**Resume / About The Candidate Questions**

**10 minutes**

* Tell us about your current role?
* What's your current project?
* What’s the type of role you're looking for?
* What's the level of role you're looking for?
* Why do you want to join a giant company like Charter?
* Angular 2 / Typescript experience?
* If TS exp, what are some pros / cons of TS?

**JavaScript Questions**

**20 minutes**

### What is scope?

### Prior to es2016 we had var keyword

Var usually binds to ‘this’ property so in it refers to the current parent or function

Similarly inside a function if we use var to declare a variable it will be local to that function and can be used anywhere in any block statement of that function. if we forget or avoid to use var keyword inside function it will be declared as a global variable upon execution of the function like a global intermediate. Also if we declare outside the function like an orphan statement it will be a global scope as well or we can use window.variable name to create global.

Now in that if we ever had to create a lexical or block level scope we would use IFFE which is immediate invoke function expression to wrap that block inside an function so it can have its own scope and gets destroyed as if would invoke itself.

In es6 they have added let and const. Let is for. Lexical scope and avoid using iffy, Const is for immutability for primitive data types such as string, array, object literal.

### function

### We have function declaration

function myFunction(a, b) {

return a \* b;

}

function assignment

var x = function (a, b) {return a \* b};

var z = x(4, 3);

1. How does scope work in JS?
2. What are the implications of a variable defined with var vs without?
3. What are the implications of a variable defined with let vs var?
4. What is hoisting?

Javascript hoist variables and function and what I mean by the is that once the function is about to execute the fist thing javascript compile is the variable reference, so it will move all the declarations to the top of the scope that is the function declaration or the global scope.

var a=30;

function abc(){

console.log(a); // undefined beucase a is local and it will be hoisted to the top of the scope of the function abc();

var a=10; // 10 beucase even thos the declaration is hoited the assignment still takes place where its originally declared

console.log(a);

}

abc();

## How does hoisting work with declarations vs initializations?

## If you want to do this as a code question:

var text = 'outside';

function logIt() {

console.log(text);

var text = 'inside';

};

logIt();

## What is a Closure? IIFE?

**Closure:**

**closure is an inner function that has access to the outer function variables scope chain.**

**The closure has three scope chains:**

* **it has access to its own scope**
* **it has access to the outer function variables.**
* **it has access to the global variables.**

**The inner function has access not only to the outer function’s variables, but also to the outer function parameters. The inner function cannot call the outer function arguments object, however, even though it can call the outer function’s parameters directly.**

**closure example**

**function makeCounter() {**

**let count = 0;**

**return function() {**

**return count++;**

**};**

**}**

**let counter1 = makeCounter();**

**let counter2 = makeCounter();**

**alert( counter1() ); // 0**

**alert( counter1() ); // 1**

**alert( counter2() ); // 0 (independent)**

**A closure is one way of supporting first-class functions; it is an expression that can reference variables within its scope (when it was first declared), be assigned to a variable, be passed as an argument to a function, or be returned as a function result.**

**Among other things, closures are commonly used to give objects data privacy. Data privacy is an essential property that helps us program to an interface, not an implementation. This is an important concept that helps us build more robust software because implementation details are more likely to change in breaking ways than interface contracts.**

**IFFE:**

**This pattern is often used when trying to avoid polluting the global namespace, because all the variables used inside the IIFE (like in any other *normal* function) are not visible outside its scope.**

**(function(){  
 // all your code here  
 var foo = function() {};  
 window.onload = foo;  
 // ...  
})();  
// foo is unreachable here (it’s undefined)**

## What is == vs ===?

**The identity (===) operator behaves identically to the equality (==) operator except no type conversion is done, and the types must be the same to be considered equal.**

**The == operator is checking the values of the two objects and returning true, but the === is seeing that they're not the same type and returning false. Which one is correct? That really depends on what you're trying to compare**.

## Can you explain object coercion?

**When it comes to objects and engine encounters expression like [1] + [2,3], first it needs to convert an object to a primitive value, which is then converted to the final type. And still there are only three types of conversion: numeric, string and boolean.**

**The simplest case is boolean conversion: any non-primitive value is always coerced to true, no matter if an object or an array is empty or not. Objects are converted to primitives via the internal [[ToPrimitive]] method, which is responsible for both numeric and string conversion. In general the cohesion algorithm is as follows:**

* **If input is already a primitive, do nothing and return it.**
* **Call input.toString(), if the result is primitive, return it.**
* **Call input.valueOf(), if the result is primitive, return it.**
* **If neither input.toString() nor input.valueOf() yields primitive, throw**

## What are the differences between null vs undefined?

**In JavaScript, undefined means a variable has been declared but has not yet been assigned a value, such as:**

**var TestVar;**

**console.log(TestVar); //shows undefined**

**console.log(typeof TestVar); //shows undefined**

**null is an assignment value. It can be assigned to a variable as a representation of no value:**

**var TestVar = null;**

**console.log(TestVar); //shows null**

**console.log(typeof TestVar); //shows object**

**undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object.**

**null === undefined // false**

**null == undefined // true**

**null === null // true**

**and**

**null = 'value' // ReferenceError**

**undefined = 'value' // 'value'**

## How do those two values differ from the concept of not defined?

**‘undefined’ and ‘null’ are javascript primitive types like Number, String, etc.**

**not defined doesn’t exist as a type or value in javascript.**

## In JS, are parameters passed by value or by reference?

* primitive types: passed by value
* object passed by reference

## What's the difference between a promise and observables?

**A Promise handles a single event when an async operation completes or fails.**

**An Observable is like a Stream (in many languages) and allows to pass zero or more events where the callback is called for each event. Often Observable is preferred over Promise because it provides the features of Promise and more. With Observable it doesn't matter if you want to handle 0, 1, or multiple events. You can utilize the same API in each case. Observable also has the advantage over Promise to be cancelable. If the result of an HTTP request to a server or some other expensive async operation isn't needed anymore, the Subscription of an Observable allows to cancel the subscription, while a Promise will eventually call the success or failed callback even when you don't need the notification or the result it provides anymore.**

**Promises:**

**returns a single value**

**not cancellable**

**Observables:**

**works with multiple values over time**

**an array whose items arrive asynchronously over time**

**cancellable**

**supports map, filter, reduce and similar operators**

**proposed feature for ES 2016**

**use Reactive Extensions (RxJS)**

**CSS Questions**

## What is the difference between display:none, visibility:hidden?

**display:none removes the element from the normal flow of the page, allowing other elements to fill in.**

**visibility:hidden leaves the element in the normal flow of the page such that is still occupies space.**

## How do those differ from \*ngIf = false;

## Ng-if, like most built-in directives relies upon evaluating an expression as true or false. In the case of ng-if, if the expression evaluates false, it removes the element from the DOM entirely (as in, the node is gone). If it’s true, the element gets recreated.

**Put more simply, ng-if allows you to display or remove elements based on scope (or model) values. Even more simply: If a value is false in your model, ng-if will remove the appropriate element from the DOM (and vice versa).**

## What is flexbox?

So far, we’ve only had fixed-width items. But what if we want them to be responsive? To achieve that we have a property called flex. It makes it a lot easier than the old way of using percentages to improve the items align, directions and order in the container even when they are with dynamic or even unknown size. The two main components of a Flexbox layout are the **container** and the **items**. The prime characteristic of the flex container is the ability to modify the width or height of its children to fill the available space in the best possible way on different screen sizes.

We’ll simply target all the items and give them a flex value of 1.

container > div {

flex: 1;

}

values you can set for justify-content:

* flex-start (**default**)
* flex-end
* center
* space-between
* space-around
* space-evenly

## What are some pros and cons of some features of it?

For example;

there is this inline-block grid problem that people get real puzzled when it happens with all of the grid systems. It's to say that when you have a grid layout, if one of the row items outgrows in height the others in the same row, the next row won't get started from the leftmost but will get stuck at the outgrown previous row item's position. This is a terrible situation in responsive designs and i have seen many so called "responsive" wordpress themes prone to this failure. Without utilizing Flexbox this problem can only be fixed with stupid meaningless CSS hacks. **With flexbox it's a matter of the past. Handling such layout issues is a breeze**

**Whiteboard Questions**

**20 - 40 minutes**

* Run through the architecture of current project or of a side project?
* Or
* You have a vehicle API that returns a JSON stringified response object with strings describing vehicles. We have the business logic in place to understand how to map these vehicles back to requirements.
* We have 2 views that we want to display this information in, but we’d like them split up into buckets- one for cars, one for trucks, and one for SUVs.
* These views are separate states in our single page application. We can expect the buckets to display a little differently on each view, but to contain the same information. The information is not time-sensitive.
* How, using Angular patterns, can we structure our app best, all while ensuring that the vehicle API GET request is only made once?
* If they need hints, we can talk about the things we’ll have to do: GET request, mapping, modeling, getting it into our views somehow. Where do you do each of these things? Why are they doing it here, instead of there? Etc.
* Poor Icon Component Design Question

Question setup

<ul>

<li \*ngFor="let appt of appts">

<appt \*ngIf="!appt.warning"><spc-icon ico="check"></spc-icon>{{appt.type}} {{appt.date}} - {{appt.time}} </appt>

<appt \*ngIf="appt.warning"><spc-icon ico="alert" dark></spc-icon>{{appt.type}} {{appt.date}} - {{appt.time}}</appt>

</li>

</ul>

appts:[

{

warning: false,

date: '07/10/2017',

time: '8 AM - 9 AM PDT',

type: 'Installation'

},

{

warning: true,

date: '07/13/2017'

time: '9 AM - 10 AM PDT',

type: 'Service'

}

]

## Answer:

<ul> <li \*ngFor="let appt of appts"></li>

</ul>

<ng-template #isWarning>

<div>

<appt>

<spc-icon ico="check"></spc-icon>

{{appt.type}}

{{appt.date}}

{{appt.time}}

</appt>

</div>

</ng-template>

<ng-template #noWarning>

<div>

<appt \*ngIf="appt.warning">

<spc-icon ico="alert" dark></spc-icon>

{{appt.type}}

{{appt.date}}-

{{appt.time}}

</appt>

</div>

</ng-template>

< /li>

</ul >

**Question overview**

* A service provides the appts array to some code-behind somewhere which is mapped to the appts variable in the HTML and creates the screenshot that is above.
* We're looking to refactor the above HTML, what stands out to you as things that could be altered in order to make this code more readable, maintainable, and testable?
* Feel free to be adjusting the appts objects, adding functions to the code-behind, or really whatever you need to do to make the refactoring successful

Hints

* 1st hint: Look at the markup, and the results, what is actually changing?
* 2nd hint: Do you need 2 <appt></appt> component instances? So, what seems to be changing what is displayed?  (They might say classes, but make sure they seem them as directives)
* How do we make a property calculable?  (bound dynamically)
* Super Bonus points: if they get this far, lead them down the road to how to hide an attribute.  (return null as in this example:  [[foo](https://chalk.charter.com/pages/createpage.action?spaceKey=XGEXPGROUP&title=foo&linkCreation=true&fromPageId=193588099)]="someProp" . Where prop would be a function or property that returns null.)

**Computer Science Pattern Questions**

**20 minutes**

## What is functional vs imperative programming?

## Functional programming

### it is an awesome tool to have in your arsenal, since it will at least. An imperative approach (HOW): I see that table located under the Gone Fishin’ sign is empty. My husband and I are going to walk over there and sit down. A declarative approach (WHAT): Table for two, please.

### Highlights of functional or declarative approach

### Reduce the amount of *mutations*;

### Make the code base easier to test;

### Composition and changes are made easier;

### You can apply it right now in your code base since it’s just a programming paradigm (a set of rules).

**Declarative programming**

* it is “the act of programming in languages that conform to the mental model of the developer rather than the operational model of the machine”.
* Declarative Programming is programming with declarations, i.e. declarative sentences. The declarative property is where there can exist only one possible set of statements that can express each specific modular semantic.The imperative property is the dual, where semantics are inconsistent under composition and/or can be expressed with variations of sets of statements.
* Declarative languages contrast with imperative languages which **specify explicit manipulation of the computer’s internal state**; or procedural languages which specify an explicit sequence of steps to follow.

In computer science, declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow.

### 

### What parts of SOLID object oriented design principles do you know?

<https://hackernoon.com/solid-principles-made-easy-67b1246bcdf>

**S — Single Responsibility Principle(S.R.P)**

*A class should have one, and only one, reason to change.*

***O — Open-Closed Principle***

*You should be able to extend a classes behavior, without modifying it.*

***L — Liskov Substitution Principle***

*Derived classes must be substitutable for their base classes.*

***I — Interface Segregation Principle***

*Make fine grained interfaces that are client specific.*

***D — Dependency Inversion Principle***

*Depend on abstractions, not on concretions.*

### 

### Abstract class vs interface?

Abstract classes are used for **Modelling** a class hierarchy of similar looking classes

(For example Animal can be abstract class and Human , Lion, Tiger can be concrete derived classes)

AND

Interface is used for **Communication** between 2 similar / non similar classes which does not care about type of the class implementing Interface(e.g. Height can be interface property and it can be implemented by Human , Building , Tree. It does not matter if you can eat , you can swim you can die or anything.. it matters only a thing that you need to have Height (implementation in you class) ).

### What are generics and why are they useful?

### A generic program is one that the programmer writes once, but which works over many different data types.

### Broadly speaking, generic programming aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined data types. A generic function such as a pretty printer or a parser is written once and for all times; its specialization to different instances of data types happens without further effort from the user. This way generic programming greatly simplifies the construction and maintenance of software systems as it automatically adapts functions to changes in the representation of data.

### generic programming appears to add an extra level of complication and abstraction to programming. However, I claim that generic programming is in many cases actually simpler than conventional programming. The fundamental reason is that genericity gives you "a lot of things for free"

## Dependency Injection?

Basically, instead of having your objects creating a dependency or asking factory object to make one for them, you pass the needed dependencies in to the object externally, and you make it somebody else's problem. This "someone" is either an object further up the dependency graph, or a dependency injector (framework) that builds the dependency graph. A dependency as I'm using it here is any other object the current object needs to hold a reference to. One of the major advantages of dependency injection is that it can make testing lots easier.

### DI as it works in Angular?

A *service* is just a class in Angular until you register it with an Angular dependency injector.An Angular injector is responsible for creating service instances and injecting them into classes. Angular creates injectors for you as it executes the app, starting with the *root injector* that it creates during the [bootstrap process](https://angular.io/guide/bootstrapping). You do have to register *providers* with an injector before the injector can create that service.

**Providers** tell the injector *how to create the service*. Without a provider, the injector would not know that it is responsible for injecting the service nor be able to create the service.

### *@Component* providers

**A component's providers** (@Component.providers) are registered with each component instance's own injector. Angular can only inject the corresponding services in that component instance or one of its descendant component instances. Angular cannot inject the same service instance anywhere else. Each new instance of the component gets its own instance of the service and, when the component instance is destroyed, so is that service instance.

import { [Component](https://angular.io/api/core/Component) } from '@angular/core';  
 import { DataService } from './data.service';  
  
 @[Component](https://angular.io/api/core/Component)({  
 selector: 'app-data’',  
 providers: [ Data ],  
 template: `  
 <h2>Data</h2>  
 <app-data-list></app-data-list>  
 `  
 })  
 export class DataComponent { }

### 

### *@NgModule* providers

**Angular module providers** (@[NgModule.providers](https://angular.io/api/core/NgModule#providers)) are registered with the application's root injector. Angular can inject the corresponding services in any class it creates. Once created, a service instance lives for the life of the app and Angular injects this one service instance in every class that needs it. The root AppModule registers two providers in its providers array.

providers: [  
 DataService,  
 { provide: APP\_CONFIG, useValue: DATA\_DI\_CONFIG }  
],

The first entry registers the DataService class under the DataService *injection token*. The second registers a value (DATA\_DI\_CONFIG) under the APP\_CONFIG *injection token*.

### Singleton services

Services are singletons *within the scope of an injector*. There is at most one instance of a service in a given injector.

### @Injectable()

The [**@Injectable()**](https://angular.io/api/core/Injectable) decorator identifies a service class that *might* require injected dependencies.

### 

### Providers

A service provider *provides* the concrete, runtime version of a dependency value. The injector relies on **providers** to create instances of the services that the injector injects into components, directives, pipes, and other services.

You must register a service *provider* with an injector, or it won't know how to create the service.

### DI as it relates to testing?

### Services with dependencies

Services often depend on other services that Angular injects into the constructor. In many cases, it easy to create and *inject* these dependencies by hand while calling the service's constructor.

Angular application relies on [dependency injection (DI)](https://angular.io/guide/dependency-injection) to create services. When a service has a dependent service, DI finds or creates that dependent service. And if that dependent service has its own dependencies, DI finds-or-creates them as well.

As service *consumer*, you don't worry about any of this. You don't worry about the order of constructor arguments or how they're created.

As a service *tester*, you must at least think about the first level of service dependencies but you *can* let Angular DI do the service creation and deal with constructor argument order when you use the [TestBed](https://angular.io/api/core/testing/TestBed) testing utility to provide and create services.

#### Mocking as it relates to testing?

A *component-under-test* doesn't have to be injected with real services. In f act, it is usually better if they are test doubles (stubs, fakes, spies, or mocks). The purpose of the spec is to test the component, not the service, and real services can be trouble.

Injecting the real Service could be a nightmare. The real service might ask the user for login credentials and attempt to reach an authentication server. These behaviors can be hard to intercept. It is far easier and safer to create and register a test double in place of the real Service.

#### Testing with a spy

When testing a component, only the service's public API should matter. In general, tests themselves should not make calls to remote servers.

The spy should designed such that any call to Service receives an Observable with a test data. Unlike the real getData() method,

this spy bypasses the server and returns a synchronous Observable whose value is available immediately.

You can write many useful tests with this spy, even though its Observable is synchronous.

## MVC vs MVVM vs MV\*?

**MVC** (Model View Controller)

* **Models:** Models contain data information. Does not call or use Controller and View. Contains the business logic and ways to represent data. Some of this data, in some form, may be displayed in the view. It can also contain logic to retrieve the data from some source.
* **Controller:** Acts as the connection between view and model. View calls Controller and Controller calls the model. It basically informs the model and/or the view to change as appropriate.
* **View:** Deals with UI part. Interacts with the user.

**MVVM**

* **Model**: This represents the data model that your app consumes. For example, in a picture sharing app, this layer might represent the set of pictures available on a device and the API used to read and write to the picture library.
* **View**: An app typically is composed of multiple pages of UI. Each page shown to the user is a view in MVVM terminology. The view is the XAML code used to define and style what the user sees. The data from the model is displayed to the user, and it’s the job of the ViewModel to feed the UI this data based on the current state of the app. .
* **ViewModel**:
  + It is the representation of the state of the view.
  + It holds the data that’s displayed in the view.
  + Responds to view events, aka presentation logic.
  + Calls other functionalities for business logic processing.
  + Never directly asks the view to display anything.

**MV\***

MV\* is Model-View-Whatever architecture. The term is being used in AngularJS, where the Model represents (Data) any variable from javascript or something, View represents HTML side, and \* i.e whatever represents whatever that binds these two together.

## Test-driven development (TDD)

"Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be succinctly described by the following set of rules:

* write a "single" unit test describing an aspect of the program
* run the test, which should fail because the program lacks that feature
* write "just enough" code, the simplest possible, to make the test pass
* "refactor" the code until it conforms to the simplicity criteria
* repeat, "accumulating" unit tests over time

Expected Benefits

* many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort.
* the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases
* although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling

Common Pitfalls

Typical individual mistakes include:

* forgetting to run tests frequently
* writing too many tests at once
* writing tests that are too large or coarse-grained
* writing overly trivial tests, for instance omitting assertions
* writing tests for trivial code, for instance accessors

Typical team pitfalls include:

* partial adoption - only a few developers on the team use TDD
* poor maintenance of the test suite - most commonly leading to a test suite with a prohibitively long running time
* abandoned test suite (i.e. seldom or never run) - sometimes as a result of poor maintenance, sometimes as a result of team turnover

## Lazy Loading

Lazy loading is the process in taking already “code split” chunks of our application, and simply loading them on demand. With Angular, the router is what allows us to lazy load. We call it “lazy” because it’s not “eagerly” loading - which would mean loading assets upfront. Lazy loading helps boost performance - as we’re only downloading a fraction of our app’s bundle instead of the entire bundle. Instead, we can code split per @NgModule with Angular, and we can serve them lazily via the router. Only when a specific route is matched, Angular’s router will load the code split module.