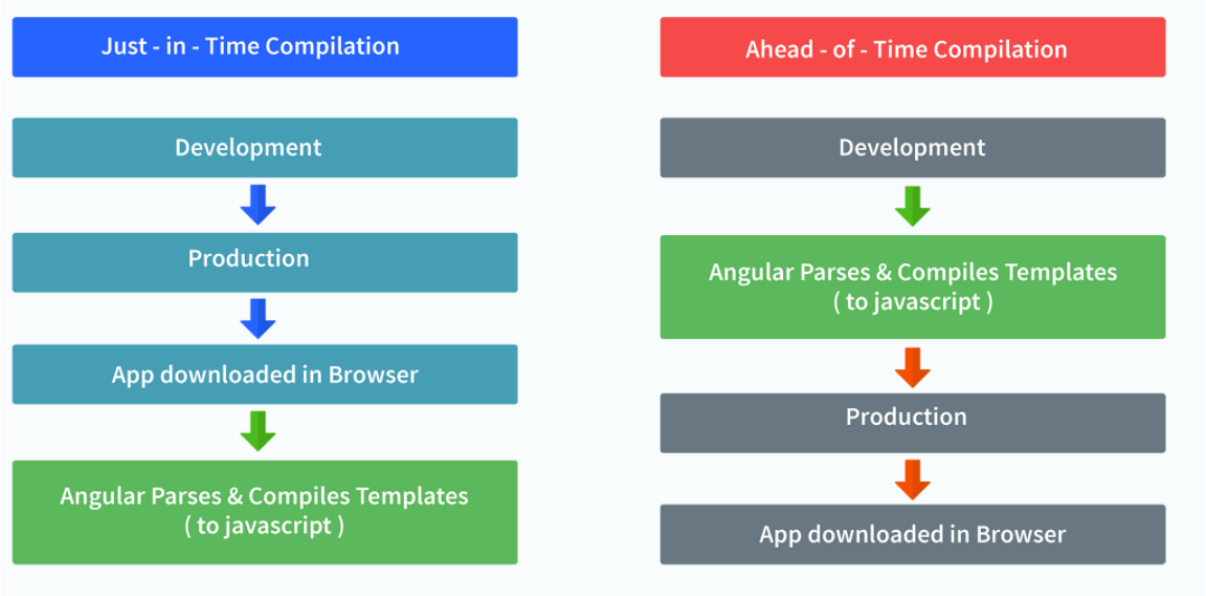
https://www.interviewbit.com/angular-interview-questions/

Mi az AOT-compilation? Melyek az AOT előnyei?

Minden Angular alkalmazás olyan komponensekből és sablonokból áll, amelyeket a böngésző nem ért. Ezért az összes Angular alkalmazást először le kell fordítani, mielőtt a böngészőben elindulna.



A JIT-fordítás során az alkalmazás futás közben fordít a böngészőben.

Míg az AOT-fordításban az alkalmazás a production idő alatt fordít.

Az AOT- fordítás előnyei:

- Mivel az alkalmazás a böngészőben történő futtatás előtt lefordult, a böngésző betölti a futtatható kódot, és azonnal megjeleníti az alkalmazást, ami gyorsabb rendereléshez vezet.
- Az AOT-fordítás során a fordító elküldi a külső HTML- és CSS-fájlokat az alkalmazással együtt, kiküszöbölve a különálló AJAX-kéréseket ezekhez a forrásfájlokhoz, ami kevesebb ajax-kérelmet eredményez.
- A fejlesztők észlelni és kezelni tudják a hibákat a build szakaszban, ami segít a hibák minimalizálásában.
- Az AOT fordító HTML-t és sablonokat ad hozzá a JS fájlokhoz, mielőtt azok a böngészőben futnának. Emiatt nincsenek extra HTML fájlok, amelyeket el kell olvasni, amelyek nagyobb biztonságot nyújtanak az alkalmazás számára.

Alapértelmezés szerint az Angular a JIT fordító segítségével fordít:

```
ng build
ng serve
```

AOT fordítás:

```
ng build --aot
ng serve --aot
```

Miben különböznek az Observables a Promise-tól?

Az első különbség az, hogy egy Observable lazy a Promise pedig eagle.

Promise	Observable
Egyetlen értéket bocsát ki	Egy adott idő alatt több értéket bocsát ki
Nem Lusta	Lusta. Observable-t addig nem hívnak meg, amíg nem vagyunk feliratkozva az observable-re
Nem vonható vissza (cancelled)	A unsubscribe () módszerrel törölhető
	olyan operátorokat nyújt, mint a map, forEach, filter, reduce, retry, retryWhen stb.

Observable

```
const observable = rxjs.Observable.create(observer => {
    console.log('Text inside an observable');
    observer.next('Hello world!');
    observer.complete();
  });

  console.log('Before subscribing an Observable');

  observable.subscribe((message)=> console.log(message));
```

Eredmény:

```
Before subscribing an Observable
Text inside an observable
Hello world!
```

Promise:

```
  const promise = new Promise((resolve, reject) => {
    console.log('Text inside promise');
    resolve('Hello world!');
  });

  console.log('Before calling then method on Promise');

  greetingPoster.then(message => console.log(message));
```

Eredmény:

```
Text inside promise
Before calling then method on Promise
Hello world!
```

Amint láthatja, a Promise belsejében lévő üzenet jelenik meg először. Ez azt jelenti, hogy egy Promise fut, mielőtt a then metódust meghívják.

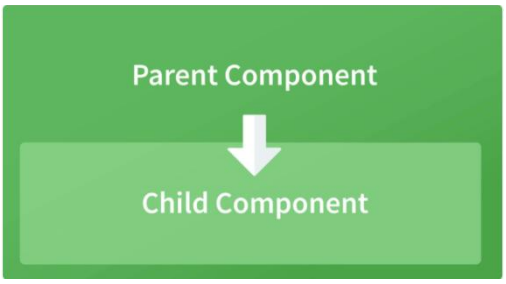
A következő különbség az, hogy a Promise-ok mindig aszinkronak. Akkor is, ha a Promise azonnal resolved. Míg egy Observable, szinkron és aszinkron egyaránt lehet.

```
const observable = rxjs.Observable.create(observer => {
  setTimeout(()=>{
      observer.next('Hello world');
      observer.complete();
  },3000)
});

console.log('Before calling subscribe on an Observable');

observable.subscribe((data)=> console.log(data));

console.log('After calling subscribe on an Observable');
```

Eredmény:

```
Before calling subscribe on an Observable
After calling subscribe on an Observable
Hello world!
```

Hogyan lehet megosztani az adatokat az Angular komponensei között?



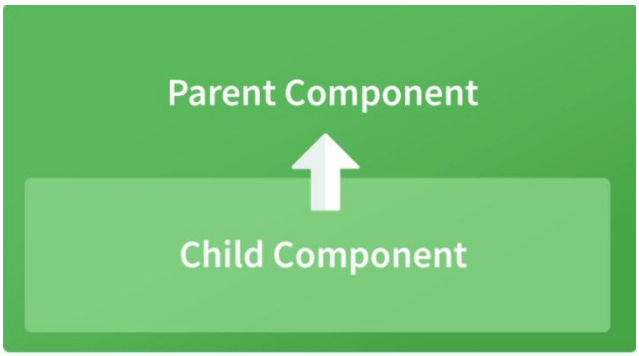
Parent to child using @Input decorator

Consider the following parent component:

```
@Component({
  selector: 'app-parent',
  template: `
    <app-child [data]=data></app-child>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent{
  data:string = "Message from parent";
  constructor() { }
}
```

In the above parent component, we are passing “data” property to the following child component:

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

@Component({
  selector: 'app-child',
  template:`
    <p>{{data}}</p>
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Input() data:string
  constructor() { }
}
```



Child to parent using @ViewChild decorator

Child component:

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

@Component({
  selector: 'app-child',
  template:`
    <p>{{data}}</p>
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  data:string = "Message from child to parent";
  constructor() { }
}
```

Parent Component

```
import { Component,ViewChild, AfterViewInit} from '@angular/core';
import { ChildComponent } from './../child/child.component';

@Component({
  selector: 'app-parent',
  template: `
    <p>{{dataFromChild}}</p>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent implements AfterViewInit {
```

```
  dataFromChild: string;
  @ViewChild(ChildComponent,{static:false}) child;

  ngAfterViewInit(){
    this.dataFromChild = this.child.data;
  }
  constructor() { }
}
```

Child to parent using @Output and EventEmitter

Child Component:

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

@Component({
  selector: 'app-child',
  template:`
    <button (click)="emitData()">Click to emit data</button>
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {

  data:string = "Message from child to parent";

  @Output() dataEvent = new EventEmitter<string>();

  constructor() { }

  emitData(){
    this.dataEvent.emit(this.data);
  }
}
```

In the parent component's template we can capture the emitted data like this:

```
<app-child (dataEvent)="receiveData($event)"></app-child>
```

Then inside the receiveData function we can handle the emitted data:

```
receiveData($event){
  this.dataFromChild = $event;
}
```