribe( () => {
    let enemiesDiceRolls: string[] = [];

    forkJoin(
      this._rollDiceService.roll(),
      this._rollDiceService.roll(),
      this._rollDiceService.roll(),
      this._rollDiceService.roll(),
      this._rollDiceService.roll(),
      this._rollDiceService.roll()
    ).subscribe( ([dice1, dice2, dice3, dice4, dice5, dice6]) => {
      enemiesDiceRolls.push(this._rollDiceService.addDice(dice1, this.enemyAttackType));
      enemiesDiceRolls.push(this._rollDiceService.addDice(dice2, this.enemyAttackType));
      enemiesDiceRolls.push(this._rollDiceService.addDice(dice3, this.enemyAttackType));
      enemiesDiceRolls.push(this._rollDiceService.addDice(dice4, this.enemyAttackType));
      enemiesDiceRolls.push(this._rollDiceService.addDice(dice5, this.enemyAttackType));
      enemiesDiceRolls.push(this._rollDiceService.addDice(dice6, this.enemyAttackType));
      this.enemyDiceRolls.emit(enemiesDiceRolls);
    })
  });
}
```

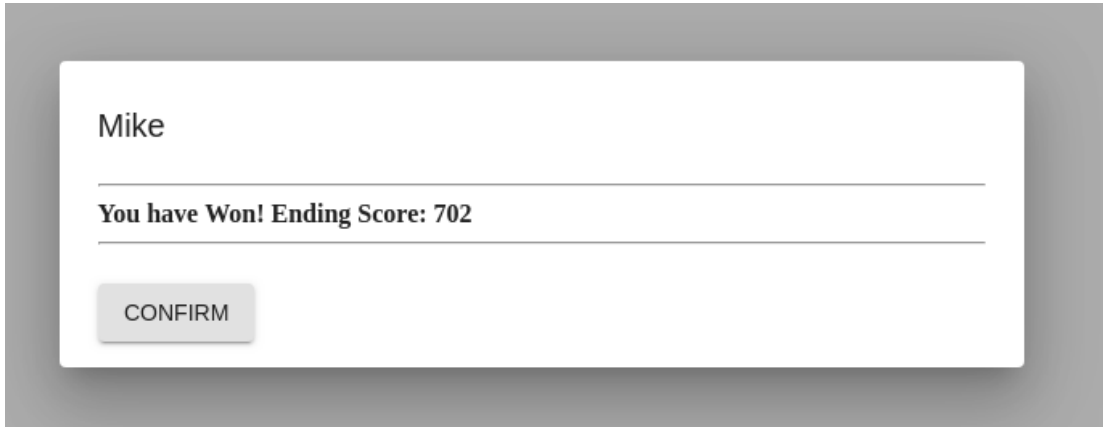
Now you'll test will go from failing to passing. With that array, our parent "dungion-crawler.component.ts" can use this array to compare against the user's dice for damage calculations. Now with this method implemented our enemy UI is now complete. You can compare your TDD against the git branch "feature/enemyUiComponentFinishedTDD".

ANGULAR ENDINGDIALOG COMPONENT TDD

1. EndingDialog Setup Pg 95-97
2. NgOnInit method TDD Pg 98-102
3. Checkpoint 1 (Using a spy/injecting it into a constructor) Pg 100-101
4. OnCloseConfirm method TDD (MatDialogRef method) Pg 103-104

ENDINGDIALOG WIREFRAME

1. Let's check our wireframe to start with.



2. If we look at this information, we'll need the "username" that was entered on our HomePage component, as well as some ending score number.

ENDINGDIALOG SETUP

1. Let's check our current branch to start off,

```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git branch
* develop
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

2. Now let's switch to our "feature/endingDialogComponentUnfinishedTDD" branch so we can begin implementing our code.

```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/endingDialogComponentUnfinishedTDD
Branch 'feature/endingDialogComponentUnfinishedTDD' set up to track remote branch 'feature/endingDialogComponentUnfinishedTDD' from 'origin'.
Switched to a new branch 'feature/endingDialogComponentUnfinishedTDD'
```

ENDINGDIALOG SETUP

3. Now we need to check our tests at the very beginning, so let's run our "ng test" command. You'll notice we have 1 failing test already, this is because of our HTML. The error we're getting is a "mat-dialog-content" error.

```
HeadlessChrome 0.0.0 (Linux 0.0.0) EndingDialogComponent should create FAILED
        'mat-dialog-content' is not a known element:
        1. If 'mat-dialog-content' is an Angular component, then verify that it is part of this module.
        2. If 'mat-dialog-content' is a Web Component then add 'CUSTOM ELEMENTS SCHEMA' to the '@NgModule.schemas' of this component to suppress this message. ("
          <h2 mat-dialog-title>{{username}}</h2>
          <hr>
```

We are getting this error because we are using "mat-dialog-content" and "mat-dialog-actions" inside of our HTML code. Since our "shallow.spec.ts" file uses HTML as will, we need to import this module so that our TestBed can find it.

```
fdescribe('EndingDialogComponent', () => {
  let component: EndingDialogComponent;
  let fixture: ComponentFixture<EndingDialogComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ EndingDialogComponent ],
      imports: [MatDialogModule]
    })
    .compileComponents();
  }));

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

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

Now we see that our starting 2 tests are passing.

ENDINGDIALOG COMPONENT TDD

1. Let's begin by taking a look at our HTML code.

```
<div>
  <h2 mat-dialog-title>{{username}}</h2>
  <hr>
  <mat-dialog-content>
    <strong>{{data}}</strong>
  </mat-dialog-content>
  <hr>
  <mat-dialog-actions>
    <button mat-raised-button (click)="onCloseConfirm()">CONFIRM</button> 
  </mat-dialog-actions>
</div>
```

2. Just looking at this, we need a username global variable since we're using interpolation binding to a variable called "username". So let's just make a global variable.

```
@Component({
  selector: 'app-ending-dialog',
  templateUrl: './ending-dialog.component.html',
  styleUrls: ['./ending-dialog.component.css']
})
export class EndingDialogComponent implements OnInit {
  username: string = "";

  constructor() { }

  ngOnInit() {
  }
}
```

ENDINGDIALOG COMPONENT TDD

3. Our next step, according to our HTML, is another interpolation variable called "data". When it comes to "MatDialog", this data variable comes in through an "@Inject(MAT_DIALOG_DATA)" inside of our constructor which we'll see shortly. Furthermore, we need to accept a "MatDialogRef" inside of our constructor so we can close our Dialog box when some action is performed. The reason we use "MatDialogRef" is because we need a reference to our Dialog Box so we can access it's methods.

4. Before we can fully implement our constructor, we need to change our tests first. Since we're changing the constructor, we need to change both our "shallow.spec.ts" file and our Unit Testing file "spec.ts". So let's start by adding "providers" to our "shallow.spec.ts" file.

```
providers: [
    {provide: MatDialogRef, useValue: ({})},
    {provide: MAT_DIALOG_DATA, useValue: {data: 'Data Info'}}
]
```

By doing this, we're setting our MatDialogRef to an empty object, since all we're trying to do in this file is get the one test to continue passing. Our final "shallow.spec.ts" file looks like;

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ EndingDialogComponent ],
    imports: [MatDialogModule],
    providers: [
        {provide: MatDialogRef, useValue: ({})},
        {provide: MAT_DIALOG_DATA, useValue: {data: 'Data Info'}}
    ]
  })
  .compileComponents();
}));
```

ENDINGDIALOG CHECKPOINT 1

1. Now that our "shallow.spec.ts" file is done, let's go ahead and add it to our unit testing "spec.ts" file. As a quick reminder the reason we're using a "MatDialogRef" is to use it's method, more specifically, we're using it's "close" method so we can close our "MatDialog" box. With this being said we can mock this method functionality as we've done before.

```
const mockDialogRef = {
  close: jasmine.createSpy('close')
}
```

With this mock and our starting code,

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

  beforeEach(() => {
    component = new EndingDialogComponent();
  });

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

Your goal is to setup our component using this mock as our first parameter into our constructor, and using some "data" to be injected into our second parameter.

ENDINGDIALOG CHECKPOINT 1 SOLUTION

1. First thing we need to do is to declare another variable and make it's type "any". This is a necessary step as we cannot inject a spy into a "MatDialogRef".

```
let component: EndingDialogComponent;
let matDialogRef: any;
```

2. Then before each test we need to initialize this variable to our spy and inject it into our constructor.

```
beforeEach(() => {
  matDialogRef = mockDialogRef;
  component = new EndingDialogComponent(matDialogRef);
});
```

3. Lastly, we need to just add some data for our second parameter.

```
beforeEach(() => {
  matDialogRef = mockDialogRef;
  component = new EndingDialogComponent(matDialogRef, 'Data info');
});
```

With all this, we should have.

```
describe('EndingDialogComponent', () => {
  let component: EndingDialogComponent;
  let matDialogRef: any;

  beforeEach(() => {
    matDialogRef = mockDialogRef;
    component = new EndingDialogComponent(matDialogRef, 'Data info');
  });

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

ENDINGDIALOG COMPONENT TDD

5. With our testing files setup, let's go ahead and change our implementation constructor to make our "red" compiler error go to "green".

```
constructor(private _dialogRef: MatDialogRef<EndingDialogComponent>,
  @Inject(MAT_DIALOG_DATA) private data: string) { }
```

6. Now that our constructor is setup we need to actually use our "username" global variable. This information was stored into our "sessionStorage('username')" so we can simply just fetch it out on "ngOnInit". Let's write a test for this,

```
it('should grab the username from sessionStorage on ngOnInit', () =>
  sessionStorage.setItem('username', 'Michael');
  component.ngOnInit();
  expect(component.username).toEqual('Michael');
});
```

If we don't set the item ourselves it will be undefined in our actual code. Then we call out method "ngOnInit". After that we simply just check to see if our components username was set to the stored name to make sure it will be displayed into our HTML "{{username}}" by using interpolation binding. With this test being setup, let's go ahead and create our method.

```
ngOnInit() {
  this.username = sessionStorage.getItem('username');
}
```

ENDINGDIALOG COMPONENT TDD

7. For our next step, let's go ahead and check our HTML.

```
<button mat-raised-button (click)="onCloseConfirm()">CONFIRM</button>
```

We have a button that is calling a method "onCloseConfirm()" when it's clicked, so we can create this method. This method's only job is to close the dialog box, so we can call our spy's close method. Let's create our test for our new method.

```
it('should close the dialog reference', () => {
  component.onCloseConfirm();
  expect(matDialogRef.close).toHaveBeenCalledTimes(1);
  expect(matDialogRef.close).toHaveBeenCalledWith('Confirm');
});
```

With this test, we'll purely checking to make sure our mocked close was called only one time, and that we pass "Confirm" back to the component that opened this dialog box. We pass this back so that our other component can check this string to see how it should process it's body of code. For example, if we open a dialog box and all it displays is "Are you sure you want to delete this?". We could send back a string "yes" if the yes button was clicked, or a "no" if the no button was clicked which would execute different logic.

ENDINGDIALOG COMPONENT TDD

8. With our test now in place we can implement our actual method.

```
onCloseConfirm() {
  this._dialogRef.close("Confirm");
}
```

With that being added our final code looks like.

```
@Component({
  selector: 'app-ending-dialog',
  templateUrl: './ending-dialog.component.html',
  styleUrls: ['./ending-dialog.component.css']
})
export class EndingDialogComponent implements OnInit {
  username: string = "";

  constructor(private _dialogRef: MatDialogRef<EndingDialogComponent>,
    @Inject(MAT_DIALOG_DATA) private data: string) { }

  ngOnInit() {
    this.username = sessionStorage.getItem('username');
  }

  onCloseConfirm() {
    this._dialogRef.close("Confirm");
  }
}
```

That's all we need for our dialog box to display data for us. Now we can check our code against our finished branch.

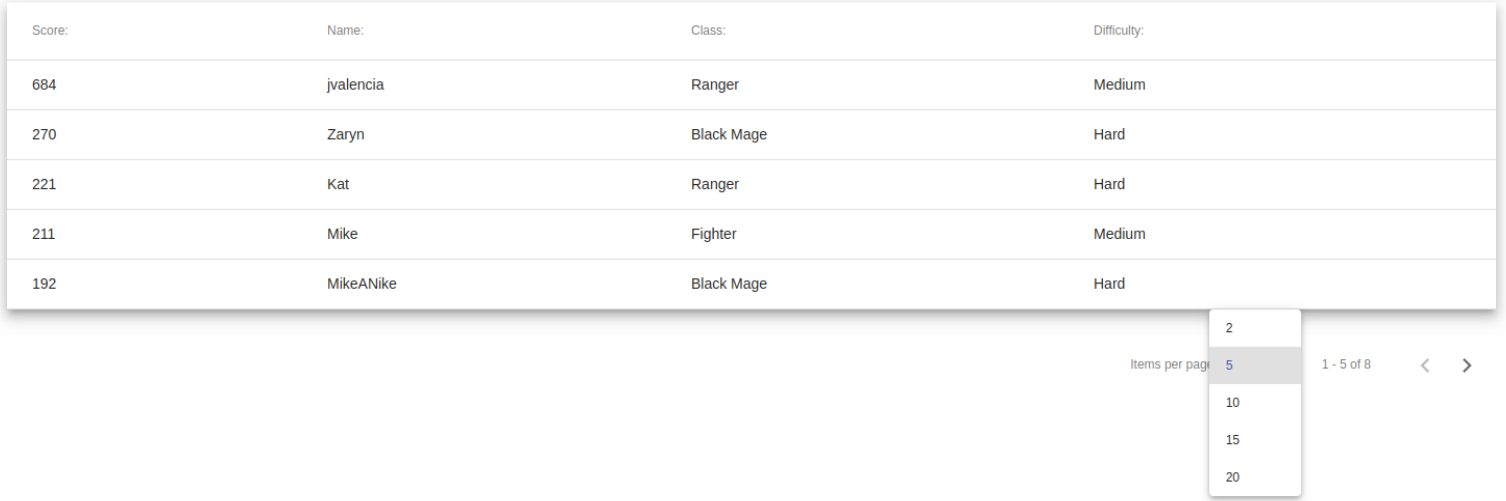
```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/endingDialogComponentFinishedTDD
Branch 'feature/endingDialogComponentFinishedTDD' set up to track remote branch 'feature/endingDialogComponentFinishedTDD' from 'origin'.
Switched to a new branch 'feature/endingDialogComponentFinishedTDD'
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$
```

ANGULAR SCOREBOARD TDD

1. ScoreBoard Setup Pg 106-109
2. Checkpoint 1 (Reading Error Messages and importing modules) Pg 108-109
3. NgOnInit method TDD Pg 110-121
4. Checkpoint 2 (Creating/Sending a Model to the service) Pg 119-120
5. AddScoreToBackend TDD Pg 122-131
6. Checkpoint 3 (Nested Subscribe with TDD) Pg 127-128

SCOREBOARD WIREFRAME

1. For our scoreboard, we need something at the end of the game to display your score and those of other players. We also need it to be sorted so we can see the top players and compare them to ourself.



In this scoreboard we also need our columns to be "Score", "Name", "Class", and "Difficulty". Furthermore, we'll need pagination with sizes of 2, 5, 10, 15, and 20. However, in our HTML most of this is setup for us but we'll need to include some things into our TypeScript file.

SCOREBOARD SETUP

1. First thing we need to go ahead and do is to check our current git branch.

```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git branch
* develop
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

2. Now we're going to go ahead and checkout to our feature branch to begin our implementation of this new component.

```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/scoreboardUnfinishedTDD
Branch 'feature/scoreboardUnfinishedTDD' set up to track remote branch 'feature/scoreboardUnfinishedTDD' from 'origin'.
Switched to a new branch 'feature/scoreboardUnfinishedTDD'
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$
```

With us now in the correct branch, we can continue our setup.

SCOREBOARD CHECKPOINT 1

1. For this checkpoint, if we run our tests from the very beginning with our HTML we currently have our "shallow.spec.ts" file is failing with this error message.

```
HeadlessChrome 0.0.0 (Linux 0.0.0) ScoreBoardComponent shallow should create FAILED
        Can't bind to 'dataSource' since it isn't a known property of 'table'. ("<table mat-table [ERROR ->][dataSource]="dataSource" class="mat-elevation-z8">
          <ng-container matColumnDef="playersScore">
            "): ng:///DynamicTestModule/ScoreBoardComponent.html@0:17
        Can't bind to 'matHeaderRowDef' since it isn't a known property of 'tr'. ("
```

2. Once that is fixed, our test is still failing due to another error.

```
HeadlessChrome 0.0.0 (Linux 0.0.0) ScoreBoardComponent shallow should create FAILED
        Can't bind to 'pageSize' since it isn't a known property of 'mat-paginator'.
        1. If 'mat-paginator' is an Angular component and it has 'pageSize' input, then verify that it is part of this module.
        2. If 'mat-paginator' is a Web Component then add 'CUSTOM ELEMENTS SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.
        3. To allow any property add 'NO ERRORS SCHEMA' to the '@NgModule.schemas' of this component. ("
        <div class="paginator">
```

3. After both of those errors are fixed, our test is still failing do to one more error.

```
HeadlessChrome 0.0.0 (Linux 0.0.0) ScoreBoardComponent shallow should create FAILED
        Error: Found the synthetic property @transitionMessages. Please include either "BrowserAnimationsModule" or "NoopAnimationsModule" in your application.
            at checkNoSyntheticProp (http://localhost:9876/node_modules/@angular/platform-browser/fesm5/platform-browser.js?:1148:1)
```

4. After fixing these 3 errors, we should now have 2 passing tests which concludes our setup.

SCOREBOARD CHECKPOINT 1 SOLUTION

1. The first error message we see is because in our HTML we have a "MatTable". We have to import this module so that when TestBed compiles and runs our CSS, HTML, and TypeScript it can find it. So we can add it to our "imports".

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

2. The second error is because we're using a MatPaginator in our HTML.

```
<div class="paginator">
  <mat-paginator [pageSize]="5" [pageSizeOptions]="[2, 5, 10, 15, 20]">
  </mat-paginator>
</div>
```

So now just add that to our "imports" and that error is fixed.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ ScoreBoardComponent ],
    imports: [MatTableModule, MatPaginatorModule]
  })
  .compileComponents();
}));
```

3. Our last error is litterally stated in the error message directly. "Please include either BrowserAnimationModule or NoopAnimationsModule". So let's add it to fix our last error.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ ScoreBoardComponent ],
    imports: [MatTableModule, MatPaginatorModule, BrowserAnimationsModule]
  })
  .compileComponents();
}));
```

SCOREBOARD COMPONENT TDD

1. First thing we're going to do is create global variables for our scoreboard component. We're doing this because our "MatTable" and "MatPaginator" from our HTML need these to work correctly. Inside of our HTML

```
<table mat-table [dataSource]="dataSource" class="mat-elevation-z8">
```

We have a dataSource from "mat-table" and we need to inject our own "dataSource" into it, for this we can use property binding. Property binding occurs when we take the "mat-table" property "dataSource" and connect (bind) it to a variable of our choice, in this case a global variable "dataSource" inside of our TypeScript file.

```
dataSource: MatTableDataSource<Scoreboard>;
```

To declare it as a "MatTableDataSource", it wants a model that it can follow, in this case our model "Scoreboard" is setup for this task.

```
export class Scoreboard {
    id ?: number;
    playersScore: number;
    playersName: string;
    playersClass: ClassListEnum;
    difficulty: DifficultyEnum;
}
```

This model incorporates everything we need from our wireframe (Pg 105).

SCOREBOARD COMPONENT TDD

2. Now that we have our "dataSource" setup, let's go ahead and create our "displayedColumns" global variable that we need according to our HTML.

```
<tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>
<tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>
```

Our global variable should look like,

```
displayedColumns: string[] = ["playersScore", "playersName", "playersClass", "difficulty"];
```

Each index corresponds to a column in our table, so the names here have to match exactly to the names inside our "ng-container matColumnDef".

```
<ng-container matColumnDef="difficulty">
```

3. Our last global we need right now is a paginator declared to a "MatPaginator". The thing with this paginator, we set it up on the HTML with it's pageSize and pageSizeOptions.

```
<div class="paginator">
  <mat-paginator [pageSize]="5" [pageSizeOptions]="[2, 5, 10, 15, 20]">
  </mat-paginator>
</div>
```

So we have to do a "@ViewChild(MatPaginator)". This will allow us to view our HTML and grab these options, then save them into our variable so we can inject them directly into our "dataSource" paginator to filter through the table.

```
@ViewChild(MatPaginator) paginator: MatPaginator;
```

SCOREBOARD COMPONENT TDD

4. With these global variables our overall code should look like,

```
@Component({
  selector: 'app-score-board',
  templateUrl: './score-board.component.html',
  styleUrls: ['./score-board.component.css']
})
export class ScoreBoardComponent implements OnInit {

  displayedColumns: string[] = ["playersScore", "playersName", "playersClass", "difficulty"];
  dataSource: MatTableDataSource<Scoreboard>;

  @ViewChild(MatPaginator) paginator: MatPaginator;
  constructor() { }

  ngOnInit() {
  }
```

5. This component's only job is to display the overall scoreboard while adding your score to it. For example, this is displayed at the very end of the game once the user has finished. That user's information needs to be sent to the Java Backend to be saved to the database. Once that information is saved, we then get back the entire scoreboard from our Java backend which will include our most recently added player.

6. For this to happen, our "ngOnInit" method needs to build a model "Scoreboard" so that we can send it to our service "ScoreBoardService" method "addScore". So we can begin thinking about what we need.

SCOREBOARD COMPONENT TDD

7. As a reminder, our model looks like,

```
export class Scoreboard {
    id ?: number;
    playersScore: number;
    playersName: string;
    playersClass: ClassListEnum;
    difficulty: DifficultyEnum;
}
```

With the "id" being optional, denoted by the '?', we only need to get the score, name, class, and difficulty the user had selected at the beginning. From other portions of this guide, our "playersClass" is saved into "sessionStorage", so is the "difficulty" selected. We also have our "playersName" stored into "sessionStorage". However, we're still missing our "playersScore" which is implemented in the next section of this guide, for now, we can still assume it's stored in "sessionStorage" for TDD purposes.

8. So now we can begin by building our model inside of our method "ngOnInit". The tricky thing here, all the "sessionStorage" variables can be local to our method, so for us to create a test, we're going to be using our mocked file "mockScoreBoard", and setting our "sessionStorage" variables to match these mocked values. The way we're going to test this functionality is by creating another method "addScoreToBackend" and seeing if this method is called with our mocked model.

SCOREBOARD COMPONENT TDD

9. The main reason this is tricky is because we have 3 classes, and 3 difficulty settings to test for inside of a method with all local variables. So our 3 mocked values need to include all cases. For example, mockScoreBoardOne could have a class as "Fighter" and difficulty as "Hard" so therefore, we would want mockScoreBoardTwo to have class "BlackMage" or "Ranger" and difficulty to be "Easy" or "Medium". Overall we're going to need 3 tests to cover all this functionality while checking to make sure our other empty method "addScoreToBackend" is being called with our mocked models by purely setting "sessionStorage" variables.

10. Let's begin by writing a test using the data inside of mockScoreBoardOne located in "mocks/mockScoreBoard.ts".

```
export let mockScoreBoardOne = {
    playersScore: 951,
    playersName: "Wargods3",
    playersClass: ClassListEnum.Fighter,
    difficulty: DifficultyEnum.Hard
}
```

By doing this, in our test we can set our "sessionStorage" variables to match this data. For example, "sessionStorage.setItem('playersClass', 'Fighter');" would replicate this playersClass, so we just need to do that for all 4 variables.

SCOREBOARD COMPONENT TDD

11. So following this ideology the beginning of our first test could look like,

```
it('should make a player model and send a request', () => {
  sessionStorage.setItem('playersClass', 'Fighter');
  sessionStorage.setItem('difficulty', 'Hard');
  sessionStorage.setItem('username', 'Wargods3');
  sessionStorage.setItem('score', '951');
});
```

This data matchs the data inside of our mocked file. So now let's pretend we have a empty method "addScoreToBackend" so we can "spyOn".

```
it('should make a player model and send a request', () => {
  sessionStorage.setItem('playersClass', 'Fighter');
  sessionStorage.setItem('difficulty', 'Hard');
  sessionStorage.setItem('username', 'Wargods3');
  sessionStorage.setItem('score', '951');
  spyOn(component, 'addScoreToBackend');
  component.ngOnInit();
});
```

With this "spyOn" we're effectively mocking this other method "addScoreToBackend". Now all we need to do is finish our test.

```
it('should make a player model and send a request', () => {
  sessionStorage.setItem('playersClass', 'Fighter');
  sessionStorage.setItem('difficulty', 'Hard');
  sessionStorage.setItem('username', 'Wargods3');
  sessionStorage.setItem('score', '951');
  spyOn(component, 'addScoreToBackend');
  component.ngOnInit();
  expect(component.addScoreToBackend).toHaveBeenCalledTimes(1);
  expect(component.addScoreToBackend).toHaveBeenCalledWith(mockScoreBoardOne);
});
```

If our "ngOnInit" method is setup properly the model we create from our "sessionStorage" variables and send to our other method will match.

SCOREBOARD COMPONENT TDD

12. Our first step is to turn our test from "red" compiler error to a "green" state. To do this simply just add a blank signature.

```
addScoreToBackend(currScore: Scoreboard) {
}
```

13. Next if we run this test it will fail. We need to implement just enough code to make this test pass. First let's get our "playersClass" and "difficulty" from session storage.

```
ngOnInit() {
  let playersClass = sessionStorage.getItem('playersClass');
  let difficulty = sessionStorage.getItem('difficulty');
}
```

There's a problem with this though. This will return two strings, but our model takes a "ClassListEnum" and a "DifficultyEnum". We'll need to also convert them for our model format so we can send this to our Java backend. To convert them we can just create "switch" statements on our two strings.

```
let convertClass: ClassListEnum;
let convertDifficulty: DifficultyEnum;
switch(playersClass){
  case "Fighter":
    convertClass = ClassListEnum.Fighter;
    break;
}
switch(difficulty){
  case "Hard":
    convertDifficulty = DifficultyEnum.Hard;
}
```

SCOREBOARD COMPONENT TDD

14. Lastly, all we need to do is convert this information into a model and pass it into our other method.

```
ngOnInit() {
  let playersClass = sessionStorage.getItem('playersClass');
  let difficulty = sessionStorage.getItem('difficulty');

  let convertClass: ClassListEnum;
  let convertDifficulty: DifficultyEnum;
  switch(playersClass){
    case "Fighter":
      convertClass = ClassListEnum.Fighter;
      break;
  }
  switch(difficulty){
    case "Hard":
      convertDifficulty = DifficultyEnum.Hard;
  }
  let currScore: Scoreboard = {
    playersScore : parseInt(sessionStorage.getItem('score')),
    playersName : sessionStorage.getItem('username'),
    playersClass : convertClass,
    difficulty : convertDifficulty
  };
  this.addScoreToBackend(currScore);
}
```

By doing this, the "currScore" object will equal our "mockScoreBoardOne" and thus our test will now pass.

SCOREBOARD COMPONENT TDD

15. With this logic now implemented, let's create another test and add onto our "switch" statements. This test is going to use our mockScoreBoardTwo instead as it is using a different "difficulty" and "playersClass"

```
it('should make a player model and send a request', () => {
  sessionStorage.setItem('playersClass', 'BlackMage');
  sessionStorage.setItem('difficulty', 'Easy');
  sessionStorage.setItem('username', 'RaiseTheRoof');
  sessionStorage.setItem('score', '431');
  spyOn(component, 'addScoreToBackend');
  component.ngOnInit();
  expect(component.addScoreToBackend).toHaveBeenCalledTimes(1);
  expect(component.addScoreToBackend).toHaveBeenCalledWith(mockScoreBoardTwo);
});
```

This test will check the difficulty "Easy" and class "BlackMage" cases inside of our "switch" statements. So run this test, watch it fail, then now let's implement just enough code to make it pass.

```
ngOnInit() {
  let playersClass = sessionStorage.getItem('playersClass');
  let difficulty = sessionStorage.getItem('difficulty');

  let convertClass: ClassListEnum;
  let convertDifficulty: DifficultyEnum;
  switch(playersClass){
    case "Fighter":
      convertClass = ClassListEnum.Fighter;
      break;
    case "BlackMage":
      convertClass = ClassListEnum.Black_Mage;
  }
  switch(difficulty){
    case "Easy":
      convertDifficulty = DifficultyEnum.Easy;
      break;
    case "Hard":
      convertDifficulty = DifficultyEnum.Hard;
  }
  let currScore: Scoreboard = {
    playersScore : parseInt(sessionStorage.getItem('score')),
    playersName : sessionStorage.getItem('username'),
    playersClass : convertClass,
    difficulty : convertDifficulty
  };
  this.addScoreToBackend(currScore);
}
```

SCOREBOARD CHECKPOINT 2

1. For this checkpoint, we're given the following model.

```
export let mockScoreBoardThree = {
    playersScore: 689,
    playersName: "KitDaKat",
    playersClass: ClassListEnum.Ranger,
    difficulty: DifficultyEnum.Medium
}
```

Create a test to check to see if our method "ngOnInit" is sending the correct Object using the implementation we currently have as well as the two previous tests.

SCOREBOARD CHECKPOINT 2 SOLUTION

1. Following our previous two tests with this model our new test would look like,

```
it('should make a player model and send a request', () => {
  sessionStorage.setItem('playersClass', 'Ranger');
  sessionStorage.setItem('difficulty', 'Medium');
  sessionStorage.setItem('username', 'KitDaKat');
  sessionStorage.setItem('score', '689');
  spyOn(component, 'addScoreToBackend');
  component.ngOnInit();
  expect(component.addScoreToBackend).toHaveBeenCalledTimes(1);
  expect(component.addScoreToBackend).toHaveBeenCalledWith(mockScoreBoardThree);
});
```

SCOREBOARD COMPONENT TDD

16. With our new test implemented, let's go ahead and finish implementing our method "ngOnInit".

```
switch(playersClass){
  case "Fighter":
    convertClass = ClassListEnum.Fighter;
    break;
  case "Ranger":
    convertClass = ClassListEnum.Ranger;
    break;
  case "BlackMage":
    convertClass = ClassListEnum.Black_Mage;
}
switch(difficulty){
  case "Easy":
    convertDifficulty = DifficultyEnum.Easy;
    break;
  case "Medium":
    convertDifficulty = DifficultyEnum.Medium;
    break;
  case "Hard":
    convertDifficulty = DifficultyEnum.Hard;
}
```

With this implemented we now have a model setup with the user's information, now we just need to make some calls to our service inside of our currently empty method "addScoreToBackend".

SCOREBOARD COMPONENT TDD

17. Before we begin implementing our next method, we're going to need to make calls to our service, so we need to inject it into our constructor. Before we can do that, we need to setup our tests first to incorporate an adjustment to the constructor. First let's setup our "shallow.spec.ts" file. Since our service has two methods we're going to be using, "addScore" and "getAllScores", we're going to need to create an Object with 2 spies to copy the logic from the real methods. In our real service, these methods return an Observable, so we need to return "of(object)". Observables have a subscribe method that we need to run when it get's a response. By using "of(object)" this returns that response so the subscribes body of code will actually run.

```
const mockScoreBoardService = {
  addScore: jasmine.createSpy('addScore').and.returnValue(of(mockScoreBoardOne)),
  getAllScores: jasmine.createSpy('getAllScores').and.returnValue(of(mockScoreBoardList))
}
```

18. With our object created, we need to use TestBed to get the real service, but then provide it with our fake Object instead. To do this, let's begin by creating a new variable we can reference.

```
let component: ScoreBoardComponent;
let fixture: ComponentFixture<ScoreBoardComponent>;
let service: ScoreBoardService;
```

Notice it's the real service, in a "shallow.spec.ts" file we have to inject our mocks in a different way. Our next step is to tell "TestBed" to get this service.

```
beforeEach(() => {
  fixture = TestBed.createComponent(ScoreBoardComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
  service = TestBed.get(ScoreBoardService);
});
```

SCOREBOARD COMPONENT TDD

19. Our last step inside of our "shallow.spec.ts" file is to provide it inside of our providers since it's a service.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ ScoreBoardComponent ],
    imports: [MatTableModule, MatPaginatorModule, BrowserAnimationsModule],
    providers: [{provide: ScoreBoardService, useValue: mockScoreBoardService}]
  })
  .compileComponents();
}));
```

However, when we provide the real service, we specifically tell it to use our object we mocked in replace of the real service. So when TestBed get's our "ScoreBoardService" it's actually getting our "mockScoreBoardService" instead, which contains our two spies.

20. Now we just need to create our mocked service and inject it into our unit testing "spec.ts" file. To demonstrate that we can do this without making a global variable "mockScoreBoardService" we're going to create our variable "service" first. However, our variable still needs to be of type "any" so that we can inject it into our component's constructor "ScoreBoardService", otherwise we have to use the real service which we don't want to do for the purposes of TDD.

```
fdescribe('ScoreBoardComponent', () => {
  let component: ScoreBoardComponent;
  let service: any;
```

SCOREBOARD COMPONENT TDD

21. With our service being declared as "any", we can now set it manually to an object that has our spies within.

```
beforeEach(() => {
  service = {
    addScore: jasmine.createSpy('addScore').and.returnValue(of(fn => fn())),
    getAllScores: jasmine.createSpy('getAllScores').and.returnValue(of(mockScoreBoardList))
  }
 component = new ScoreBoardComponent(service);
});
```

After we initialize our "service" variable to an object with 2 spies we can inject it into our constructor. As a quick reminder, these 2 spies will copy the behavior of our real "ScoreBoardService", more specifically, our 2 methods "addScore" and "getAllScores".

22. Now let's make our testing file turn green by implementing this into our constructor.

```
displayedColumns: string[] = ["playersScore", "playersName", "playersClass", "difficulty"];
dataSource: MatTableDataSource<Scoreboard>;

@ViewChild(MatPaginator) paginator: MatPaginator;
constructor(private _scoreBoardService: ScoreBoardService) { }
```

With our constructor now being setup, we can start writing another test for our method "addScoreToBackend".

SCOREBOARD COMPONENT TDD

23. Our new method is going to need to make 2 calls. It first is going to need to call our service's method "addScore", so we can add our created model (aka our most recent player), to the Java backend's list of scores. We're then going to need to make another call to this service's method "getAllScores", to then retrieve the entire scoreboard with the most recent players information within it.

24. To accomplish this, we can do a nested subscribe. So basically, we can make one call "addScore", then within it's body of code, we can do another call "getAllScores" so that way it waits for the "addScore" to come back with a response (in this case just a status code from our Java backend) before we send our second request. Lastly, once our second request receives a response, we need to set our "MatTable" dateSource to the data, and to set the "MatTable" paginator so we can filter through this data.

25. So let's begin by writing an empty test.

```
it('should add the current score, then get all scores', () => {
});
```

Now we can begin by writing our test.

SCOREBOARD COMPONENT TDD

26. So for this test, we need to start with our most basic test then add just enough implementation to make it pass. As the previous page described, we need to call "addScore" first, so let's do that first.

```
it('should add the current score, then get all scores', () => {
  component.addScoreToBackend(mockScoreBoardOne);
  expect(service.addScore).toHaveBeenCalledTimes(1);
  expect(service.addScore).toHaveBeenCalledWith(mockScoreBoardOne);
});
```

All we're doing is calling our method with one of our mocked scoreboard models and then checking to see if our spy "addScore" was called once and if it was passed our model. If we run the test it will fail, so now add the implementation.

```
addScoreToBackend(currScore: Scoreboard) {
  this._scoreBoardService.addScore(currScore).subscribe( () => {
  });
}
```

This test will now pass. Next we can change our test to check for a second call.

SCOREBOARD CHECKPOINT 3

1. For this checkpoint, we're going to add more to our current test,

```
it('should add the current score, then get all scores', () => {
});
```

To check to see if our second spy "getAllScores" is being called. The testing file will then look like.

```
it('should add the current score, then get all scores', () => {
  component.addScoreToBackend(mockScoreBoardOne);
  expect(service.addScore).toHaveBeenCalledTimes(1);
  expect(service.addScore).toHaveBeenCalledWith(mockScoreBoardOne);
  expect(service.getAllScores).toHaveBeenCalledTimes(1);
  expect(service.getAllScores).toHaveBeenCalled();
});
```

2. If we run this test now, it fails. Our goal in this checkpoint is to see if we can find a way to make this test pass.

SCOREBOARD CHECKPOINT 3 SOLUTION

1. All we need to do is nest a second "subscribe" call into our first "subscribe".

```
addScoreToBackend(currScore: Scoreboard) {
  this._scoreBoardService.addScore(currScore).subscribe( () => {
    this._scoreBoardService.getAllScores().subscribe( response => {
    });
  });
}
```

If we run our test now it will pass. Both of our spies are being used properly and return some value or function so that our "subscribe" method body is actually executed.

SCOREBOARD COMPONENT TDD

27. Now that our nested call is setup, we just need to add this "response" which is an array or list of models into our "MatTable" dataSource so we can display this information. The way we can check this behavior is by making a mock "dataSource" with a "paginator" variable since we set this to our HTML's paginator inside of our TypeScript. We need to mock and inject it before our method call so that we can verify that it changes after the method is called.

```
it('should add the current score, then get all scores', () => {
  const mockDataSource: any = {
    paginator: 'Does not matter'
  };
  component.dataSource = mockDataSource;
  component.addScoreToBackend(mockScoreBoardOne);
  expect(service.addScore).toHaveBeenCalledTimes(1);
  expect(service.addScore).toHaveBeenCalledWith(mockScoreBoardOne);
  expect(service.getAllScores).toHaveBeenCalledTimes(1);
  expect(service.getAllScores).toHaveBeenCalled();
});
```

SCOREBOARD COMPONENT TDD

28. With our mocked "dataSource" injected into the real one, we can now check to see if it changes after our "component.addScoreToBackend()" is called.

```
it('should add the current score, then get all scores', () => {
  const mockDataSource: any = {
    paginator: 'Does not matter'
  };
  component.dataSource = mockDataSource;
  component.addScoreToBackend(mockScoreBoardOne);
  expect(service.addScore).toHaveBeenCalledTimes(1);
  expect(service.addScore).toHaveBeenCalledWith(mockScoreBoardOne);
  expect(service.getAllScores).toHaveBeenCalledTimes(1);
  expect(service.getAllScores).toHaveBeenCalled();
  expect(component.dataSource.data).toBe(mockScoreBoardList);
  expect(component.dataSource.paginator).toEqual(component.paginator);
});
```

Here we need to make sure that we set the data to our mocked list, which we returned from our spy "getAllScores", and that our "dataSource.paginator" is being set to our "component.paginator" which is the "@ViewChild" global variable. If we run our test now, it fails again. Lastly, let's go ahead and make this test pass by implementing our code.

```
addScoreToBackend(currScore: Scoreboard) {
  this._scoreBoardService.addScore(currScore).subscribe( () => {
    this._scoreBoardService.getAllScores().subscribe( response => {
      this.dataSource = new MatTableDataSource(response);
      this.dataSource.paginator = this.paginator;
    });
  });
}
```

Now our test will pass and that concludes our scoreboard.

SCOREBOARD COMPONENT TDD

29. Since we've implemented our last method, we can now check our code against the finished branch.

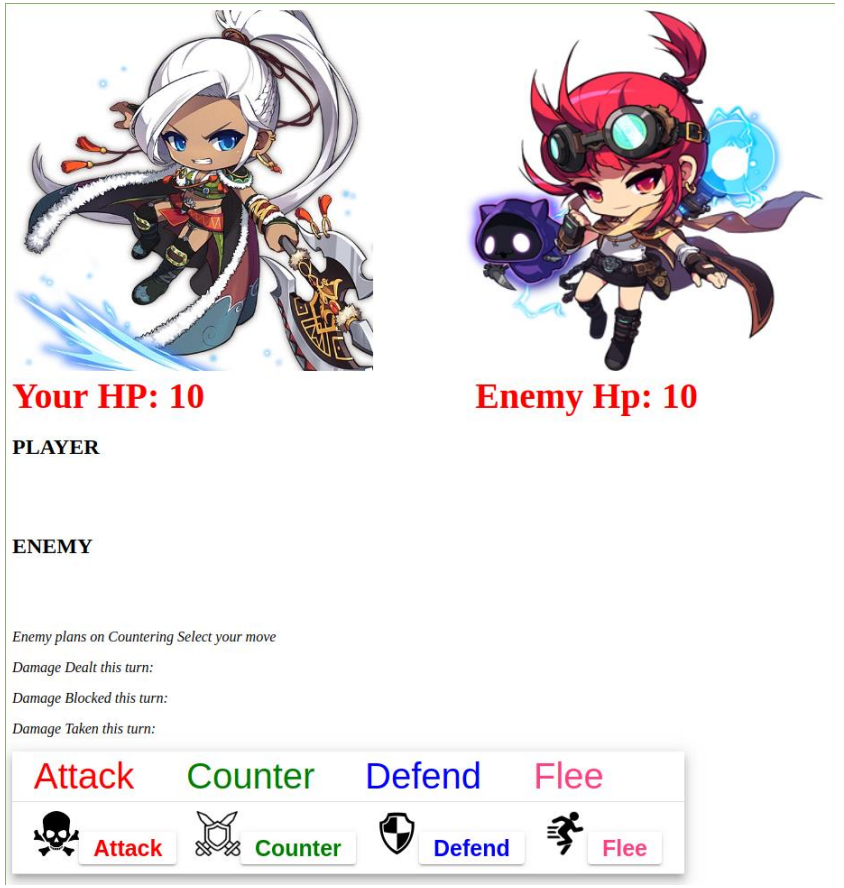
```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/scoreboardFinishedTDD
Branch 'feature/scoreboardFinishedTDD' set up to track remote branch 'feature/scoreboardFinishedTDD' from 'origin'.
Switched to a new branch 'feature/scoreboardFinishedTDD'
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$
```

ANGULAR DUNGIONCRAWLER COMPONENT TDD

1. DungionCrawler Wireframe Pg 133-133
2. DungionCrawler Setup Pg 134-140
3. Checkpoint 1 (Mocking a Component using the "selector" for HTML) Pg 138-139
4. NgOnInit Method TDD (With a Delay) Pg 141-153
5. delay Method TDD Pg 147-149
6. getPlayersAttack Method TDD Pg 154-159
7. Checkpoint 2 (SpyOn an Empty Method TDD) Pg 156-157
8. getEnemyAttack Method TDD Pg 160-162
9. getEnemyDiceRolls Method TDD Pg 162-164
10. playerRoll method TDD (With a forkjoin) Pg 164-170
11. openEndingDialog method TDD Pg 171-180
12. Checkpoint 3 (Mocked values/methods and checking if called) Pg 178-180
13. calculateDiceRolls method TDD Pg 181-189
14. calculateDamage method TDD Pg 190-216
15. Checkpoint 4 (SpyOn a Method with Parameters) Pg 214-216

DUNGIONCRAWLER COMPONENT WIREFRAME

1. As a quick reminder, our wireframe looks like,



2. In this section we are creating the core logic for our game, mainly pertaining to the points awarded to the player for actions committed/damage done. This is the parent component to all of our other components which means they're all injected into this component's HTML. Furthermore, previous sections that had "@Output" variables, are having their information sent to this component. Because of all of this, this section will be the longest and most challenging.

DUNGIONCRAWLER SETUP

1. First let's check the branch we're currently on.

```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git branch
* develop
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

2. Now let's checkout to the starting point for this section.

```
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$ git checkout feature/dungionCrawlerComponentUnfinishedTDD
Switched to branch 'feature/dungionCrawlerComponentUnfinishedTDD'
Your branch is up to date with 'origin/feature/dungionCrawlerComponentUnfinishedTDD'.
cdelabs@studios-plano-01-06-02:~/workspace/enablementorium$
```

Next, let's run our initial two tests and see if they both pass. If our HTML was blank to start with they should, but since we're given the HTML our one test inside of "shallow.spec.ts" will fail. This test is failing because it's using "TestBed" to create the component for us, which uses the components CSS, HTML, and TypeScript files.

3. The reason this test is failing is because we have two other components injected into our HTML, so when TestBed runs the HTML it's attempting to use the real components that where injected. When it comes to our HTML, we need to mock these other components since we're only testing this component. In our pure unit testing "spec.ts" file we don't need to worry about any of this since it only tests our TypeScript file.

DUNGIONCRAWLER SETUP

4. So our first step is to fix this broken test. To do this, we need to understand our HTML code. Let's begin by looking at line 5 in our HTML.

```
<app-enemy-ui [nextMob] = "triggerNextMob.asObservable()" [enemyAttackTypeEvent] = "triggerEnemyAttackType.asObservable()"
[enemyDiceRollEvent]="triggerEnemyDiceRolls.asObservable()" (attackType)="getEnemyAttack($event)"
(enemyDiceRolls)="getEnemyDiceRolls($event)"></app-enemy-ui>
```

Since this component is injected into our HTML we need to mock it. To properly mock a component like this, we need the same "@Input" variables in our mock. As a quick reminder from the "enemyUI" section of this guide the variables we had for these where.

```
@Input() enemyDiceRollEvent: Observable<void>;
@Input() enemyAttackTypeEvent: Observable<void>;
@Input() nextMob: Observable<void>;
```

If the variables aren't the same including the declarations we can get the following error.

```
HeadlessChrome 0.0.0 (Linux 0.0.0) DungionCrawlerComponent shallow should create FAILED
      Can't bind to 'nextMob' since it isn't a known property of 'app-enemy-ui'.
      1. If 'app-enemy-ui' is an Angular component and it has 'nextMob' input, then verify that it is part of this module.
      2. If 'app-enemy-ui' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.
```

5. The reason we get this error is because we use different types of binding in our injected component, and without these variables it cannot bind to an unknown variable. Notice how it says we can add "CUSTOM_ELEMENTS_SCHEMA", don't do this. By adding this schema, you're basically attempting to turn off TestBed's HTML feature in a hacky way. If you don't want to include HTML either only use pure unit testing "spec.ts" files or mock the components that are injected.

DUNGIONCRAWLER SETUP

6. So the way we go about creating this mock is by setting it up inside our "shallow.spec.ts" file, where the TestBed and our failing test are located. Then we create our own class at the top.

```
class EnemyUIComponent{
  @Output() enemyDiceRolls = new EventEmitter<string[]>();
  @Output() attackType = new EventEmitter<string>();
  @Input() enemyDiceRollEvent: Observable<void>;
  @Input() enemyAttackTypeEvent: Observable<void>;
  @Input() nextMob: Observable<void>;
}

fdescribe('DungionCrawlerComponent shallow', () => {
  let component: DungionCrawlerComponent;
  let fixture: ComponentFixture<DungionCrawlerComponent>;
```

For demonstration purposes, we're going to include the "@Output" variables as well, but since they're just binded to a method we don't need them. However, if we run the tests now, they still fail for the same reason. This is only part of the solution. Notice that at the beginning of the HTML on line 5, we have `<app-enemy-ui` which is how we know the component we're injecting here. If we take a look at any component, it's declared with a "@Component" on the top so we know it's a component.

7. Each component starts with three fields even though we don't need them all. In this case, all we care about is the "selector" field since this is our reference for injection. Another field we could include would be "template" and that allows us to write HTML directly into our component.

DUNGIONCRAWLER SETUP

8. So with us knowing about the "@Component" annotation let's add it onto the top of our mocked class inside our "shallow.spec.ts" file.

```
@Component({
  selector: 'app-enemy-ui',
  template: ''
})
class EnemyUIComponent{
  @Output() enemyDiceRolls = new EventEmitter<string[]>();
  @Output() attackType = new EventEmitter<string>();
  @Input() enemyDiceRollEvent: Observable<void>;
  @Input() enemyAttackTypeEvent: Observable<void>;
  @Input() nextMob: Observable<void>;
}
```

This "selector" has to match the real one, aka it has to match the HTML one; in this case "app-enemy-ui". With this now being setup it can bind to these variables instead getting rid of that error.

9. If we run our tests now, they still fail. Due to the same error but coming from another injection.

DUNGIONCRAWLER CHECKPOINT 1

1. We solved one error by creating a mocked component, but we still have a similar error coming from another injected component.

2. If we look at our HTML code on line 24,

```
<app-attack-buttons (currentPlayerCommand)="getPlayersAttack($event)"></app-attack-buttons>
```

We're injecting another component with an "@Output" variable "currentPlayerCommand" that is bound to a method.

3. For this checkpoint, the goal is to come up with a way to create another mocked component for our "shallow.spec.ts" file so that our test passes.

DUNGIONCRAWLER CHECKPOINT 1 SOLUTION

1. If we follow the same logic from our previous mocked class "EnemyUiComponent" inside our "shallow.spec.ts" file, we should end up with another mocked class that looks like.

```
class AttackButtonsComponent{
}
```

We don't need any variables, because as stated before, "@Output" are just binded to methods so we don't need them. We don't need them because "@Output" just emits a "string" through the parameter "$event". But in our tests we don't care about this functionality since we can just call those methods with a string. We only need to bind to the "@Input" variables since they actually bind to variables that are of type "Subject" which we'll cover later.

2. The overall solution with our "@Component" looks like this.

```
@Component({
  selector: 'app-attack-buttons',
  template: ''
})
class AttackButtonsComponent{
}
```

DUNGIONCRAWLER SETUP

10. Now with our second injected component being mocked our overall code should look similar to.

```
@Component({
  selector: 'app-enemy-ui',
  template: ''
})
class EnemyUIComponent{
  @Output() enemyDiceRolls = new EventEmitter<string[]>();
  @Output() attackType = new EventEmitter<string>();
  @Input() enemyDiceRollEvent: Observable<void>;
  @Input() enemyAttackTypeEvent: Observable<void>;
  @Input() nextMob: Observable<void>;
}

@Component({
  selector: 'app-attack-buttons',
  template: ''
})
class AttackButtonsComponent{
}

fdescribe('DungionCrawlerComponent shallow', () => {
  let component: DungionCrawlerComponent;
  let fixture: ComponentFixture<DungionCrawlerComponent>;
```

11. Let's run our tests again. You'll notice they still fail. The last step we need to do, is to include them in our "declarations" since they are component's that we're using. Make sure they're in "declarations"; "imports" are used for modules and "providers" are used for services.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ DungionCrawlerComponent, EnemyUIComponent, AttackButtonsComponent]
  })
  .compileComponents();
}));
```

DUNGIONCRAWLER COMPONENT TDD

1. The first thing we need to do is to setup our "ngOnInit" method. If we remember our wireframe, every section of that is covered by other portions of this guide except the player dice rolls and the players image. With that in mind we can start there. For our "ngOnInit" method we're going to need to get the players selected class from "sessionStorage", and with that information set the image based on that. This component is also in charge of calculating damage, keeping track of the health bars and determining win conditions. So we need to set this all up from our "ngOnInit" method since they're mostly global variables.

2. Let's start by implementing the most basic case, let's create a test and see if a global variable called "currClass" is set to the players class that they selected which was saved in our "sessionStorage".

```
it('should grab the current class selected from sessionStorage on startup', () => {
  sessionStorage.setItem("playersClass", "Ranger");
  component.ngOnInit();
  expect(component.currClass).toEqual("Ranger");
});
```

To make this "red" compiler error turn "green", let's create the global variable.

```
currClass: string;
```

3. Now let's run our newly added test. Watch the test fail, now implement the code and run the tests again to watch them pass.

```
ngOnInit() {
  this.currClass = sessionStorage.getItem("playersClass");
}
```

DUNGIONCRAWLER COMPONENT TDD

4. Next, let's take another look at our HTML since we have the players class.

```
<img id="playersClass" [src]="currentImage"/>
```

The image "src" is being property binded to a global variable "currentImage". This variable is going to be a string that is actually a file path to the correct image which is based on the class that we just got with our global variable "currClass".

5. Let's create another test now with a new global variable "currentImage" and see if it gets set to the correct image file path.

```
it('should set the image of the player to the class selected if Fighter', () => {
  sessionStorage.setItem("playersClass", "Fighter");
  component.ngOnInit();
  expect(component.currentImage).toEqual("assets/images/Fighter.png");
});
```

Once again, let's fix this compiler error by adding the global variable.

```
currentImage: string = "";
currClass: string;
```

Now watch the test fail so we can implement the code to make it pass.

6. To make this test pass, let's implement a switch statement.

```
ngOnInit() {
  this.currClass = sessionStorage.getItem("playersClass");
  switch(this.currClass) {
    case "Fighter":
      this.currentImage = "assets/images/Fighter.png";
      break;
```

Now run the tests again and watch them pass.

DUNGIONCRAWLER COMPONENT TDD

7. Our next step is to check to see if the player's class was a "Ranger" or a "Black Mage", so we'll need 2 more tests but let's start with "Ranger".

```
it('should set the image of the player to the class selected if Ranger', () => {
  sessionStorage.setItem("playersClass", "Ranger");
  component.ngOnInit();
  expect(component.currentImage).toEqual("assets/images/Ranger.png");
});
```

Now run our newly added test, watch it fail. Now let's implement just enough code to make it pass, which is simply adding another case to our switch statement.

```
ngOnInit() {
  this.currClass = sessionStorage.getItem("playersClass");
  switch(this.currClass) {
    case "Fighter":
      this.currentImage = "assets/images/Fighter.png";
      break;
    case "Ranger":
      this.currentImage = "assets/images/Ranger.png";
      break;
  }
```

8. Once again, let's go ahead and add 1 more test to check if they selected "Black Mage".

```
it('should set the image of the player to the class selected if Black Mage', () => {
  sessionStorage.setItem("playersClass", "BlackMage");
  component.ngOnInit();
  expect(component.currentImage).toEqual("assets/images/BlackMage.png");
});
```

And watch it fail.

DUNGIONCRAWLER COMPONENT TDD

9. If we now add our last case in our switch statement,

```
ngOnInit() {
  this.currClass = sessionStorage.getItem("playersClass");
  switch(this.currClass) {
    case "Fighter":
      this.currentImage = "assets/images/Fighter.png";
      break;
    case "Ranger":
      this.currentImage = "assets/images/Ranger.png";
      break;
    case "BlackMage":
      this.currentImage = "assets/images/BlackMage.png";
      break;
  }
```

We should see all the tests pass. Our final testing for this switch should look like,

```
it('should set the image of the player to the class selected if Fighter', () => {
  sessionStorage.setItem("playersClass", "Fighter");
  component.ngOnInit();
  expect(component.currentImage).toEqual("assets/images/Fighter.png");
});

it('should set the image of the player to the class selected if Ranger', () => {
  sessionStorage.setItem("playersClass", "Ranger");
  component.ngOnInit();
  expect(component.currentImage).toEqual("assets/images/Ranger.png");
});

it('should set the image of the player to the class selected if Black Mage', () => {
  sessionStorage.setItem("playersClass", "BlackMage");
  component.ngOnInit();
  expect(component.currentImage).toEqual("assets/images/BlackMage.png");
});
```

DUNGIONCRAWLER COMPONENT TDD

10. Now that we have our players image being displayed properly based on the class they selected, we need to setup the initial starting point for the rest of our game. For this we'll need a counter for the opponent's health, a counter for the player's health, a counter for the player's score, and lastly a enemy counter so we can see which enemy we're on so we can check for ending the game conditions. For instance, "Easy" difficulty as 2 enemies, "Medium" has 3, and "Hard" has 4. So for this setup let's create a nested describe inside of our unit testing "spec.ts" file.

```
describe("setup", () => {
});
```

Next we can create a test to check for these simple setup variables.

```
describe("setup", () => {
  it('should setup enemyHP, playerHP, playerScore, mobCounter', () => {
    component.enemyHealth = 0;
    component.playerHealth = 0;
    component.playerScore = 10000;
    component.mobCounter = 10000;
    component.ngOnInit();
    expect(component.enemyHealth).toEqual(10);
    expect(component.playerHealth).toEqual(10);
    expect(component.playerScore).toEqual(0);
    expect(component.mobCounter).toEqual(1);
  });
```

We begin by setting our variables to bad values, then check to see if they change to the initial values after call our our "ngOnInit" method. If we remember our wireframe, that's how we get the initial values for the two health fields. Furthermore, it's obvious that the playerScore should start at 0, and that we begin on the first enemy, so it's 1.

DUNGIONCRAWLER COMPONENT TDD

11. If we wanted to, we could break this test up into 4 separate tests and have each one check for one variable being set, but since the logic is simple we'll keep it one test. To make this test compile, which is the same thing as going from a "red" test to "green", we just need to implement our global variables.

```
mobCounter: number;
enemyHealth: number;
playerHealth: number;
playerScore: number;
```

So now that our test is green, we can run it again. Watch it turn red, so now we can implement our actual code inside of "ngOnInit" to make this test pass.

```
ngOnInit() {
  this.currClass = sessionStorage.getItem("playersClass");
  switch(this.currClass) {
    case "Fighter":
      this.currentImage = "assets/images/Fighter.png";
      break;
    case "Ranger":
      this.currentImage = "assets/images/Ranger.png";
      break;
    case "BlackMage":
      this.currentImage = "assets/images/BlackMage.png";
      break;
  }
  this.enemyHealth = 10;
  this.playerHealth = 10;
  this.playerScore = 0;
  this.mobCounter = 1;
```

Notice, we need the "playerHealth" and "enemyHealth" to match exactly because of our HTML, since we're using interpolation to display the data.

```
<span id="playersHP">Your HP: {{playerHealth}}</span>
<span id ="enemyHP">Enemy Hp: {{enemyHealth}}</span>
```

DUNGIONCRAWLER COMPONENT TDD

12. Our last piece of logic we need inside of our "ngOnInit" method is more difficult. If we go back to our wireframe (Pg 132), this is taken when the game first launches so we need the enemy's attack type they plan on performing. We set the functionality up for this during the "enemyUI" section of this guide. As a quick reminder, we created the method "getEnemyAttackType" and we had a global variable of data type "Observable<void>" called "enemyAttackTypeEvent". In that section, we set up this observables "subscribe" method on "ngOnInit", so that way when we send some signal it would generate another attack type the enemy would perform. We are now going to setup a way to send that signal to generate the initial attack the enemy is attempting to perform.

13. First thing we need to do in our parent component "dungion-crawler.component.ts", is make sure that our child component "enemy-ui.component.ts" is fully loaded since our parent technically loads and launchs "ngOnInit" first when it comes to priority. The way we can implement this is by creating a "delay" method that will simply just wait for some amount of time before running it's body of code.

DUNGIONCRAWLER COMPONENT TDD

14. The way a delay method works, we take in some time as a paramater, then return a "promise". A promise is basically a way we can calculate some advance piece of code via "async" without holding up the rest of our code. Promise's are "async" which means they run in the background as the application continue's to run and return some value once that value has been calculated or is finished. In this case we're just going to use this promise as a way to wait some time before we execute some block of code.

15. The way we create a test for this method is by making it a "fakeAsync", so that we can fake the "async" portion of a promise. By making a test "fakeAsync" we can also use the method "tick" which will pass some time by before continuing.

```
it('should delay for 40 ms', fakeAsync(() => {
}));
```

The way we can continue this test is by setting some boolean value to "false", and checking to see if it changes to "true" after the timeout has been met.

```
it('should delay for 40 ms', fakeAsync(() => {
  let called: boolean = false;
  component.delay(40).then(() => {
    called = true;
  });
  expect(called).toBeFalsy();
  tick(20);
  expect(called).toBeFalsy();
  tick(20);
  expect(called).toBeTruthy();
}));
```

DUNGIONCRAWLER COMPONENT TDD

16. With this test, notice how we have a ".then" on the end of our "delay" method. Since our method needs to return a promise, promises have a "then" method automatically built in so we don't need to worry about mocking it. The way we make this test compile is by implementing an empty signature to our code.

```
delay(ms: number) {
}
```

Now notice, the compiler error has switched to our ".then" method because we're not returning anything. To make this next compiler error go from "red" to "green", let's go ahead and implement just enough code to our delay method.

```
delay(ms: number) {
  return new Promise( () => {});
}
```

Now our test will compile and run. If we run our test now, it should fail. This is because all we do is return a promise that has a ".then" method, except that fact our promise isn't actually doing anything. Now with our test failing, implement the rest of the code to make it pass.

```
delay(ms: number) {
  return new Promise( resolve => setTimeout( () => {
    resolve(ms);
  }, ms));
}
```

All this code is doing is setting a "timeout" equal to the parameter which in return will send back a promise once this timeout is reached.

DUNGIONCRAWLER COMPONENT TDD

17. With our delay method now setup, we can add it to our "ngOnInit" method. This way we can wait for some time to make sure our "enemy-ui.component" is fully loaded. Before we implement it, we need to mock this method since it's already been tested and we're only testing our "ngOnInit" method. The way we're going to do this is by mocking the "then" method of our promise. Basically we just create an empty "then" method that instantly returns the promise so we can access our delay's body of code.

```
let mockthen: any = {
  then: fn => fn()
}
```

Now since our "delay" method is inside of our same component we're testing, we don't need to inject it by creating an object with a spy. Instead, we're going to simply "spyOn" it and then return our "mockthen" which is the same as returning our promise.

```
spyOn(component, 'delay').and.returnValue(mockthen);
```

So inside of our nested describe "setup", let's add another test to check if our delay was called.

```
it('should delay for 40 then call .next on our Subject to get the push event from our child', () => {
  let mockthen: any = {
    then: fn => fn()
  }
  spyOn(component, 'delay').and.returnValue(mockthen);
  component.ngOnInit();
  expect(component.delay).toHaveBeenCalled();
});
```

DUNGIONCRAWLER COMPONENT TDD

18. With our new test, let's go ahead and run "ng test". You'll see it fails, so now we can add it to our "ngOnInit" method.

```
ngOnInit() {
  this.currClass = sessionStorage.getItem("playersClass");
  switch(this.currClass) {
    case "Fighter":
      this.currentImage = "assets/images/Fighter.png";
      break;
    case "Ranger":
      this.currentImage = "assets/images/Ranger.png";
      break;
    case "BlackMage":
      this.currentImage = "assets/images/BlackMage.png";
      break;
  }
  this.enemyHealth = 10;
  this.playerHealth = 10;
  this.playerScore = 0;
  this.mobCounter = 1;
  this.delay(40).then( () => {
  });
}
```

Now our test passes, so our next step is to refactor our test so we check for the logic inside of our body.

19. Our next step is to look at our HTML again.

```
[enemyAttackTypeEvent] = "triggerEnemyAttackType.asObservable()"
```

This is our signal to our child component.

DUNGIONCRAWLER COMPONENT TDD

20. That HTML snippet on line 5, is how we send our signal to our "enemyUI", which triggers it's subscribe to generate us another attack the enemy is going to do. The way we need to do this is by creating another global variable with the name "triggerEnemyAttackType" which is seen in our HTML. Notice that in our HTML we call the method ".asObservable()" on our variable. This method is part of the "Subject" class. In this case, subject's are perfect for what we're trying to accomplish, which is to send "next" signals when we want the enemy to generate another attack. By definition, "A Subject is a special type of Observable that allows values to be multicasted to many Observables. Subjects are like EventEmitters." If we remember, our "@Output" variables from our "enemyUI" are of the type "EventEmitter", so by using subject's in our parent component, we can send event triggers to our child's "@Input" variables to trigger the "subscribe" methods we setup inside of the TypeScript file.

21. With that being said we can now create our new global variable.

```
triggerEnemyAttackType: Subject<void> = new Subject<void>();
```

22. For demonstration purposes, if we look at our HTML on line 5, we have three different ".asObservable". One of them is to generate the next enemy to fight, one is to trigger the enemy rolling their dice, and the last one is what we just implemented. Let's create the other two here variables here so we can use them later in this section of the guide.

```
triggerEnemyDiceRolls: Subject<void> = new Subject<void>();
triggerEnemyAttackType: Subject<void> = new Subject<void>();
triggerNextMob: Subject<void> = new Subject<void>();
```

DUNGIONCRAWLER COMPONENT TDD

23. So now our next step is to implement this variable into our delay's body of code. So first thing we need to do is refactor our test. Subject's have a ".next" method which is a way to trigger our child's "subscribe" methods. But for testing purposes, we need to mock that method, inject that mock into our components variable, and then check to see if it was called.

```
it('should delay for 40 then call .next on our Subject to get the push event from our child', () => {
  const mockSubject: any = {
    next : jasmine.createSpy('next')
  }
  component.triggerEnemyAttackType = mockSubject;
  let mockthen: any = {
    then: fn => fn()
  }
  spyOn(component, 'delay').and.returnValue(mockthen);
  component.ngOnInit();
  expect(component.delay).toHaveBeenCalled();
  expect(component.triggerEnemyAttackType.next).toHaveBeenCalled();
});
```

We should run our test now and watch it fail.

24. Now we can implement the body to our "delay" method call to make this test pass.

```
this.delay(40).then( () => {
  this.triggerEnemyAttackType.next();
});
```

With this last test in place for our method "ngOnInit", we can move onto another method.

DUNGIONCRAWLER COMPONENT TDD

25. The next method we're going to implement is a way to roll dice for our player. If we look at our HTML again we can find the method name.

```
<app-attack-buttons (currentPlayerCommand)="getPlayersAttack($event)"></app-attack-buttons>
```

Reminder, all of the buttons that our user can interact with are in another component "attack-buttons". So when we click a button that actually get's sent to that other component, so with this injection it sends the button clicked back to our current method "getPlayersAttack" as a string. So with this method name we can create a test for it.

26. This method is going to need to reset our current dice rolls to an empty array, set our player's current attack type equal to the "$event" which is a string in this case, roll a set of new dice for our player, trigger the enemy to roll dice, and lastly to trigger the next attack our enemy is going to perform. We trigger our enemies next attack ahead of time so we can notify the player.

27. This next method is actually quite simple, first we need another global variable to manage our player's current dice rolls. The second global we'll need is our player's current attack. Let's create a test for these 2 globals.

```
it('should grab the attackType, reset the dice rolled, and emit them to the child', () => {
  component.playerDiceRolls.push("3");
  component.playersCurrentAttackType = "";
  component.getPlayersAttack('Attack');
  expect(component.playerDiceRolls).toEqual([]);
  expect(component.playersCurrentAttackType).toEqual("Attack");
});
```

Notice how we pass a string into our "$event" parameter. We also check to make sure the array is empty and that our players current attack is equal to our paramater. Now we can fix these compiler errors one at a time.

DUNGIONCRAWLER COMPONENT TDD

28. First let's implement our two new global variables.

```
playerDiceRolls : string[] = [];
playersCurrentAttackType : string;
```

Now let's implement a method signature for our test.

```
getPlayersAttack($event) {
}
```

Next, since our test is now fully compiling, let's go ahead and run it. Once again, we now need to implement the code to make it pass.

```
getPlayersAttack($event) {
  this.playerDiceRolls = [];
  this.playersCurrentAttackType = $event;
}
```

29. Since we have an array for our dice rolls and our attack saved our next step is to roll dice. However, for now we're just going to create an empty method "playerRoll" so we can "spyOn" it, we'll TDD that method when we implement it.

DUNGIONCRAWLER CHECKPOINT 2

1. Before we can write even an empty method "playerRoll" we need to refactor our current test to account for it.

2. The goal in this checkpoint is to refactor our test to "spyOn" this empty method and check to see if it's been called one time.

DUNGIONCRAWLER CHECKPOINT 2 SOLUTION

1. The first thing we need to do is create a "spyOn" for this method.

```
it('should grab the attackType, reset the dice rolled, and emit them to the child', () => {
  component.playerDiceRolls.push("3");
  component.playersCurrentAttackType = "";
  spyOn(component, 'playerRoll');
  component.getPlayersAttack('Attack');
  expect(component.playerDiceRolls).toEqual([]);
  expect(component.playersCurrentAttackType).toEqual("Attack");
  expect(component.playerRoll).toHaveBeenCalledTimes(1);
});
```

Then check to make sure it was called one time.

2. From here, we have to create an empty method so our compiler error goes from "red" to "green" which follows TDD.

```
playerRoll() {
}
```

3. Now we watch our test fail by running "ng test". Our next step is make this test pass which is by adding a call to our spy within the method we're testing.

```
getPlayersAttack($event) {
  this.playerDiceRolls = [];
  this.playersCurrentAttackType = $event;
  this.playerRoll();
}
```

Even though this method is currently empty inside our implementation, our test's don't actually care because we mock the method regardless.

DUNGIONCRAWLER COMPONENT TDD

30. The last step we need to implement here is to trigger both our enemy rolling dice and our enemy determining there next attack. If we follow our "ngOnInit" method with our Subject's ".next" method we can do the same thing here. Since we're just triggering two seperate actions inside of our child component "enemyUI" all we need to do is run there ".next" methods to send those events. Futhermore, since that component has been loaded we don't need a delay method call here.

31. First thing we need to do is refactor our test with 2 spies for both of these ".next" methods.

```
it('should grab the attackType, reset the dice rolled, and emit them to the child', () => {
  const mockEnemyAttackType: any = {
    next : jasmine.createSpy('next')
  }
  const mockEnemyDiceRolls: any = {
    next : jasmine.createSpy('next')
  }
```

Then we need to inject these spies into our component's global variables.

```
component.triggerEnemyDiceRolls = mockEnemyDiceRolls;
component.triggerEnemyAttackType = mockEnemyAttackType;
```

Lastly, all we need to do is make sure that the spies are called.

```
expect(component.triggerEnemyDiceRolls.next).toHaveBeenCalledTimes(1);
expect(component.triggerEnemyAttackType.next).toHaveBeenCalledTimes(1);
```

DUNGIONCRAWLER COMPONENT TDD

32. Now our test is finished and looks like,

```
it('should grab the attackType, reset the dice rolled, and emit them to the child', () => {
  const mockEnemyAttackType: any = {
    next : jasmine.createSpy('next')
  }
  const mockEnemyDiceRolls: any = {
    next : jasmine.createSpy('next')
  }
  component.playerDiceRolls.push("3");
  component.playersCurrentAttackType = "";
  component.triggerEnemyDiceRolls = mockEnemyDiceRolls;
  component.triggerEnemyAttackType = mockEnemyAttackType;
  spyOn(component, 'playerRoll');
  component.getPlayersAttack('Attack');
  expect(component.playerDiceRolls).toEqual([]);
  expect(component.playersCurrentAttackType).toEqual("Attack");
  expect(component.triggerEnemyDiceRolls.next).toHaveBeenCalledTimes(1);
  expect(component.triggerEnemyAttackType.next).toHaveBeenCalledTimes(1);
  expect(component.playerRoll).toHaveBeenCalledTimes(1);
});
```

So let's run it. Watch it fail, so we can implement the rest of our method to make it pass.

```
getPlayersAttack($event) {
  this.playerDiceRolls = [];
  this.playersCurrentAttackType = $event;
  this.playerRoll();
  this.triggerEnemyDiceRolls.next();
  this.triggerEnemyAttackType.next();
}
```

DUNGIONCRAWLER COMPONENT TDD

33. Now that we have the a way to get the players attack and there dice rolls, let's implement a method to get the enemies attack type and display it on our HTML page. For this method we can look at our HTML on line 18.

```
<p><i>{{enemyAttackTypeOnHtml}}</i></p>
```

We are directly displaying the enemies attack in the HTML while using interpolation which means we need another global variable. Furthermore, on line 5 in our HTML we also have,

```
(attackType)="getEnemyAttack($event)"
```

Which means when we trigger the enemy to decide there next move from our method "getPlayersAttack", they send that move back to us through it's variable "@Output() attackType". It's going to emit that event back to our method "getEnemyAttack" as a string in the parameter. So we also have our new method name.

34. With all that being stated, we need to set a global variable "enemyAttackTypeOnHtml" to some string so it displays on the HTML page. We're also going to need to save the enemies attack type for later on when we calculate damage and points awarded or deducted.

DUNGIONCRAWLER COMPONENT TDD

35. So our first step is to create a test for this method with these two new global variables.

```
it('should set the enemyAttackTypeOnHtml, and the enemies current attack type TS', () => {
  component.enemyAttackTypeOnHtml = "";
  component.enemyCurrentAttackType = "";
  component.getEnemyAttack("Attack");
  const fakeString: string = "Enemy plans on Attacking\nSelect your move";
  expect(component.enemyAttackTypeOnHtml).toEqual(fakeString);
  expect(component.enemyCurrentAttackType).toEqual("Attack");
});
```

Once again, we can pass a string into the parameter even though it's a "$event". So following the same steps we've been doing, let's go ahead and implement an empty method signature and two global variables just to make this test compile. As a reminder, since we're following true TDD, we don't write any implementation code without writing the tests first which is why we need these compile errors, this is the same as going from "red" to "green" initially. Also as a note, we can create a fakeString and check to see if our string's are the same after our method call.

36. So implementing our two global variables first.

```
enemyCurrentAttackType: string;
enemyAttackTypeOnHtml: string;
```

Now we just implement an empty method.

```
getEnemyAttack($event) {
}
```

DUNGIONCRAWLER COMPONENT TDD

37. With these new global variables and an empty method, run our test. It will fail, so now let's implement the code to make it pass.

```
getEnemyAttack($event) {
  this.enemyAttackTypeOnHtml = "Enemy plans on " + $event + "ing\nSelect your move";
  this.enemyCurrentAttackType = $event;
}
```

38. Now that we have the enemy's current attack, let's create a method to get the enemy's dice rolls. As a reminder, our "enemyUI" component actually rolls the dice for our opponent, all we have to do here is get it via our HTML code again. The way we get this is the same way we just did it; take a look at our HTML on line 5.

```
(enemyDiceRolls)="getEnemyDiceRolls($event)"
```

So we just need to create a method "getEnemyDiceRolls" to take in a "$event". In this situation the "$event" is actually a string array since it corresponds to a list of dice rolls.

39. This new method needs to only do two things, it needs to set yet another global variable "enemyDiceRolls" to the "$event". We'll need these dice rolls to calculate damage. Next, we need to call another method to calculate the damage now that we have both our player's dice rolls and our enemies dice rolls.

DUNGIONCRAWLER COMPONENT TDD

40. As we've just done previously, we can create another empty method "calculateDamage" and just "spyOn" it for now so we can continue TDD on our current method "getEnemyDiceRolls($event)". So with this in mind, let's go ahead and create our next test.

```
it('should get the enemies dice, and call calculateDamage', () => {
  spyOn(component, 'calculateDamage');
  component.enemyDiceRolls = [];
  const fakeArray: string[] = ["Attack", "Defend", "Flee"];
  component.getEnemyDiceRolls(fakeArray);
  expect(component.enemyDiceRolls).toBe(fakeArray);
  expect(component.calculateDamage).toHaveBeenCalledTimes(1);
});
```

Following our previous tests, we mock our "calculateDamage" method, set our "enemyDiceRolls" array to empty to begin with, and then create a fake string array for fake dice rolls that we pass to our method we're testing. After our method is called, we just make sure our global was set and that our spy was called once.

41. First thing's first, we need to create another global variable,

```
enemyDiceRolls : string[] = [];
```

and now a blank method signature.

```
getEnemyDiceRolls($event) {
}
```

DUNGIONCRAWLER COMPONENT TDD

42. Now that our compiler error's are now "green", let's run our test. Notice how it fails, then implement the code to make it pass.

```
getEnemyDiceRolls($event) {
  this.enemyDiceRolls = $event;
  this.calculateDamage();
}
```

43. The next method we're going to implement already has it's signature created from when we did TDD on "getPlayersAttack" which is our method "playerRoll". For this method we need to bring in our "diceService".

44. To bring this service in, we're going to need to inject it into our constructor. Luckily for us, since this method isn't being called from our "ngOnInit" method, we don't actually need to include this service into our "shallow.spec.ts" file. This is because we only create the component inside of our "shallow.spec.ts" file which only runs our "ngOnInit" method. So with this ideology, anything we inject into our constructor is purely unit testing since we don't use any of it inside of our "ngOnInit" method, and hence we only need to include it into our "spec.ts" file.

DUNGIONCRAWLER COMPONENT TDD

45. So since we need to inject it into our constructor, let's mock it first inside of our unit testing "spec.ts". Because we want to demonstrate multiple ways to implement mocks, we're going to create our own class "DiceService" with our own methods so we can "spyOn" them rather than having to create an object with spies.

```
class DiceService {

  constructor(){}

  roll() {}

  addDice(response: number, attackType: string) {}
}
```

Notice, we don't need actually need return types at all. Later on we can just simply "spyOn" and have it return whatever we want directly from there.

46. Our next step is to now inject it into our real component we're testing. First we need to create a variable of type "any". `let mockDiceService: any;`
Then we need to initialize this variable to our new service we just made.

```
mockDiceService = new DiceService();
```

Once we have it declared and initialized, we just need to inject it into the constructor.

```
mockDiceService = new DiceService();
component = new DungionCrawlerComponent(mockDiceService);
```

DUNGIONCRAWLER COMPONENT TDD

47. Now to make this compiler error go from "red" to "green" we can now inject it into our constructor.

```
constructor(private _diceService: DiceService) { }
```

48. Our next step is to implement our "playerRoll" method. For learning purposes, we're going to go ahead and use a "forkJoin" to accomplish this task. The way "forkJoin" works, it runs multiple Observable return methods and waits for all of them to come back with a response before running it's subscribe. For example, we typically run "diceService.roll().subscribe(() => {})", which will wait for a single response from our roll method. Here we can call our roll method multiple times, and subscribe to all of them at once.

49. Since we need to roll six dice, we can begin writing our test.

```
it('should call playerRoll 6 times using a forkjoin', fakeAsync(() => {
  spyOn(mockDiceService, 'roll');
  component.playerRoll();
  tick(100);
  expect(mockDiceService.roll).toHaveBeenCalledTimes(6);
}));
```

We begin by making the test "fakeAsync". Since we have to wait for six calls to our "roll" method to return some value. We do this because we're trying to imitate the actual behavior. If we made six calls to a Java backend, it could take awhile. This is also why after we call our method, we wait 100ms to make sure it fully processed the six rolls.

DUNGIONCRAWLER COMPONENT TDD

50. The next thing we do in the test is just make sure our roll method that we injected has a "spyOn" so that way we just mock the behavior, currently it's only being used to see if it's called or not. Then we check to see if it was called six times. If we run the test now, it fails. So if we implement just enough code for this test to pass we end up with.

```
playerRoll() {
  this._diceService.roll();
  this._diceService.roll();
  this._diceService.roll();
  this._diceService.roll();
  this._diceService.roll();
  this._diceService.roll();
}
```

Now our test will pass, but we're not actually doing anything with the responses, so let's refactor our test. The first thing we need to refactor is changing our "spyOn" to return values.

```
it('should call playerRoll 6 times using a forkjoin', fakeAsync(() => {
  spyOn(mockDiceService, 'roll').and.returnValues(of(1), of(4), of(2), of(5), of(3), of(6));
  component.playerRoll();
  tick(100);
  expect(mockDiceService.roll).toHaveBeenCalledTimes(6);
}));
```

Since we need to "subscribe" to these roll values, they need to be "of()" which just returns an Observable with a body of a number in this case. Notice how the test will continue to pass though.

DUNGIONCRAWLER COMPONENT TDD

51. Since the test is still passing, we can continue to refactor our test. Our next step is to actually add these rolls to our global variable "component.playerDiceRolls". Since this is a string array we also need to use our mocked services method "addDice" from our "DiceService" inside of our "spec.ts" file. The actual method "addDice" take's in a number and the current attack type to determine what a roll will correspond with. For example, if we roll a 1, 2, or 3 when we're trying to do an "Attack", it succeeds, otherwise if we roll a 4, 5, or 6 this die results in "Nothing". We have to do this logic for all 6 dice on an "Attack".

52. For this test we're going to fake an "Attack". If you remember our return values from the roll "spyOn",

```
it('should call playerRoll 6 times using a forkjoin', fakeAsync(() => {
  spyOn(mockDiceService, 'roll').and.returnValues(of(1), of(4), of(2), of(5), of(3), of(6));
  component.playerRoll();
  tick(100);
  expect(mockDiceService.roll).toHaveBeenCalledTimes(6);
}));
```

We are returning a dice roll of 1, then 4, then 2, then 5, then 3, then 6. So with these values and an "Attack" as the command, our "addDice" should return a "Attack", "Nothing", "Attack", "Nothing", "Attack", "Nothing". So we can refactor our test to include this logic.

```
it('should call playerRoll 6 times using a forkjoin', fakeAsync(() => {
  spyOn(mockDiceService, 'roll').and.returnValues(of(1), of(4), of(2), of(5), of(3), of(6));
  spyOn(mockDiceService, 'addDice').and.returnValues("Attack", "Nothing", "Attack", "Nothing", "Attack", "Nothing");
  component.playersCurrentAttackType = "Attack";
  component.playerRoll();
  tick(100);
  expect(mockDiceService.roll).toHaveBeenCalledTimes(6);
}));
```

DUNGIONCRAWLER COMPONENT TDD

53. Now that we have our new logic tested, our test still passes. So now we can add another "expect" to make our test fail so we can include our "addDice" method into our implementation.

```
it('should call playerRoll 6 times using a forkjoin', fakeAsync(() => {
  spyOn(mockDiceService, 'roll').and.returnValues(of(1), of(4), of(2), of(5), of(3), of(6));
  spyOn(mockDiceService, 'addDice').and.returnValues("Attack", "Nothing", "Attack", "Nothing", "Attack", "Nothing");
  component.playersCurrentAttackType = "Attack";
  component.playerRoll();
  tick(100);
  expect(mockDiceService.roll).toHaveBeenCalledTimes(6);
  expect(mockDiceService.addDice).toHaveBeenCalledTimes(6);
}));
```

Now watch our test fail since we're not calling "addDice". Now let's go ahead and implement the code to make this test pass.

```
playerRoll() {
  forkJoin(
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll()
  ).subscribe( ([dice1, dice2, dice3, dice4, dice5, dice6]) => {
    this._diceService.addDice(dice1, this.playersCurrentAttackType);
    this._diceService.addDice(dice2, this.playersCurrentAttackType);
    this._diceService.addDice(dice3, this.playersCurrentAttackType);
    this._diceService.addDice(dice4, this.playersCurrentAttackType);
    this._diceService.addDice(dice5, this.playersCurrentAttackType);
    this._diceService.addDice(dice6, this.playersCurrentAttackType);
  })
}
```

The way a "forkJoin" works, the "subscribe" will wait for all six calls, and then put the response into an array in the same order they where called. So now we can just call our "addDice" method on these dice rolls with our "playersCurrentAttackType" set to "Attack".

DUNGIONCRAWLER COMPONENT TDD

54. Our last test refactor is to actually store these dice rolls returned from our "addDice" method into our global variable "playerDiceRolls". The way we can test for this is by setting our global to an empty array, creating a fake string array inside our test, then checking to see if they're equal after we call our method.

```
it('should call playerRoll 6 times using a forkjoin', fakeAsync(() => {
  spyOn(mockDiceService, 'roll').and.returnValues(of(1), of(4), of(2), of(5), of(3), of(6));
  spyOn(mockDiceService, 'addDice').and.returnValues("Attack", "Nothing", "Attack", "Nothing", "Attack", "Nothing");
  const mockAddDice: string[] = ["Attack", "Nothing", "Attack", "Nothing", "Attack", "Nothing"];
  component.playerDiceRolls = [];
  component.playersCurrentAttackType = "Attack";
  component.playerRoll();
  tick(100);
  expect(mockDiceService.roll).toHaveBeenCalledTimes(6);
  expect(mockDiceService.addDice).toHaveBeenCalledTimes(6);
  expect(component.playerDiceRolls).toEqual(mockAddDice);
}));
```

Next step, run our test watch it fail.

55. Lastly, for this method let's make our last refactor pass. The only implementation we're missing is pushing these dice rolls returned from "addDice" into our array. So let's implement it and watch our test pass.

```
playerRoll() {
  forkJoin(
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll(),
    this._diceService.roll()
  ).subscribe( ([dice1, dice2, dice3, dice4, dice5, dice6]) => {
    this.playerDiceRolls.push(this._diceService.addDice(dice1, this.playersCurrentAttackType));
    this.playerDiceRolls.push(this._diceService.addDice(dice2, this.playersCurrentAttackType));
    this.playerDiceRolls.push(this._diceService.addDice(dice3, this.playersCurrentAttackType));
    this.playerDiceRolls.push(this._diceService.addDice(dice4, this.playersCurrentAttackType));
    this.playerDiceRolls.push(this._diceService.addDice(dice5, this.playersCurrentAttackType));
    this.playerDiceRolls.push(this._diceService.addDice(dice6, this.playersCurrentAttackType));
  })
}
```

DUNGIONCRAWLER COMPONENT TDD

56. The next method we're going to implement is our "Mat Dialog" box. As a reminder, we created a "Mat Dialog" component (Pg 93-103) that would display information through the field "MAT_DIALOG_DATA", we are now going to open this dialog box and pass it the data.

57. For us to open this dialog box, we need to inject a "MatDialog" variable into our constructor. Since "MatDialog" uses HTML, we just need to add it into our "shallow.spec.ts" file in our imports as a "MatDialogModule". Just so when TestBed creates our CSS, HTML, and TypeScript it can find it.

```
fdescribe('DungionCrawlerComponent shallow', () => {
  let component: DungionCrawlerComponent;
  let fixture: ComponentFixture<DungionCrawlerComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ DungionCrawlerComponent, EnemyUIComponent, AttackButtonsComponent ],
      imports: [MatDialogModule]
    })
    .compileComponents();
  }));
```

If we forget about this step, our test will fail and notify us that it cannot find "Mat Dialog".

58. Now we need to create a fake mockDialog inside of our unit tests. "MatDialog" has an "open" method we need, so our mock is going to include a spy for this open method. However, once we "open" this "MatDialog", it returns a "MatDialogRef" which we're also going to need.

DUNGIONCRAWLER COMPONENT TDD

59. The reason we need this "MatDialogRef", is because we can call it's "afterClosed" method which only runs once we have closed our dialog box. Furthermore, since it waits for the dialog box to close before it runs, it returns an Observable that we need to subscribe to. So we need to mock this method as well by creating another variable "mockDialogRef" and having a spy mock the behavior of "afterClosed". Continuing down the chain of events, since our "afterClosed" method is mocked, it isn't an Observable in our tests, so we can create another mockVariable for our "subscribe" function, and just return this variable when "afterClosed" is called. The code for this looks like the following.

```
fdescribe('DungionCrawlerComponent', () => {
  let component: DungionCrawlerComponent;
  let mockDiceService: any;
  let mockDialog: any;
  let mockDialogRef: any;

  beforeEach(() => {
    let mockSubscribe: any = {
      subscribe: fn => fn()
    }
    mockDialogRef = {
      afterClosed: jasmine.createSpy('afterClosed').and.returnValue(mockSubscribe)
    }
    mockDialog = {
      open: jasmine.createSpy('open').and.returnValue(mockDialogRef)
    }
    mockDiceService = new DiceService();
    component = new DungionCrawlerComponent(mockDiceService, mockDialog);
  });
```

Notice the chain we built. Our dialog returns our dialogRef, and our dialogRef returns our custom built "subscribe" function. We have to declare our mockDialog and mockDialogRef outside the "beforeEach" so we can use them in our tests directly. With this all setup, let's inject our "MatDialog".

```
constructor(private _diceService: DiceService, private _dialog: MatDialog) { }
```

DUNGIONCRAWLER COMPONENT TDD

60. So now, let's go ahead and create a test for opening this dialog box. This method need's to take in some string so we can directly set our dialog boxes data equal to it. We're doing it this way because our data is directly displayed inside of our dialog box so we can setup our data as a single string for displaying. This data also needs to have the player's score appended to our string. Lastly, we're also going to set the "width" of our dialog box to "600px", which automatically get's converted to HTML.

61. For our test, we can begin by setting our global variable "playerScore" to some value, and then calling our method.

```
it('should open our injected dialog, and once closed, saves player score and navigates to scoreboard', () => {
  component.playerScore = 1290;
  component.openEndingDialog("You're awesome");
});
```

From here, we can implement an empty method to go from "red" to "green".

```
openEndingDialog(endingText: string) {
}
```

Now, let's add more to our test to actually test for some functionality.

```
it('should open our injected dialog, and once closed, saves player score and navigates to scoreboard', () => {
  component.playerScore = 1290;
  component.openEndingDialog("You're awesome");
  expect(mockDialog.open).toHaveBeenCalledTimes(1);
  expect(mockDialog.open).toHaveBeenCalledWith(EndingDialogComponent, {
    width: '600px',
    data: "You're awesome" + component.playerScore
  });
});
```

DUNGIONCRAWLER COMPONENT TDD

62. What we're checking for here is to see if our injected "dialog.open" spy is only being called one time. Then we make sure that once it's called, it's being called with our "MatDialog" component "EndingDialogComponent", and with an object. This object we're making sure has 2 fields, width and data. Furthermore, we're checking to make sure the field width is being set to '600px' and our field data is being set to our method's parameter "You're awesome", which is a string; then we append on our "component.playerScore".

63. If we run our test now that it actually tests some logic, we're notice it fails. So let's implement just enough code to make it pass.

```
openEndingDialog(endingText: string) {
  this._dialog.open(EndingDialogComponent, {
    width: '600px',
    data: endingText + this.playerScore
  });
}
```

Now that our test is passing again, let's refactor our test to include our "mockDialogRef.afterClosed()" spy.

```
it('should open our injected dialog, and once closed, saves player score and navigates to scoreboard', () => {
  component.playerScore = 1290;
  component.openEndingDialog("You're awesome");
  expect(mockDialog.open).toHaveBeenCalledTimes(1);
  expect(mockDialog.open).toHaveBeenCalledWith(EndingDialogComponent, {
    width: '600px',
    data: "You're awesome" + component.playerScore
  });
  expect(mockDialogRef.afterClosed).toHaveBeenCalledTimes(1);
});
```

Run our test again to watch it fail.

DUNGIONCRAWLER COMPONENT TDD

64. Now let's turn our test green again by implementing the following code.

```
openEndingDialog(endingText: string) {
  let dialogRef = this._dialog.open(EndingDialogComponent, {
    width: '600px',
    data: endingText + this.playerScore
  });
  dialogRef.afterClosed();
}
```

65. Our next step is to implement the body for our "afterClosed.subscribe()" method. What we're going to do here is inject a router into our constructor, and use this router. Basically what we're doing is once we have closed our dialog box, we want to route the user to our scoreboard component since the game is over. We're also going to want to save the player's score in "sessionStorage" so we can retrieve it inside of our "score-board.component.ts". This file was setup in a separate part of this guide (Pg 104-130).

66. Since our next step involves injecting into our constructor, let's go ahead and make sure our "shallow.spec.ts" file can find the router, otherwise it will break.

```
fdescribe('DungionCrawlerComponent shallow', () => {
  let component: DungionCrawlerComponent;
  let fixture: ComponentFixture<DungionCrawlerComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ DungionCrawlerComponent, EnemyUIComponent, AttackButtonsComponent ],
      imports: [RouterTestingModule, MatDialogModule]
    })
    .compileComponents();
  }));
```

By adding "RouterTestingModule" to our imports, it can find our router and it automatically will use a test module, which means a mocked router.

DUNGIONCRAWLER COMPONENT TDD

67. Now we need to mock a router for our unit testing "spec.ts". To demonstrate another way we can declare mocks, we're going to create a spy object this time as opposed to a regular spy. First thing we need to do is declare our variable and set it's type to "any".

```
fdescribe('DungionCrawlerComponent', () => {
  let component: DungionCrawlerComponent;
  let mockRouter: any;
  let mockDiceService: any;
  let mockDialog: any;
  let mockDialogRef: any;
```

Next we're going to initialize it.

```
beforeEach(() => {
  mockRouter = jasmine.createSpyObj(['navigate']);
  let mockSubscribe: any = {
    subscribe: fn => fn()
  }
  mockDialogRef = {
    afterClosed: jasmine.createSpy('afterClosed').and.returnValue(mockSubscribe)
  }
  mockDialog = {
    open: jasmine.createSpy('open').and.returnValue(mockDialogRef)
  }
  mockDiceService = new DiceService();
  component = new DungionCrawlerComponent(mockDiceService, mockDialog, mockRouter);
});
```

By doing it this way, we can actually specify a single spy that has multiple methods, as opposed to the other way which has multiple spies and each spy only manages one method. In this case we only need the 'navigate' method of our router to be mocked, but we could add more if we needed to by putting more method names into our "createSpyObj" array.

DUNGIONCRAWLER COMPONENT TDD

68. With our router mocked and injected, let's go ahead and implement a router into our actual constructor.

```
constructor(private _diceService: DiceService, private _router: Router, private _dialog: MatDialog) { }
```

So now we can return to our current test.

69. In our current test, we first need to add a test for our "sessionStorage", to make sure the player's score is being saved.

```
it('should open our injected dialog, and once closed, saves player score and navigates to scoreboard', () => {
  component.playerScore = 1290;
  component.openEndingDialog("You're awesome");
  expect(mockDialog.open).toHaveBeenCalledTimes(1);
  expect(mockDialog.open).toHaveBeenCalledWith(EndingDialogComponent, {
    width: '600px',
    data: "You're awesome" + component.playerScore
  });
  expect(mockDialogRef.afterClosed).toHaveBeenCalledTimes(1);
  expect(sessionStorage.getItem('score')).toEqual("1290");
});
```

All we do here is manually set it into the global variable, then grab it from "sessionStorage" to make sure they are equal. Run our test again to make sure it fails. Now let's implement our "sessionStorage" to make it pass.

```
openEndingDialog(endingText: string) {
  let dialogRef = this._dialog.open(EndingDialogComponent, {
    width: '600px',
    data: endingText + this.playerScore
  });
  dialogRef.afterClosed().subscribe( () => {
    sessionStorage.setItem('score', this.playerScore.toString());
  });
}
```

DUNGIONCRAWLER CHECKPOINT 3

1. As a quick note, we don't care about what this "subscribe" returns, so we leave the response empty and just run the body. Next, we need to refactor our test one more time to check to make sure our router is called once.

2. The goal for this checkpoint is to refactor our test one last time to check to make sure our "router.navigate" is being called one time. Once our test is failing again, the next step is to implement the code to make it pass.

DUNGIONCRAWLER CHECKPOINT 3 SOLUTION

1. The solution for our test should look like this.

```
it('should open our injected dialog, and once closed, saves player score and navigates to scoreboard', () => {
  component.playerScore = 1290;
  component.openEndingDialog("You're awesome");
  expect(mockDialog.open).toHaveBeenCalledTimes(1);
  expect(mockDialog.open).toHaveBeenCalledWith(EndingDialogComponent, {
    width: '600px',
    data: "You're awesome" + component.playerScore
  });
  expect(mockDialogRef.afterClosed).toHaveBeenCalledTimes(1);
  expect(sessionStorage.getItem('score')).toEqual("1290");
  expect(mockRouter.navigate).toHaveBeenCalledTimes(1);
});
```

We only need to add one more expect "expect(mockRouter.navigate).toHaveBeenCalledTimes(1);".

2. Once our test fails, we then implement the code to make it pass, which looks like the following.

```
openEndingDialog(endingText: string) {
  let dialogRef = this._dialog.open(EndingDialogComponent, {
    width: '600px',
    data: endingText + this.playerScore
  });
  dialogRef.afterClosed().subscribe( () => {
    sessionStorage.setItem('score', this.playerScore.toString());
    this._router.navigate(['/scoreboard']);
  });
}
```

Once again, we only needed to add one more line. Which is navigating to our "score-board.component.ts". Now our test passes.

DUNGIONCRAWLER CHECKPOINT 3 SOLUTION

```
openEndingDialog(endingText: string) {
  let dialogRef = this._dialog.open(EndingDialogComponent, {
    width: '600px',
    data: endingText + this.playerScore
  });
  dialogRef.afterClosed().subscribe( () => {
    sessionStorage.setItem('score', this.playerScore.toString());
    this._router.navigate(['/scoreboard']);
  });
}
```

3. So the real "navigate" method takes an array, and in this case it's looking for the path defined in our "app-routing.module.ts" file.

```
{
  path: 'scoreboard',
  component: ScoreBoardComponent,
  pathMatch: 'full'
},
```

This is the component from that "module.ts" file. The way we route to this component then is by putting a "/" in front of the path defined, which is how we got "/scoreboard".

DUNGIONCRAWLER COMPONENT TDD

70. Now with our "openEndingDialog" method finished, the next method we're implement will be a "calculateDiceRolls" method. This method will add up all dice the player and enemy has so we can calculate damage and check for the end of the game. For example, we're going to iterate through our global dice arrays and add up how many of each dice was rolled. Furthermore, if the player chooses to "Attack", they can only roll "Attack" or "Nothing" sided dice, which is a 50/50 on which you'll get, so if we end up with 4 dice that are "Attack" we'll add those up and compare those to how many "Defense" dice the opponent rolled.

71. The implementation is going to have if statement's since we need to check the dice for all possibilities. However, we're able to check all of these "if" statements in a single test since this method is going to return a single string that has all the dice added up seperated by comma's so we can split the string later on.

72. With this string, our first number will correspond to how many "Attack" dice the player rolled. The second number is the players "Shield" (aka Defense). The third number will be how many "Counter" dice our player rolled and the fourth will correspond to how many "Flee" dice our player rolled. The fifth number will be our enemies "Attack" dice; the sixth number is the enemies "Shield" dice, and lastly, the seventh number is how many "Counter" dice our enemy rolled. With this logic, we can now write a test for this.

DUNGIONCRAWLER COMPONENT TDD

73. Our string is going to look similar to, `"2,2,1,1,2,2,2";` With this string our player has rolled 2 Attack dice, 2 Shield dice, 1 Counter die, and 1 Flee die; while our enemy has rolled 2 Attack dice, 2 Shield dice, and 2 Counter dice. So now let's create a test that will slowly add onto this string. The way pure TDD is done, we start most basic and work our way up.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "1,0,0,0,0,0,0";
  expect(result).toEqual(mockedString);
});
```

With this being our initial test, the first thing we need to do is implement an empty method with a return type of string.

```
calculateDiceRolls(): string {
  return "";
}
```

74. Now with our test compiling, we can run it. The test will now fail, which means we can implement the actual code to our method.

DUNGIONCRAWLER COMPONENT TDD

75. To make our initial test pass, our implementation need's to return the same string format, so let's implement that.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

The string at this point would be "0,0,0,0,0,0,0" since all the values are 0. Because of this, the test still fails but now we're at least checking the same string format.

76. Next we need to implement an "if" statement to check for "Attack" so we can pass our test.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  for(let i = 0; i < 6; i++){
    if (this.playerDiceRolls[i]==="Attack"){
      playersAttack += 1;
    }
  }
  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

Now we'll see the test pass.

DUNGIONCRAWLER COMPONENT TDD

77. Now we're going to refactor our test to check for a player "Shield" roll.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "1,1,0,0,0,0,0";
  expect(result).toEqual(mockedString);
});
```

Run the test, watch it fail, now implement the next "if" statement.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  for(let i = 0; i < 6; i++){
    if (this.playerDiceRolls[i]==="Attack"){
      playersAttack += 1;
    } else if (this.playerDiceRolls[i]==="Shield"){
      playersDefense += 1;
    }
  }
  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

78. We're going to follow this pattern until all seven variables are covered. So let's refactor our test again checking for a "Counter" roll.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield", "Counter"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "1,1,1,0,0,0,0";
  expect(result).toEqual(mockedString);
});
```

DUNGIONCRAWLER COMPONENT TDD

79. Now with our test failing again, let's implement the code to make it pass.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  for(let i = 0; i < 6; i++){
    if (this.playerDiceRolls[i]==="Attack"){
      playersAttack += 1;
    } else if (this.playerDiceRolls[i]==="Shield"){
      playersDefense += 1;
    } else if (this.playerDiceRolls[i]==="Counter"){
      playersCounter += 1;
    }
  }
  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

Once again, with our test passing, let's refactor it to check for "Flee" dice rolls.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield", "Counter", "Flee"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "1,1,1,1,0,0,0";
  expect(result).toEqual(mockedString);
});
```

Now that it's failing again, let's implement the code to make it pass.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  for(let i = 0; i < 6; i++){
    if (this.playerDiceRolls[i]==="Attack"){
      playersAttack += 1;
    } else if (this.playerDiceRolls[i]==="Shield"){
      playersDefense += 1;
    } else if (this.playerDiceRolls[i]==="Counter"){
      playersCounter += 1;
    } else if (this.playerDiceRolls[i]==="Flee"){
      flee += 1;
    }
  }
  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

DUNGIONCRAWLER COMPONENT TDD

80. Since now we've covered all four cases for our player, let's just add two more to our array for six dice rolls.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield", "Counter", "Flee", "Attack", "Shield"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "2,2,1,1,0,0,0";
  expect(result).toEqual(mockedString);
});
```

Notice that the number associated with "Attack" and "Shield" both went up by one.

81. Now let's go ahead and begin counting the enemy dice rolls by refactoring our test again.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield", "Counter", "Flee", "Attack", "Shield"];
  component.enemyDiceRolls = ["Attack"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "2,2,1,1,1,0,0";
  expect(result).toEqual(mockedString);
});
```

Watch the test fail, so we can make it pass.

DUNGIONCRAWLER COMPONENT TDD

82. Now that we're testing for the enemy "Attack" die roll, let's implement it.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  for(let i = 0; i < 6; i++){
    if (this.playerDiceRolls[i]==="Attack"){
      playersAttack += 1;
    } else if (this.playerDiceRolls[i]==="Shield"){
      playersDefense += 1;
    } else if (this.playerDiceRolls[i]==="Counter"){
      playersCounter += 1;
    } else if (this.playerDiceRolls[i]==="Flee"){
      flee += 1;
    }
    if (this.enemyDiceRolls[i]==="Attack"){
      enemiesAttack += 1;
    }
  }
  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

Watch the test pass, now let's refactor again.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield", "Counter", "Flee", "Attack", "Shield"];
  component.enemyDiceRolls = ["Attack", "Shield"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "2,2,1,1,1,1,0";
  expect(result).toEqual(mockedString);
});
```

Take note, as we've been adding a different dice roll to our arrays, we've been slowly incrementing the number in our "mockedString" shifting over by one each time.

DUNGIONCRAWLER COMPONENT TDD

83. Now that our test is failing again, let's implement the next "if else" statement to make it pass.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  for(let i = 0; i < 6; i++){
    if (this.playerDiceRolls[i]==="Attack"){
      playersAttack += 1;
    } else if (this.playerDiceRolls[i]==="Shield"){
      playersDefense += 1;
    } else if (this.playerDiceRolls[i]==="Counter"){
      playersCounter += 1;
    } else if (this.playerDiceRolls[i]==="Flee"){
      flee += 1;
    }
    if (this.enemyDiceRolls[i]==="Attack"){
      enemiesAttack += 1;
    } else if (this.enemyDiceRolls[i]==="Shield"){
      enemiesDefense += 1;
    }
  }
  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

Now all we need to check for is if the enemy rolls a "Counter", so let's refactor again.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield", "Counter", "Flee", "Attack", "Shield"];
  component.enemyDiceRolls = ["Attack", "Shield", "Counter"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "2,2,1,1,1,1,1";
  expect(result).toEqual(mockedString);
});
```

The test will fail again, so let's implement the rest of the code to make it pass.

DUNGIONCRAWLER COMPONENT TDD

84. The final method should look like the one below.

```
calculateDiceRolls(): string {
  let playersAttack = 0;
  let playersDefense = 0;
  let playersCounter = 0;
  let flee = 0;
  let enemiesAttack = 0;
  let enemiesDefense = 0;
  let enemiesCounter = 0;

  for(let i = 0; i < 6; i++){
    if (this.playerDiceRolls[i]==="Attack"){
      playersAttack += 1;
    } else if (this.playerDiceRolls[i]==="Shield"){
      playersDefense += 1;
    } else if (this.playerDiceRolls[i]==="Counter"){
      playersCounter += 1;
    } else if (this.playerDiceRolls[i]==="Flee"){
      flee += 1;
    }
    if (this.enemyDiceRolls[i]==="Attack"){
      enemiesAttack += 1;
    } else if (this.enemyDiceRolls[i]==="Shield"){
      enemiesDefense += 1;
    } else if (this.enemyDiceRolls[i]==="Counter"){
      enemiesCounter += 1;
    }
  }
  return `${playersAttack},${playersDefense},${playersCounter},${flee},${enemiesAttack},${enemiesDefense},${enemiesCounter}`
}
```

Now our test should completely pass, but to be completely thorough, let's refactor our test one more time so that our enemy has six dice rolled in total.

```
it('should calculate all dice rolls and return the result', () => {
  component.playerDiceRolls = ["Attack", "Shield", "Counter", "Flee", "Attack", "Shield"];
  component.enemyDiceRolls = ["Attack", "Shield", "Counter", "Attack", "Shield", "Counter"];
  let result: string = component.calculateDiceRolls();
  let mockedString: string = "2,2,1,1,2,2,2";
  expect(result).toEqual(mockedString);
});
```

Notice how our final string is now the same as the one we originally took a look at, but we followed true TDD to reach it.

DUNGIONCRAWLER COMPONENT TDD

85. We have one final method to implement, which is our "calculateDamage" method. This method is going to check for damage, award score points, check for end of game, and display this damage on a turn by turn basis.

86. As a quick reminder, our wireframe has three fields in the HTML which are associated with "dmgDealt", "dmgBlocked", and "dmgTaken". These will be calculated every turn so the player can see these details. So let's start there.

87. First let's look at the HTML code on line 19-21.

```
<p><i>Damage Dealt this turn: {{dmgDealt}}</i></p>
<p><i>Damage Blocked this turn: {{dmgBlocked}}</i></p>
<p><i>Damage Taken this turn: {{dmgTaken}}</i></p>
```

From this detail, we're going to need to create three more global variables, so let's create them.

```
dmgDealt : number;
dmgBlocked : number;
dmgTaken : number;
```

88. Our new method is going to have to start off by calling the method we just created "calculateDiceRolls", and then split/parse the string into the individual numbers so we can do operations on them. Let's create a basic test just to make sure we call the other method and add onto our test over time.

```
it('should calculate damage if playersAttack > enemiesDefense + enemiesCounter', () => {
  spyOn(component, 'calculateDiceRolls');
  component.calculateDamage();
  expect(component.calculateDiceRolls).toHaveBeenCalled();
});
```

DUNGIONCRAWLER COMPONENT TDD

89. So now let's go back to our blank method we created eariler for TDD purposes.

```
calculateDamage() {
}
```

Now, run our test and watch it fail. Next let's get the test to pass by implementating the code.

```
calculateDamage() {
  this.calculateDiceRolls();
}
```

90. Next we're going to have our spy return a string which needs to match the same format as what "calculateDiceRolls" returns. So we can refactor our test to include this.

```
it('should calculate damage if playersAttack > enemiesDefense + enemiesCounter', () => {
  let mockedString: string = "3,0,0,0,0,2,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.calculateDiceRolls).toHaveBeenCalled();
});
```

As a reminder, given this "mockedString" above, our player rolled three "Attack" dice, and our enemy rolled two "Shield" dice. Which means, we should have dealt one damage to the enemy. During that action, our components global variable "dmgDealt" should increment to one, the components global variable "enemyHealth" should decrement by one, and for every one damage we deal, we get thirty points. So we should also set the global variable "playerScore" to thirty.

DUNGIONCRAWLER COMPONENT TDD

91. Following the system we just talked about we can refactor our test to check for it.

```
it('should calculate damage if playersAttack > enemiesDefense + enemiesCounter',
  component.dmgDealt = 0;
  component.enemyHealth = 10;
  component.playerScore = 0;
  component.dmgTaken = 0;
  component.playerHealth = 10;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "3,0,0,0,0,2,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.calculateDiceRolls).toHaveBeenCalled();
  expect(component.dmgDealt).toEqual(1);
  expect(component.enemyHealth).toEqual(9);
  expect(component.playerScore).toEqual(30);
});
```

Notice, we put the order of the string in the comment so we know what the number's inside of our string mean at all times. So we begin by initializing everything we need inside of our method, and then checking to make sure that the correct amount of points was awarded, damage dealt is correct, and that the enemys health is correct. If we run this test now, it will fail.

DUNGIONCRAWLER COMPONENT TDD

92. Next we need to get our test to pass. The first code we need to implement is a string array that splits on comma's for our string. Then we need to parse each one as a number for us to use.

```
calculateDamage() {
  let getDiceRolled: string[] = this.calculateDiceRolls().split(',');
  let playersAttack: number = JSON.parse(getDiceRolled[0]);
  let playersDefense: number = JSON.parse(getDiceRolled[1]);
  let playersCounter: number = JSON.parse(getDiceRolled[2]);
  let flee: number = JSON.parse(getDiceRolled[3]);
  let enemiesAttack: number = JSON.parse(getDiceRolled[4]);
  let enemiesDefense: number = JSON.parse(getDiceRolled[5]);
  let enemiesCounter: number = JSON.parse(getDiceRolled[6]);
```

When we run the "split(',')" method, it will convert the string into a string array for us. The next thing we need to do is use the built-in "JSON.parse" method on each index which will convert our string at that index into a number. If we run our test now, it will still fail; all we've done thus far is setup. Next thing we need to do, is reset our global variables "dmgDealt", "dmgBlocked", and "dmgTaken" to zero since they are calculated on a turn by turn basis.

```
calculateDamage() {
  let getDiceRolled: string[] = this.calculateDiceRolls().split(',');
  let playersAttack: number = JSON.parse(getDiceRolled[0]);
  let playersDefense: number = JSON.parse(getDiceRolled[1]);
  let playersCounter: number = JSON.parse(getDiceRolled[2]);
  let flee: number = JSON.parse(getDiceRolled[3]);
  let enemiesAttack: number = JSON.parse(getDiceRolled[4]);
  let enemiesDefense: number = JSON.parse(getDiceRolled[5]);
  let enemiesCounter: number = JSON.parse(getDiceRolled[6]);
  this.dmgDealt = 0;
  this.dmgBlocked = 0;
  this.dmgTaken = 0;
```

If we run the test now, it still fails, but our setup inside our method is done.

DUNGIONCRAWLER COMPONENT TDD

93. Now let's get this test to completely pass by implementing out first "if" statement.

```
if (playersAttack > enemiesDefense) {
  this.dmgDealt = playersAttack-enemiesDefense;
  this.enemyHealth -= playersAttack-enemiesDefense;
  this.playerScore += this.dmgDealt*30;
}
```

Now run our test again and watch it pass. Most of the next test's we create are going to be very similar just testing for different combinations of dice rolls. They will be checking to make sure the playerScore, playerHealth, dmgTaken, dmgDealt, dmgBlocked, and enemyHealth are correctly calculated based on the string we retrieved from our method "calculateDiceRolls()".

DUNGIONCRAWLER COMPONENT TDD

94. So now let's go ahead and create our next test.

```
it('should calculate damage if playersAttack > 0 && enemiesCounter > 0', () => {
  component.dmgDealt = 0;
  component.enemyHealth = 10;
  component.playerScore = 0;
  component.dmgTaken = 0;
  component.playerHealth = 10;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "4,0,0,0,0,0,2";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.dmgDealt).toEqual(2);
  expect(component.enemyHealth).toEqual(8);
  expect(component.dmgTaken).toEqual(2);
  expect(component.playerHealth).toEqual(8);
  expect(component.playerScore).toEqual(36);
});
```

Here we're checking to see if the player rolls four "Attack" dice and the enemy rolls two "Counter" dice. The way "Counter" works, if the player rolls an "Attack" die, and the opponent rolls one "Counter" die, they block the players attack and deal one damage back to the player. We're also going to subtract twelve points everytime a player get's countered. In this case the opponent rolled two counter dice, so we lose twenty-four points, but gained sixty since two attacks where successful, which is a net gain of thirty-six points. Furthermore, both the enemy and player lose two health as a result. So let's first refactor our current code to make sure the player doesn't get points if the enemy rolls more counter's than the players "Attack".

```
if (playersAttack > enemiesDefense + enemiesCounter) {
  this.dmgDealt = playersAttack-enemiesDefense-enemiesCounter;
  this.enemyHealth -= playersAttack-enemiesDefense-enemiesCounter;
  this.playerScore += this.dmgDealt*30;
}
```

DUNGIONCRAWLER COMPONENT TDD

95. Now let's implement another "if" statement to decrement points, increment our global variable "dmgTaken", and decrement our global variable "playerHealth".

```
if (playersAttack > 0 && enemiesCounter > 0){
  if(playersAttack > enemiesCounter) {
    this.dmgTaken += enemiesCounter;
    this.playerHealth -= enemiesCounter;
  }
  this.playerScore -= this.dmgTaken*12;
}
```

Now our test will pass. However, we need to create another test for the case when "playersAttack" is equal to "enemiesCounter".

```
it('should calculate damage if playersAttack <= enemiesCounter', () => {
  component.dmgDealt = 0;
  component.enemyHealth = 10;
  component.playerScore = 0;
  component.dmgTaken = 0;
  component.playerHealth = 10;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "4,0,0,0,0,0,4";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.dmgDealt).toEqual(0);
  expect(component.enemyHealth).toEqual(10);
  expect(component.dmgTaken).toEqual(4);
  expect(component.playerHealth).toEqual(6);
  expect(component.playerScore).toEqual(-48);
});
```

Run this test now and watch it fail.

DUNGIONCRAWLER COMPONENT TDD

96. With our new test failing, let's go ahead and implement the "else" statement we need.

```
if (playersAttack > 0 && enemiesCounter > 0){
  if(playersAttack > enemiesCounter) {
    this.dmgTaken += enemiesCounter;
    this.playerHealth -= enemiesCounter;
  } else {
    this.dmgTaken += playersAttack;
    this.playerHealth -= playersAttack;
  }
  this.playerScore -= this.dmgTaken*12;
}
```

Now run our test again and watch it pass.

DUNGIONCRAWLER COMPONENT TDD

97. Next we're going to flip the most recent logic around, and create a test to check the enemy attacking the player.

```
it('should calculate damage if enemiesAttack > playersDefense + playersCounter', () => {
  component.dmgDealt = 0;
  component.enemyHealth = 10;
  component.playerScore = 0;
  component.dmgTaken = 0;
  component.playerHealth = 10;
  component.dmgBlocked = 0;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "0,3,0,0,4,0,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.dmgTaken).toEqual(1);
  expect(component.playerHealth).toEqual(9);
  expect(component.dmgBlocked).toEqual(3);
  expect(component.playerScore).toEqual(-6);
});
```

In this case, we rolled three "Shield" and the enemy rolled four "Attack". So we successfully blocked three damage and took one. If the player takes damage when trying to defend, they only lose six points per one point of damage dealt. Now let's implement the code to pass this failing test.

```
if (enemiesAttack > playersDefense) {
  this.dmgTaken = enemiesAttack-playersDefense;
  this.playerHealth -= enemiesAttack-playersDefense;
  this.dmgBlocked += playersDefense;
  this.playerScore -= this.dmgTaken*6;
}
```

Now run the test again and watch it pass.

DUNGIONCRAWLER COMPONENT TDD

98. Now we need implement our next test to check to see if the player is "Countering". When a "Counter" is performed, it is possible to roll a "Counter" die, as well as a "Shield" die, in which case we shouldn't subtract any points if the players defense combined with there counter is higher than the enemies attack.

```
it('should calculate damage if enemiesAttack > 0 && playersCounter > 0', () => {
  component.dmgDealt = 0;
  component.enemyHealth = 10;
  component.playerScore = 0;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "0,1,2,0,4,0,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.dmgDealt).toEqual(2);
  expect(component.enemyHealth).toEqual(8);
  expect(component.playerScore).toEqual(34);
});
```

Now to make this test pass we need to first refactor our most recently added "if" statement.

```
if (enemiesAttack > playersDefense + playersCounter) {
  this.dmgTaken = enemiesAttack-playersDefense-playersCounter;
  this.playerHealth -= enemiesAttack-playersDefense-playersCounter;
  this.dmgBlocked += playersDefense + playersCounter;
  this.playerScore -= this.dmgTaken*6;
}
```

DUNGIONCRAWLER COMPONENT TDD

99. Now that our current implementation was refactored, let's go ahead and implement our next "if" statement to make this test fully pass.

```
if (enemiesAttack > 0 && playersCounter > 0){
  if(enemiesAttack > playersCounter) {
    this.dmgDealt += playersCounter;
    this.enemyHealth -= playersCounter;
  }
  this.playerScore += this.dmgDealt*20;
}
```

Notice here, for every successful "Counter" the player performs into an enemy "Attack", they are awarded twenty points. Now we need to create yet another test to check if "enemiesAttack" is equal to our "playersCounter".

```
it('should calculate damage if enemiesAttack == playersCounter', () => {
  component.dmgDealt = 0;
  component.enemyHealth = 10;
  component.playerScore = 0;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "0,0,4,0,4,0,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.dmgDealt).toEqual(4);
  expect(component.enemyHealth).toEqual(6);
  expect(component.playerScore).toEqual(88);
});
```

DUNGIONCRAWLER COMPONENT TDD

100. Now with our test created, let's implement the "else" statement we need.

```
if (enemiesAttack > 0 && playersCounter > 0){
  if(enemiesAttack > playersCounter) {
    this.dmgDealt += playersCounter;
    this.enemyHealth -= playersCounter;
  } else {
    this.dmgDealt += enemiesAttack;
    this.enemyHealth -= enemiesAttack;
  }
  this.playerScore += this.dmgDealt*20;
}
```

Notice that our test still fails even after this implementation. We need to implement one more if statement to make this test pass because for every successful block or counter we want to also award the player an extra two points.

```
if (enemiesAttack <= playersCounter){
  this.dmgBlocked += enemiesAttack;
  this.playerScore += this.dmgBlocked*2;
}
```

Now our test will successfully pass.

DUNGIONCRAWLER COMPONENT TDD

101. Our next step is to create another test to make sure we get two points for every successful defend as well.

```
it('should calculate damage if enemiesAttack <= playersCounter + playersDefense', () => {
  component.dmgBlocked = 0;
  component.playerScore = 0;

  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "0,2,0,0,2,0,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.dmgBlocked).toEqual(2);
  expect(component.playerScore).toEqual(4);

});
```

Now we can run this test and watch it fail. Now let's refactor our most recently added "if" statement to make this test pass.

```
if (enemiesAttack <= playersCounter + playersDefense){
  this.dmgBlocked += enemiesAttack;
  this.playerScore += this.dmgBlocked*2;
}
```

DUNGIONCRAWLER COMPONENT TDD

102. Now that we have our point system in place, let's go ahead and implement the next step, which is checking for win/lose conditions. The game ends on three separate conditionals. First, if the player attempts to "flee", if they succeed then the game ends. Secondly, if the enemies health reaches zero, and that enemy is the last enemy for the difficulty selected. Lastly, if the players health reaches zero, in this case they've died and the game ends.

103. For all three of these conditionals, we're going to need to open our dialog box that we created eariler in this section of the guide ("openEndingDialog" method). So now we can start with case one, which involves the player using "flee". If the player successfully flee's we're going to deduct them fifty points for being a coward. So with this logic, we can create a test for this.

```
it('should flee > enemiesAttack', () => {
  component.playerScore = 0;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "0,0,0,3,0,0,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  component.calculateDamage();
  expect(component.playerScore).toEqual(-50);
});
```

This test is following the same outline as the previous one's we just created, with three "flee" dice rolled. So because they where successful with the "flee" attempt, we deduct 50 points, and expect their score to be –50. If we run this test, we're watch it fail.

DUNGIONCRAWLER COMPONENT TDD

104. With our new test failing, let's implement the code to make it pass.

```
if (flee > enemiesAttack) {
  this.playerScore -= 50;
}
```

By adding this "if" statement, we are now accounting for the "flee" action. We're still missing one piece though, which is opening our dialog box by calling our method "openEndingDialog". So we can begin by refactoring our current test to check for this.

```
it('should flee > enemiesAttack', () => {
  component.playerScore = 0;
  //playersAttack, playersDefense, playersCounter, flee,
  //enemiesAttack, enemiesDefense, enemiesCounter
  let mockedString: string = "0,0,0,3,0,0,0";
  spyOn(component, 'calculateDiceRolls').and.returnValue(mockedString);
  spyOn(component, 'openEndingDialog');
  component.calculateDamage();
  expect(component.playerScore).toEqual(-50);
  expect(component.openEndingDialog).toHaveBeenCalledTimes(1);
  expect(component.openEndingDialog).toHaveBeenCalledWith('You ran away, Coward! Ending Score: ');
});
```

All we have to do here, is "spyOn" our dialog box method and expect that it's been called once, and that it was called with our specific string. If we run this test again, it will now fails.

DUNGIONCRAWLER COMPONENT TDD

105. Our next step is to make our test pass again, so we need to refactor our "if" statement we just added.

```
if (flee > enemiesAttack) {
  this.playerScore -= 50;
  this.openEndingDialog('You ran away, Coward! Ending Score: ');
}
```

Notice, our test will now pass and we've covered the case for the "flee" option.

106. Our next step is to check if the enemy's health was reduced to zero. If they have zero or less health then we can check if the game is over. So let's begin creating a most basic test.

```
it('should set the players score, increase mob counter, and check for end of game', () => {
  component.enemyHealth = 0;
  component.calculateDamage();
  expect(component.enemyHealth).toEqual(10);
});
```

All we're doing here is resetting the opponent's health to ten when they die, because we have more than one opponent we need to reset this health for each opponent. So now we run our test and watch it fail. Then we implement the code we need to pass.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
}
```

DUNGIONCRAWLER COMPONENT TDD

107. So now that we set the opponent's health back to ten, our next step is to save our player's score into "sessionStorage". Let's refactor our test to account for that.

```
it('should set the players score, increase mob counter, and check for end of game', () => {
  component.enemyHealth = 0;
  component.playerScore = 1219;
  component.calculateDamage();
  expect(sessionStorage.getItem('score')).toEqual("1219");
  expect(component.enemyHealth).toEqual(10);
});
```

Now we're setting the score specifically to 1219 and checking to see if it's stored into our "sessionStorage" variable "score". Once again, watch our test fail. Now let's implement the next piece of logic to make this pass.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
  sessionStorage.setItem('score', this.playerScore.toString());
}
```

108. The next thing we need to add is to increase the global variable "mobCounter" so we know which enemy we're on. So let's refactor our test again to check for this.

```
it('should set the players score, increase mob counter, and check for end of game', () => {
  component.enemyHealth = 0;
  component.playerScore = 1219;
  component.mobCounter = 2;
  component.calculateDamage();
  expect(component.mobCounter).toEqual(3);
  expect(sessionStorage.getItem('score')).toEqual("1219");
  expect(component.enemyHealth).toEqual(10);
});
```

DUNGIONCRAWLER COMPONENT TDD

109. Now with our test failing again, let's refactor our "if" statement to make it pass.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
  sessionStorage.setItem('score', this.playerScore.toString());
  this.mobCounter = this.mobCounter+1;
}
```

Now that our test is passing again, let's continue with this test and check for an end of game condition.

110. Since we want "easy" difficulty to stop after two enemies, "medium" to stop after three enemies, and "hard" to stop after four enemies we're need to check for these. The way we're going to do this is by creating an "if" statement and checking the difficulty that was selected as well as which "mob" we're on. In this case, we increment "mobCounter" before any check, so if this counter is equal to "3" and the difficulty was set to easy, we have won since we technically only beat two enemies. Let's refactor our test to check for this.

DUNGIONCRAWLER COMPONENT TDD

111. In our test we need to set our sessionStorage difficulty.

```
sessionStorage.setItem('difficulty', 'Easy');
```

Now let's think for a second, this will pass the "if" statement we need to check for end of game, but what is the body of the statement going to do? When the game ends we want to open a dialog box with some input as a string. If we remember from pages 172-180, we already created a method "openEndingDialog" that will fit our role perfectly. So now we want to "spyOn" that method and call it inside our "if" statement. With this addition, our test should now look like the image below.

```
it('should set the players score, increase mob counter, and check for end of game', () => {
  component.enemyHealth = 0;
  component.playerScore = 1219;
  component.mobCounter = 2;
  sessionStorage.setItem('difficulty', 'Easy');
  spyOn(component, 'openEndingDialog');

  component.calculateDamage();
  expect(component.openEndingDialog).toHaveBeenCalledTimes(1);
  expect(component.openEndingDialog).toHaveBeenCalledWith('You have Won! Ending Score: ');
  expect(component.mobCounter).toEqual(3);
  expect(sessionStorage.getItem('score')).toEqual("1219");
  expect(component.enemyHealth).toEqual(10);
});
```

We first "spyOn" the method we're not testing "openEndingDialog", which is simply mocking that method. Next, we make sure we only call this method one time and that we called it with the message "You have Won! Ending Score: ");

DUNGIONCRAWLER COMPONENT TDD

112. If we run our test now, we'll notice it fails. So let's implement just enough code to make it pass.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
  sessionStorage.setItem('score', this.playerScore.toString());
  this.mobCounter = this.mobCounter+1;
  if ((sessionStorage.getItem('difficulty') === "Easy" && this.mobCounter === 3)) {
    this.openEndingDialog('You have Won! Ending Score: ');
  }
}
```

However, we now need this to do one more piece of logic. The last thing we need this outer "if" statement to include is a trigger to generate the next enemy. We can look at our HTML for a hint on how to do this. On line 5,

```
[nextMob] = "triggerNextMob.asObservable()"
```

Our "app-enemy-ui" has a variable "nextMob" which is an "Observable<void>". We set this variable in a previous section of this guide, but now we need to send that a trigger so it generates the next enemy. In the HTML we need to create a global variable "triggerNextMob" and make it a "Subject<void>". This is very similar to the other portions of this section with the global variables "triggerEnemyDiceRolls" and "triggerEnemyAttackType". So now let's refactor our test again to incorporate this global variable.

DUNGIONCRAWLER COMPONENT TDD

113. After the refactor our test should look like,

```
it('should set the players score, increase mob counter, and check for end of game', () => {
  let mockNextMob: any = {
    next: jasmine.createSpy('next')
  }
  component.enemyHealth = 0;
  component.playerScore = 1219;
  component.mobCounter = 2;
  component.triggerNextMob = mockNextMob;
  sessionStorage.setItem('difficulty', 'Easy');
  spyOn(component, 'openEndingDialog');

  component.calculateDamage();
  expect(component.openEndingDialog).toHaveBeenCalledTimes(1);
  expect(component.openEndingDialog).toHaveBeenCalledWith('You have Won! Ending Score: ');
  expect(component.mobCounter).toEqual(3);
  expect(sessionStorage.getItem('score')).toEqual("1219");
  expect(component.enemyHealth).toEqual(10);
  expect(component.triggerNextMob.next).toHaveBeenCalledTimes(1);
});
```

The first thing we do is mock this "Subject" next method by creating a spy. We then inject this new object into our actual component's global variable so when this variables "next" method is called, it will refer to our spy as opposed to calling the actual method. We then make sure our injected spy was at least called one time. Now let's implement enough code to make this test compile by adding the global variable.

```
triggerNextMob: Subject<void> = new Subject<void>();
```

Now let's run our test and watch it fail since the "spy has not been called". Next we can implement the code to make it pass.

DUNGIONCRAWLER COMPONENT TDD

114. The code we need to implement is a single line of code after our "if" statement.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
  sessionStorage.setItem('score', this.playerScore.toString());
  this.mobCounter = this.mobCounter+1;
  if ((sessionStorage.getItem('difficulty') === "Easy" && this.mobCounter === 3)) {
    this.openEndingDialog('You have Won! Ending Score: ');
  }
  this.triggerNextMob.next();
}
```

Now watch our test pass. Next let's create another test to check for "Medium" difficulty and "mobCounter" is equal to "4".

```
it('should set the players score, increase mob counter, and generate next mob for medium', () => {
  let mockNextMob: any = {
    next: jasmine.createSpy('next')
  }
  component.enemyHealth = 0;
  component.playerScore = 1519;
  component.mobCounter = 3;
  component.triggerNextMob = mockNextMob;
  sessionStorage.setItem('difficulty', 'Medium');
  spyOn(component, 'openEndingDialog');

  component.calculateDamage();
  expect(component.openEndingDialog).toHaveBeenCalledTimes(1);
  expect(component.openEndingDialog).toHaveBeenCalledWith('You have Won! Ending Score: ');
  expect(component.mobCounter).toEqual(4);
  expect(sessionStorage.getItem('score')).toEqual("1519");
  expect(component.enemyHealth).toEqual(10);
  expect(component.triggerNextMob.next).toHaveBeenCalledTimes(1);
});
```

This test is very similar to the one we just created, only difference is, we changed the "mobCounter" to "3" so it will increment to "4", and set the sessionStorage to "Medium".

DUNGIONCRAWLER COMPONENT TDD

115. Next run this new test and watch it fail. Now let's just add the conditional to our inner "if" statement to make it pass.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
  sessionStorage.setItem('score', this.playerScore.toString());
  this.mobCounter = this.mobCounter+1;
  if ((sessionStorage.getItem('difficulty') === "Easy" && this.mobCounter === 3)
  || (sessionStorage.getItem('difficulty') === "Medium" && this.mobCounter === 4)) {
    this.openEndingDialog('You have Won! Ending Score: ');
  }
  this.triggerNextMob.next();
}
```

116. The last test we need is to check for the "Hard" difficulty and "mobCounter" to equal "5". So let's make that test now.

```
it('should set the players score, increase mob counter, and generate next mob for hard', () => {
  let mockNextMob: any = {
    next: jasmine.createSpy('next')
  }
  component.enemyHealth = 0;
  component.playerScore = 1712;
  component.mobCounter = 4;
  component.triggerNextMob = mockNextMob;
  sessionStorage.setItem('difficulty', 'Hard');
  spyOn(component, 'openEndingDialog');

  component.calculateDamage();
  expect(component.openEndingDialog).toHaveBeenCalledTimes(1);
  expect(component.openEndingDialog).toHaveBeenCalledWith('You have Won! Ending Score: ');
  expect(component.mobCounter).toEqual(5);
  expect(sessionStorage.getItem('score')).toEqual("1712");
  expect(component.enemyHealth).toEqual(10);
  expect(component.triggerNextMob.next).toHaveBeenCalledTimes(1);
});
```

DUNGIONCRAWLER COMPONENT TDD

117. Now with our new test failing, let's implement the last conditional of our "if" statement to make this test pass.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
  sessionStorage.setItem('score', this.playerScore.toString());
  this.mobCounter = this.mobCounter+1;
  if ((sessionStorage.getItem('difficulty') === "Easy" && this.mobCounter === 3)
  || (sessionStorage.getItem('difficulty') === "Medium" && this.mobCounter === 4)
  || (sessionStorage.getItem('difficulty') === "Hard" && this.mobCounter === 5)) {
    this.openEndingDialog('You have Won! Ending Score: ');
  }
  this.triggerNextMob.next();
}
```

DUNGIONCRAWLER CHECKPOINT 4

1. Now that we check the enemy health and end the game, the only thing we're missing is our last check for the player's health. If the player's health is less than or equal to zero, they have died. If they die, let's dock them twenty-five points and call our method "openEndingDialog" with a message "You have Died! Ending Score:". With these requirements, create a test for these, make sure the test fails, then implement the code to make it pass.

DUNGIONCRAWLER CHECKPOINT 4 SOLUTION

1. Following true TDD, our most basic test will check for a conditional "playerHealth <= 0" and that the "playerScore" has been decremented by twenty-five points. This test would look as follows,

```
it('should check if player died', () => {
  component.playerScore = 0;
  component.playerHealth = 0;
  component.calculateDamage();
  expect(component.playerScore).toEqual(-25);
});
```

Now let's watch this test fail, and implement the code to make it pass.

```
if (this.enemyHealth <= 0){
  this.enemyHealth = 10;
  sessionStorage.setItem('score', this.playerScore.toString());
  this.mobCounter = this.mobCounter+1;
  if ((sessionStorage.getItem('difficulty') === "Easy" && this.mobCounter === 3)
  || (sessionStorage.getItem('difficulty') === "Medium" && this.mobCounter === 4)
  || (sessionStorage.getItem('difficulty') === "Hard" && this.mobCounter === 5)) {
    this.openEndingDialog('You have Won! Ending Score: ');
  }
  this.triggerNextMob.next();
}
if(this.playerHealth <= 0){
  this.playerScore -= 25;
}
```

DUNGIONCRAWLER CHECKPOINT 4 SOLUTION

2. Next, we just need to refactor our test to check if our method "openEndingDialog" is called.

```
it('should check if player died', () => {
  component.playerScore = 0;
  component.playerHealth = 0;
  spyOn(component, 'openEndingDialog');
  component.calculateDamage();
  expect(component.playerScore).toEqual(-25);
  expect(component.openEndingDialog).toHaveBeenCalledTimes(1);
  expect(component.openEndingDialog).toHaveBeenCalledWith('You have Died! Ending Score: ');
});
```

All we need to do here is "spyOn" our other method "openEndingDialog" so that we aren't calling the actual method but our spy instead. Then we check to make sure we only called it one time and that it was called with the proper message "You have Died! Ending Score: ".

3. Let's run our test now, watch it fail, and implement the code to make it pass.

```
if(this.playerHealth <= 0){
  this.playerScore -= 25;
  this.openEndingDialog('You have Died! Ending Score: ');
}
```

ANGULAR SERVICES TDD

1. Service Setup Pg 218-218
2. Roll Method TDD Pg 219-220
3. AddDice Method TDD Pg 221-223
4. Checkpoint 1 (Setup) Pg 222-223
5. HTTP TDD using Mock Data Pg 224-233
6. Setting up exported Mock Data (Mob Order) Pg 228-229
7. Checkpoint 2 (Sending 'GET' request via Tests) Pg 230-232
8. Setting up exported Mock Data (Scoreboard) Pg 234-234
9. Checkpoint 3 (Sending 'GET' and 'POST' request via Tests) Pg 235-237
10. Checkpoint 4 (Sending 'GET' and 'POST request via Code) Pg 238-239

ANGULAR SERVICES TDD

1. CD into enablementorium, and check branch.

```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git branch
* develop
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

2. Now let's checkout to "feature/serviceUnfinishedTDD", this branch has no TDD involving services. The final version is on "feature/serviceFinishedTDD".

```
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$ git checkout feature/serviceUnfinishedTDD
Branch 'feature/serviceUnfinishedTDD' set up to track remote branch 'feature/serviceUnfinishedTDD' from 'origin'.
Switched to a new branch 'feature/serviceUnfinishedTDD'
cdelabs@cdelabs-OptiPlex-7050:~/workspace/enablementorium$
```

ROLL METHOD TDD

1. For our game, we need to create a roll method that will roll a die and end in a result from 1-6.

2. Navigate to src/apps/services/dice.service.spec.ts

3. Change "describe" into an "fdescribe" so we can run just this block of tests.

```
fdescribe('DiceService', () => {
  beforeEach(() => TestBed.configureTestingModule({}));

  it('should be created', () => {
    const service: DiceService = TestBed.get(DiceService);
    expect(service).toBeTruthy();
  });
});
```

4. Now let's convert "const service" into a global value so we can use it for all of our tests inside our "fdescribe".

```
fdescribe('DiceService', () => {
  let service: DiceService;

  beforeEach(() => { TestBed.configureTestingModule({
    })
    service = TestBed.get(DiceService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });      Michael Grammens, a month ago • WOWOWOWO
});
```

ROLL METHOD TDD

5. With our service now being global, let's make a test to check for all six possibilities that a random die might return.

```
it('should roll a random number between 1 and 6', ()=> {
  spyOn(Math, 'random').and.returnValues(.10, .20, .35, .55, .75, .85);
  service.roll().subscribe(num => {
    expect(num).toBe(1);
  });
  service.roll().subscribe(num => {
    expect(num).toBe(2);
  });
  service.roll().subscribe(num => {
    expect(num).toBe(3);
  });
  service.roll().subscribe(num => {
    expect(num).toBe(4);
  });
  service.roll().subscribe(num => {
    expect(num).toBe(5);
  });
  service.roll().subscribe(num => {
    expect(num).toBe(6);
  });
});      You, 2 minutes ago • Uncommitted changes
```

6. Now create the method to make the test compile following the TDD methodology.

```
const MAX_DICE = 6;
Michael Grammens, a month ago | 1 author (Michael Grammens)
@Injectable({
  providedIn: 'root'
})
export class DiceService {

  constructor() { }

  roll(): Observable<number> {
    return of(Math.ceil(Math.random() * MAX_DICE));
  }      Michael Grammens, a month ago • Added a LOT
```

7. Now run "ng test" from the command line and watch the test pass.

ADDDICE METHOD TDD

1. The next step in the dice.service now that we can roll dice, is to add an image associated with the roll of the die and the action the player is attempting to make. For example, (if the player is attacking, and the die rolls 1, 2, or 3 we want to set the image to a skull (Attack.png) showing that 1 hit die was rolled.) We would then follow this process for all 6 dice.

```
if (attackType === "Attack" && [1, 2, 3].includes(response)){
```

2. To follow this process, we now can make tests cases given a particular die roll and the attackType to be a particular result.

```
it('should return the image Attack.png otherwise Nothing.png', () => {
  expect(service.addDice(1, "Attack")).toBe("Attack");
  expect(service.addDice(4, "Attack")).toBe("Nothing");
});

it('should return the image Counter.png otherwise Nothing.png', () => {
  expect(service.addDice(1, "Counter")).toBe("Counter");
  expect(service.addDice(5, "Counter")).toBe("Nothing");
});

it('should return the image Shield.png otherwise Nothing.png', () => {
  expect(service.addDice(3, "Counter")).toBe("Shield");
  expect(service.addDice(4, "Defend")).toBe("Shield");
  expect(service.addDice(5, "Defend")).toBe("Nothing");
});

it('should return the image Flee.png otherwise Nothing.png', () => {
  expect(service.addDice(2, "Flee")).toBe("Flee");
  expect(service.addDice(3, "Flee")).toBe("Nothing");
});
```

3. Now we write the actual code to make this pass.

```
addDice(response: number, attackType: string): string {
  let imgType = "";
  if (attackType === "Attack" && [1, 2, 3].includes(response)){
    imgType = "Attack";
  } else if (attackType === "Counter" && (response === 1 || response === 2)){
    imgType = "Counter";
  } else if (attackType === "Counter" && response === 3){
    imgType = "Shield"
  } else if (attackType === "Defend" && (response === 1 || response === 2 || response === 3 || response === 4)) {
    imgType = "Shield";
  } else if (attackType === "Flee" && (response === 1 || response === 2)) {
    imgType = "Flee";      Michael Grammens, a month ago • Added a bunch
  } else {
    imgType = "Nothing";
  }
  return imgType;
};
```

SERVICE CHECKPOINT 1

1. Now with dice.service being complete with 2 methods, we can move onto mob-order.service. This service is going to call our backend to retrieve a mobOrder. A mobOrder is just a model with multiple enemies and a difficulty, "Easy" means only 2 enemies while "Medium" means 3, and "Hard" means 4. The backend is going to create the enemies and give them a type randomly and then return the model.
2. For checkpoint 1, lets setup mob-order.service.spec.ts to where we can begin testing.

```
fdescribe('MobOrderService', () => {
  beforeEach(() => TestBed.configureTestingModule({}));

  it('should be created', () => {
    const service: MobOrderService = TestBed.get(MobOrderService);
    expect(service).toBeTruthy();
  });      Michael Grammens, a month ago • Added shallows with some
});
```

3. Given an "fdescribe" and a mocked mobOrder, we want to have the option to create a new test to create a mobOrder with "Easy". However, we get an compiler error. How do we fix this error?

```
fdescribe('MobOrderService', () => {
  beforeEach(() => TestBed.configureTestingModule({}));

  it('should be created', () => {
    const service: MobOrderService = TestBed.get(MobOrderService);
    expect(service).toBeTruthy();
  });

  const mockMobOrderEasy = {
    mob1: ClassListEnum.Ranger,
    mob2: ClassListEnum.Fighter,
    mob3: ClassListEnum.None,
    mob4: ClassListEnum.None,
    difficulty: "Easy"
  }
  it('should send a create mobOrder with difficulty set to easy', () => {
    service.createMobOrder("Easy");
  });
});
```

SERVICE CHECKPOINT 1 SOLUTION

1. As shown previously on "dice.service.spec.ts"

```
fdescribe('DiceService', () => {
  beforeEach(() => TestBed.configureTestingModule({}));

  it('should be created', () => {
    const service: DiceService = TestBed.get(DiceService);
    expect(service).toBeTruthy();
  });
});
```

2. The compiler error was simply stating that it could not find "service", so we can make it global.

```
fdescribe('DiceService', () => {
  let service: DiceService;

  beforeEach(() => { TestBed.configureTestingModule({
    })
    service = TestBed.get(DiceService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });      Michael Grammens, a month ago • WOWOWOWO
});
```

3. With the service now being global, the compiler error has switched to the method as it has not been implemented yet.

```
fdescribe('MobOrderService', () => {
  let service: MobOrderService;
      You, a minute ago • Uncommitted changes
  beforeEach(() => { TestBed.configureTestingModule({

    })
    service = TestBed.get(MobOrderService);
  });

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

  const mockMobOrderEasy = {
    mob1: ClassListEnum.Ranger,
    mob2: ClassListEnum.Fighter,
    mob3: ClassListEnum.None,
    mob4: ClassListEnum.None,
    difficulty: "Easy"
  }
  it('should send a create mobOrder with difficulty set to easy', () => {
    service.createMobOrder("Easy");
  });
});
```

CREATE MOBORDER METHOD TDD

1. We can now roll dice and add images depending on that roll, our next step is to make a service (mobOrder.service) that can make calls to our Java backend, and retrieve an order of enemies for our player to fight.

2. We begin by writing our mobOrder.service.spec.ts file, following TDD.

```
let service: MobOrderService;
let httpClient: HttpTestingController;

beforeEach(() => { TestBed.configureTestingModule({
    imports: [ HttpClientTestingModule ]
  })
  service = TestBed.get(MobOrderService);
  httpClient = TestBed.get(HttpTestingController);
});
```

3. We make the service global, and create an httpClient for mocking our http requests. Every testing file does not use app.module.ts, so all of our imports, declarations and providers we have to state ourselves.
4. Following the way TDD works, we only want to test our one method, hence, we don't want to be making actual calls to our Java backend in tests. So rather than "let httpClient: HttpClient", we use "let httpClient: HttpTestingController" so therefore it only mocks "fake calls" our Java backend. Just like for "HttpClient"; "HttpTestingController" has a module of its own, so we have to import it so angular can find it. Finally, we need to tell TestBed where to fetch our "HttpTestingController".

CREATE MOBORDER METHOD TDD

5. Now with our httpClient being defined, we can create a test that returns mockData using our service. Returning mockData is our way of faking a call to the Java backend, so the behavior is similar to that of creating an actual call.

```
const mockMobOrderEasy = {
  mob1: ClassListEnum.Ranger,
  mob2: ClassListEnum.Fighter,
  mob3: ClassListEnum.None,
  mob4: ClassListEnum.None,
  difficulty: "Easy"
}
it('should send a create mobOrder with difficulty set to easy', () => {
  const mockHttpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  }

  service.createMobOrder("Easy").subscribe(response => {
    expect(response).toBe(mockMobOrderEasy);
  });

  let request = httpClient.expectOne(environment.mobOrderSaveNew);
  expect(request.request.method).toBe('POST');
  expect(request.request.headers.get('Content-Type')).toBe(mockHttpOptions.headers.get('Content-Type'));
  request.flush(mockMobOrderEasy);      You, a few seconds ago • Uncommitted changes
});
```

6. In our test we can check for multiple things starting with the headers. Every request we send has some sort of headers, so we can test for that by creating our own mockHeaders.

7. After that, we can subscribe to the services method createMobOrder, and put a test directly into its body checking the response we get back from the method.

8. When it comes to sending calls to the Java backend, it returns an Observable. By doing so, the call is completely async, and once the call returns some value, it then proceeds to run its block of code. In our case, that block of code is 1 test.

CREATE MOBORDER METHOD TDD

```
const mockMobOrderEasy = {
  mob1: ClassListEnum.Ranger,
  mob2: ClassListEnum.Fighter,
  mob3: ClassListEnum.None,
  mob4: ClassListEnum.None,
  difficulty: "Easy"
}
it('should send a create mobOrder with difficulty set to easy', () => {
  const mockHttpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  }

  service.createMobOrder("Easy").subscribe(response => {
    expect(response).toBe(mockMobOrderEasy);
  });

  let request = httpClient.expectOne(environment.mobOrderSaveNew);
  expect(request.request.method).toBe('POST');
  expect(request.request.headers.get('Content-Type')).toBe(mockHttpOptions.headers.get('Content-Type'));
  request.flush(mockMobOrderEasy);      You, a few seconds ago • Uncommitted changes
});
```

9. With everything setup, we can now create more conditionals to our test. First we "httpClient.expectOne", this tests to make sure the particular url, saved in environment variables, is actually being called.
10. Once we make a call to the Java backend, we can then test to make sure the method we sent was a "POST", and we can compare the Headers we sent to that of the headers we have mocked to make sure they match.
11. After we have checked the actual request we sent out, we can now return some value back from our "HttpTestingController", this is done with "request.flush(value we get back from the mocked Java backend)". Once we flush a response back, that response well go into our "service.createMobOrder("Easy").subscribe(response => {});", which runs our last test "expect(response).toBe(mockMobOrderEasy);".
12. Once again, the subscribe runs last because it is async, and waits around for a response to come back, in this case it's waiting for a flush.

CREATE MOBORDER METHOD TDD

13. With a test in place, lets create our actual implementation code now. We begin by injecting a "HttpClient" into our service.ts. Make sure its imported from '@angular/common/http' and not selenium.

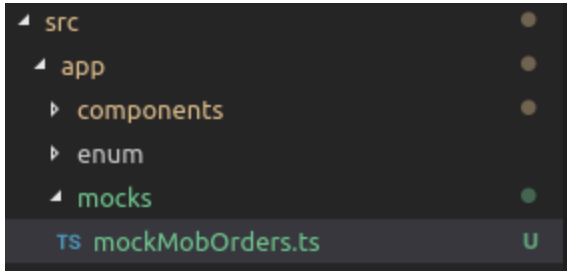
```
export class MobOrderService {

  constructor(private _httpClient: HttpClient) { }
```

14. Once injected through our constructor, we can now create the method that sends a post request, with headers and a body to our specified url. We can also specify it's return type to our specific model "MobOrder" rather than having the type of "any".

```
createMobOrder(difficulty: string): Observable<MobOrder> {
  const httpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  };

  return this._httpClient.post<MobOrder>(environment.mobOrderSaveNew, difficulty, httpOptions);
}     Michael Grammens, a month ago • Added a bunch
```

CREATING MOCKDATA USING EXPORTING

1. Now with everything setup properly, let's create a mocks file so we can have 1 location for all of our mocked data and just import them rather than creating mock data in our test suites or actual code.



2. Inside our mocks file, let's go ahead and create a .ts file and name it "mockMobOrder.ts". Inside of our mocked file, we now need to create some mocked Data, to do so, we simply create variables using "let";"const" would also work, and then export them.

```
export let mockMobOrderEasy: MobOrder = {
    mob1: ClassListEnum.Ranger,
    mob2: ClassListEnum.Fighter,
    mob3: ClassListEnum.None,
    mob4: ClassListEnum.None,
    difficulty: "Easy"
}

export let mockMobOrderMedium: MobOrder = {
    mob1: ClassListEnum.Ranger,
    mob2: ClassListEnum.Fighter,
    mob3: ClassListEnum.None,
    mob4: ClassListEnum.None,
    difficulty: "Easy"
}

export let mockMobOrderHard: MobOrder = {
    mob1: ClassListEnum.Ranger,
    mob2: ClassListEnum.Fighter,
    mob3: ClassListEnum.None,
    mob4: ClassListEnum.None,
    difficulty: "Easy"
}
```

CREATING MOCKDATA USING EXPORTING

3. Now with our actual mock data defined, let's go back to our "mob-order.service.spec.ts" file and refactor our code to include our new "mockMobOrder.ts" file.

```
// const mockMobOrderEasy = {
//   mob1: ClassListEnum.Ranger,
//   mob2: ClassListEnum.Fighter,
//   mob3: ClassListEnum.None,
//   mob4: ClassListEnum.None,
//   difficulty: "Easy"
// }
it('should send a create mobOrder with difficulty set to easy', () => {
  const mockHttpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  }

  service.createMobOrder("Easy").subscribe(response => {
    expect(response).toBe(mockMobOrderEasy);
  });

  let request = httpClient.expectOne(environment.mobOrderSaveNew);
  expect(request.request.method).toBe('POST');
  expect(request.request.headers.get('Content-Type')).toBe(mockHttpOptions.headers.get('Content-Type'));
  request.flush(mockMobOrderEasy);
});
```

4. With our testing suite refactored, we should add one more variable to our "mocks/mockMobOrder.ts" file that's an array of data for future testing or actual use in code.

```
export let mockMobOrderList: MobOrder[] = [];
mockMobOrderList.push(mockMobOrderEasy);
mockMobOrderList.push(mockMobOrderMedium);
mockMobOrderList.push(mockMobOrderHard);
```

SERVICE CHECKPOINT 2

1. Now that we have mock data and a "post" request that we solved using TDD. The goal here is to write the TDD for a "get" request in the same service "mob-order.service.spec.ts" file.

2. Starting with the code provided here, write the rest of the test to check for the url being hit, the "get" method being called, that the correct headers are being included, and the actual body of the response.

```
it('should send a get mobOrder request', () => {
  const mockHttpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  }      You, a few seconds ago • Uncommitted cha
});
```

SERVICE CHECKPOINT 2 SOLUTION

1. For our solution, let's walk step by step through each portion. Step 1, we need to send a fake request out using our service method. Remember, this service is async, so we simply "subscribe" to it, and wait for a response to come back.

```
service.getMobOrder().subscribe(response => {
  expect(response).toBe(mockMobOrderList);
});
```

2. While we wait for our fake response to come back to run the test embedded within the body, we can run some other tests. Now let's make our fake request to the specified url endpoint.

```
let request = httpClient.expectOne(environment.mobOrderController);
```

3. After we send out our fake request, we'll check for the method and compare it's headers. Checking for the requests method first, we end up with.

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

4. Now let's check for 'Content-Type':'application/json'.

```
expect(request.request.headers.get('Content-Type')).toBe(mockHttpOptions.headers.get('Content-Type'));
```

SERVICE CHECKPOINT 2 SOLUTION

5. Now with us checking the url, "get", and the headers, let us send back our response to run our embedded test inside the "subscribe." To do so, we need to include the following code.

```
request.flush(mockMobOrderList);
```

6. With everything accounted for the end solution looks like this.

```
it('should send a get mobOrder request', () => {
  const mockHttpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  }

  service.getMobOrder().subscribe(response => {
    expect(response).toBe(mockMobOrderList);
  });

  let request = httpClient.expectOne(environment.mobOrderController);
  expect(request.request.method).toBe('GET');
  expect(request.request.headers.get('Content-Type')).toBe(mockHttpOptions.headers.get('Content-Type'));
  request.flush(mockMobOrderList);      You, a few seconds ago • Uncommitted changes
});
```

GET MOBORDER METHOD TDD

1. Using our solution from checkpoint 2, we now need to implement the actual code to send a "get" request to a url with some headers.

2. The actual code implementation looks like this.

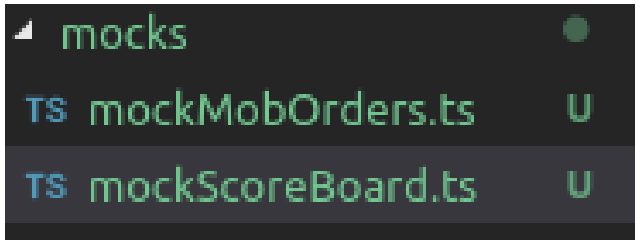
```
getMobOrder(): Observable<MobOrder[]> {
  const httpOptions = {
    headers: new HttpHeaders({      Michael Grammens, a month ago • Added a bunch
      'Content-Type': 'application/json'
    })
  };

  return this._httpClient.get<MobOrder[]>(environment.mobOrderController, httpOptions);
}
```

3. Notice how we can send a request in a single line of code, but to test it, we had to break it up into parts and test each part.

CREATING MOCK DATA FOR SCOREBOARD

1. Our next goal is to create a service for our scoreboard, the scoreboard well also us to keep track of player scores, and give them a way to compare themselves. We can follow our previous examples and begin by setting up a Mock Data file.

2. To setup a Mock Data file, go into our mocks folder, and add a file called "mockScoreBoard.ts".



3. Once the file has been created, let's add some mocked variables to it, including the total list of players to be displayed.

4. Now with our mock data stored, let's go ahead and use this data to test our scoreboard service

```
export let mockScoreBoardOne = {
    playersScore: 951,
    playersName: "Wargods3",
    playersClass: ClassListEnum.Fighter,
    difficulty: DifficultyEnum.Hard
}

export let mockScoreBoardTwo = {
    playersScore: 431,
    playersName: "RaiseTheRoof",
    playersClass: ClassListEnum.Black_Mage,
    difficulty: DifficultyEnum.Easy
}

export let mockScoreBoardThree = {
    playersScore: 689,
    playersName: "KitDaKat",
    playersClass: ClassListEnum.Ranger,
    difficulty: DifficultyEnum.Medium
}

export let mockScoreBoardList: Scoreboard[] = [];
mockScoreBoardList.push(mockScoreBoardOne);
mockScoreBoardList.push(mockScoreBoardTwo);
mockScoreBoardList.push(mockScoreBoardThree);
```

SERVICE CHECKPOINT 3

1. First let's navigate to our score-board.service.spec.ts file.

2. Now change your spec file to look like the image.

```
fdescribe('ScoreBoardService', () => {
  let service: ScoreBoardService;

  beforeEach(() => {TestBed.configureTestingModule({
    })
    service = TestBed.get(ScoreBoardService);       You,
  });

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

  it('should get all the scores ', () => {
    const mockHttpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json'
      })
    };
    //url: environment.scoreBoardController
  });

  it('should add a score to the scoreboard', () => {
    const mockHttpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json'
      })
    };
    //url: environment.scoreBoardController
  });
```

3. Your task is to implement the two tests using the environment variables for the URL, comparing the mock Headers to the actual Headers, checking the method being called, and testing the body of the response.

SERVICE CHECKPOINT 3 SOLUTION

1. The first thing we need to do is to create a httpClient using HttpTestingController, and importing the HttpClientTesting Module. With this httpMock, we only make fake calls to our Java Backend.

```
let service: ScoreBoardService;
let httpMock: HttpTestingController;

beforeEach(() => {TestBed.configureTestingModule({
    imports: [HttpClientTestingModule]
  })
  service = TestBed.get(ScoreBoardService);
  httpMock = TestBed.get(HttpTestingController);
});
```

SERVICE CHECKPOINT 3 SOLUTION

2. From there we can then send a call to our method. When that method is called it waits for a response. We then make a fake call using Http and test its method for "GET"/"POST". Once we check the method, we then check for its headers, more specifically the 'Content-Type' field in our headers. Once we have checked everything, we then flush a body back to our method called to run our final test.

```
it('should get all the scores ', () => {
  const mockHttpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  };

  service.getAllScores().subscribe(response => {
    expect(response).toBe(mockScoreBoardList);
  });

  let request = httpMock.expectOne(environment.scoreBoardController);
  expect(request.request.method).toBe('GET');
  expect(request.request.headers.get('Content-Type')).toBe(mockHttpOptions.headers.get('Content-Type'));
  request.flush(mockScoreBoardList);
});

it('should add a score to the scoreboard', () => {
  const mockHttpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  };

  service.addScore(mockScoreBoardOne).subscribe(response => {
    expect(response).toBe(mockScoreBoardOne);
  });

  let request = httpMock.expectOne(environment.scoreBoardController);
  expect(request.request.method).toBe('POST');
  expect(request.request.headers.get('Content-Type')).toBe(mockHttpOptions.headers.get('Content-Type'));
  request.flush(mockScoreBoardOne);
});
```

SERVICE CHECKPOINT 4

1. With our solution from checkpoint 3, we can now implement our actual methods. Navigate to score-board.service.ts, and implement the following code below. If you run the tests, they fail because the code below is returning an Observable, hence, of({}) with a body of mock data as opposed to making an Http call.. Your task is to change these methods to send a get and a post with a headers 'Content-Type' set to 'JSON' so that the tests we just created pass.

```
export class ScoreBoardService {

  constructor(private _httpClient: HttpClient) { }

  getAllScores() : Observable<Scoreboard[]> {
    return of(mockScoreBoardList);
  }

  addScore(currScore: Scoreboard) : Observable<Scoreboard>{
    return of(mockScoreBoardOne);
  }
```

SERVICE CHECKPOINT 4 SOLUTION

1. Our first step is to setup the headers.

2. Once our headers have been defined, we can then call our injected HttpClient to send a get and post request to the url "environment.scoreBoardController", with our headers. A post request uses a body to send/retrieve information, and therefore we include our new score into the body of our post.

```
export class ScoreBoardService {

  constructor(private _httpClient: HttpClient) { }

  getAllScores() : Observable<Scoreboard[]> {
    const httpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json'
      })
    };

    return this._httpClient.get<Scoreboard[]>(environment.scoreBoardController, httpOptions);
  }

  addScore(currScore: Scoreboard) : Observable<Scoreboard> {      Michael Grammens, a month ago • Added
    const httpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json'
      })
    };

    return this._httpClient.post<Scoreboard>(environment.scoreBoardController, currScore, httpOptions);
  }
}
```

3. With our code now implemented we can run our tests again to watch them pass.

CONTRIBUTORS

Created By: Michael Grammens

Contributions By:

BronwenFerry Hughes

Devyn Coyer

Patrick Moriarty

Alan Johnson

THANK YOU FOR READING. HOPE THE INFORMATION IS HELPFUL.