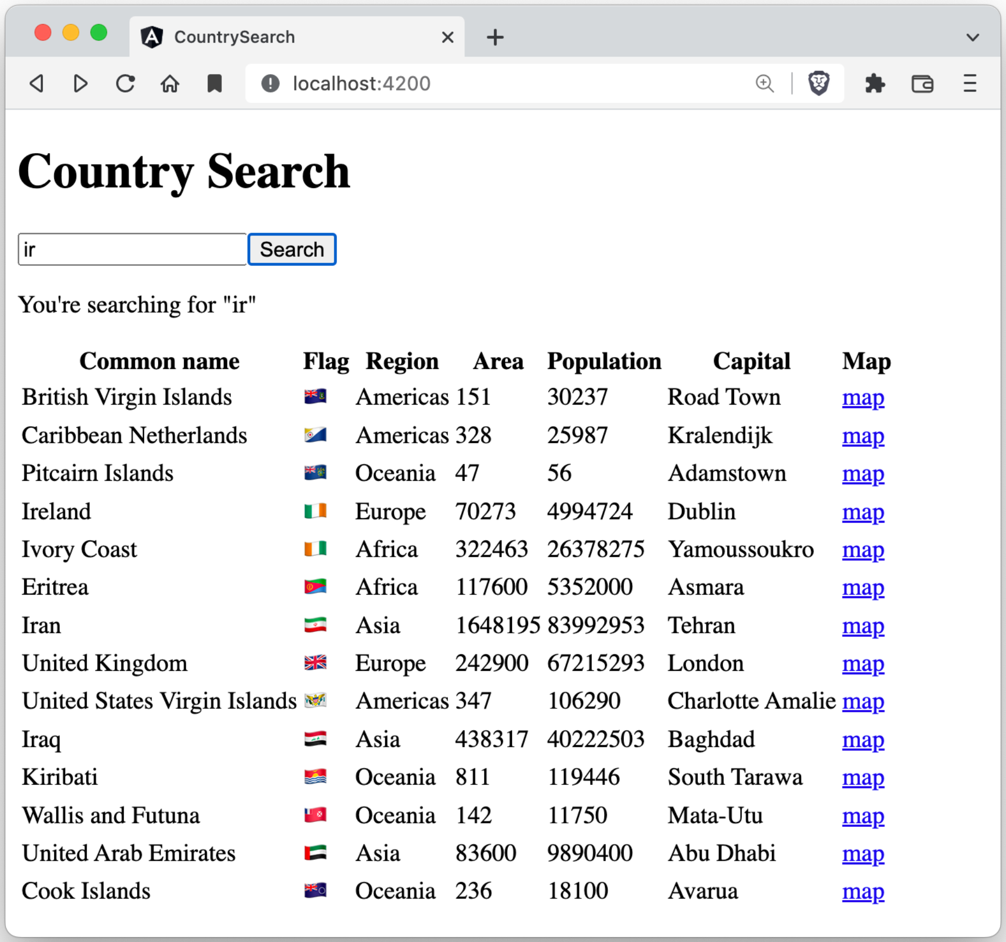
Countries Assessment

This assessment is designed to be written in three hours or less by an individual developer. It should take no special knowledge other than the things you've learned in this course and your prerequisite knowledge of HTML, CSS, and JavaScript.

You are free to refer to your notes, the class materials. You are free to use the Internet to look things up but not to collaborate with another developer.

In this final assessment, your mission will be to create a new Angular web app from scratch that will allow high school students to look up country data. They'll enter a country name or partial country name and your app will show them countries that match their search criteria. It'll look kind of like this:



Country Search

You're searching for "ir"

Common name	Flag	Region	Area	Population	Capital	Map
British Virgin Islands		Americas	151	30237	Road Town	map
Caribbean Netherlands		Americas	328	25987	Kralendijk	map
Pitcairn Islands		Oceania	47	56	Adamstown	map
Ireland		Europe	70273	4994724	Dublin	map
Ivory Coast		Africa	322463	26378275	Yamoussoukro	map
Eritrea		Africa	117600	5352000	Asmara	map
Iran		Asia	1648195	83992953	Tehran	map
United Kingdom		Europe	242900	67215293	London	map
United States Virgin Islands		Americas	347	106290	Charlotte Amalie	map
Iraq		Asia	438317	40222503	Baghdad	map
Kiribati		Oceania	811	119446	South Tarawa	map
Wallis and Futuna		Oceania	142	11750	Mata-Utu	map
United Arab Emirates		Asia	83600	9890400	Abu Dhabi	map
Cook Islands		Oceania	236	18100	Avarua	map

Creating the app and search form

1. Create a new Angular project called country-search. Choose "no" for routing and "CSS" for styles.
2. Create a new component called CountrySearch.
3. Edit AppComponent. Remove the boilerplate. Instead, show your new CountrySearch component.
4. Run your project's auto-created unit tests by running *ng test*. Expect some of the AppComponent tests to fail. Delete any tests from app.component.spec.ts that no longer make sense. We just want you to start with 100% passing tests.

Now let's do a little TDD.

Unit testing your component

Remember, TDD starts with you writing a failing unit test. Then you write code until the test passes. Let's write that first unit test.

5. Edit country-search.component.spec.ts. Write a unit test for 'displays the search criteria to the user'. It will make sure that whatever the user puts in the input box is echoed back out to them on the page. This unit test should ...
   - Locate the input box by its id 'searchString'.
   - Assert that it is not null.
   - Put a value in the input box – any value you like.
   - Locate the #feedback paragraph
   - Assert that it exists.
   - Assert that its textContent contains the value you entered in the textbox.
6. Make the test pass by adding a <form>, <input> and <p> to the HTML file and adding a searchString property to the TypeScript file.
   Hints:
   - You'll need to imports FormsModule/ReactiveFormsModule in app.module.ts,
   - You'll need to imports FormsModule/ReactiveFormsModule in country-search.component.spec.ts in the TestBed.configureTestingModule()
   - Remember about someNativeElement.dispatchEvent() and fixture.detectChanges()

Got it passing? Cool.

Next we want to make an Ajax GET request from the service at http://RESTCountries.com. We could make our Ajax call in the component class, but that's not a very clean solution. Instead we're going to ask you to create a *service*.

Creating the CountryService

7. Generate a new service called CountryService.
8. Re-run your unit tests again. You should see an automatically-created smoke test for this CountryService.
9. Edit country.service.spec.ts. Make its TestBed config look like this:

```
TestBed.configureTestingModule({
  imports: [HttpClientModule]    // Needed to make HTTP calls in the unit test
});
```

10. Write a new failing unit test. Here's a start for you. Copy this into country.service.spec.ts.

```
it('should retrieve countries', (done) => {
  service.getCountries("Tuvalu")
    .subscribe({
      next: (countries) => {
        // TODO: Assert that the length of the countries array is 1
        const [country] = countries;
        const { area, name: { common }, region, subregion } = country;
        // TODO: Assert that the area is 26
        // TODO: Assert that the common is "Tuvalu"
        // TODO: Assert that the region is "Oceania"
        // TODO: Assert that the subregion is "Polynesia"
        done();
      }
    });
});
```

11. In that code above, there are TODOs. Follow their instructions.

Got those lines written? Good. You have yourself a failing test.

12. Now write the code in country.service.ts to make it pass.
    Hints:
    - Don't forget to imports HttpClientModule into the app.module.ts
    - You'll need to inject and HttpClient into the constructor of your CountryService.
    - Make sure your service returns an Observable.

Once your test is passing, you can move on.

Using your new service in the component

13. Edit your country-search.component.ts file. Use dependency injection to get an instance of your CountryService and call it _countryService.
14. Behind the <button>'s click event or the <form>'s submit event, call this._countryService.getCountries(searchString).
15. Subscribe to the observable returned from the service to retrieve all the country data.
16. If there's an error, it should be console.error()ed.
17. For each country returned from the service, put a row in the table. Use the above screenshot for a guideline.

Zooming into the flag image

18. The flag may be hard to see. So when the user hovers over it with their mouse, increase the size of the flag using a CSS transform property. Make the flag at least three times its current size.
19. We want the increase of size to happen gradually over one second. (hint: use a transition)
20. If you mouse over the flags, they're zooming in and out. It may look too busy. So delay the transition for one quarter second before increasing its size.

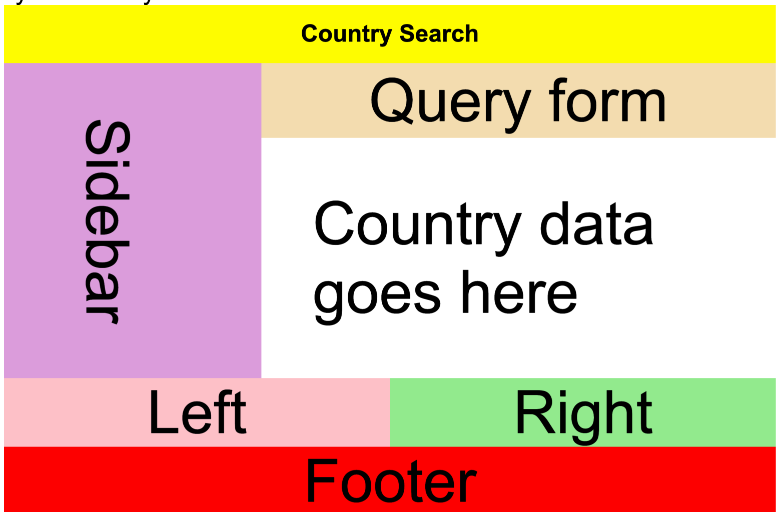
Page layout with a grid

If you haven't already done so, it's probably time to lay out the page. We're going to do that with a grid.

21. Make sure you have a <main> which wraps everything else. This will be your grid container.

22. The grid items will be
    - The <section> with the data table.
    - The <section> with the query form.
    - A new sidebar <section> (placeholder section with a purple background)
    - A page header (yellow background)
    - A page footer (placeholder section with a red background)
    - And two sections for ... other things. They're just above the footer and should take about half the window each. They're pink and light green below.

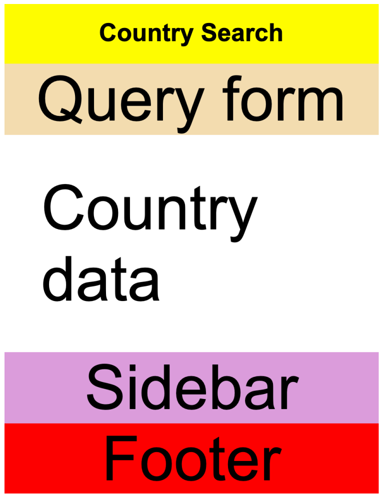
    Here's how they should lay out.



Responsive design

We have way too much data to show on a phone. Let's make the component responsive.

23. First, let's change the layout itself. If you're on a viewport that is smaller than 600 pixels, Make the layout look like this:



    Specifically,
    - The sections we're calling Left and Right are not displayed
    - Sidebar goes below country data
    - All sections take up the entire width

Once you've got the layout changed, we should probably do something similar with the country table itself. There's just too much stuff to look good on a small screen.

24. If the viewport is more than 600px show the entire table. But if the viewport is less than that, don't display the region or capital columns.

25. Bonus!! If you have extra time after finishing the last section, come back to this and adjust the widths and columns shown to tune your look and feel.

Mocking the service

Let's do one last unit test.

26. In country-search.component.spec.ts, write a test to make sure that if the user enters "Tuvalu" in the search input box and then clicks the button, the region "Oceania" is displayed on the page. Here's one way to make that work:

```
it('displays the country fetched', waitForAsync(() => {
  //Arrange
  const inputBox = fixture.nativeElement.querySelector('#searchString')
  const button = fixture.nativeElement.querySelector('button');
  const table = fixture.nativeElement.querySelector('table')
  inputBox.value = "Tuvalu";
  //Act
  inputBox.dispatchEvent(new Event('input'));
  fixture.detectChanges();
  button.dispatchEvent(new Event('click'));
  fixture.whenStable().then(() => {
    fixture.detectChanges();
    //Assert
    expect(table.innerHTML).toMatch(/Oceania/);
  })
}));
```

27. Get your test to pass.

Here's a problem, though. If Tuvalu ever changes their region, your test will suddenly start failing and you'll have no reason why. The poor dev who checks their code and sees this test breaking will think their code caused the problem! Your code is dependent on geopolitics!

We need to solve this problem with a mock.

28. Write a mock service that always returns a given region like "Oceania" given "Tuvalu" in the textbox. Heck, use a fake country with a fake region if you like.
29. In your unit test, where you had been providing an instance of the real service, now provide an instance of the mock service.
30. Adjust until your unit test passes again.

If you get to this point, congratulations! You've finished the assessment.