Question 1

1. Explain the basic Angular concepts (components, directives, modules, services, pipes)

Components:

- Components are the basic building blocks of an Angular application.
- Each component encapsulates a specific part of the user interface and its behavior.
- Components consist of a TypeScript class and an HTML template.

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

@Component({
  selector: 'app-root',
  template: '<h1>Hello, {{ name }}</h1>',
})
export class AppComponent {
  name = 'Angular';
}
```

Directives:

- Directives are instructions in the DOM (Document Object Model) that Angular uses to manipulate the behavior or appearance of elements.
- Common directives include `ngIf`, `ngFor`, and `ngSwitch`.

Types of directives:

- **Attribute Directives:** Attribute directives are responsible for changing the appearance or behavior of an element, component, or another directive. They are typically applied to host elements as attributes. Examples include ngStyle and ngClass. Attribute directives are prefixed with *ng in their usage

```
<!-- Example of ngStyle →for example for dynamic styles
<div [ngStyle]="{ 'color': textColor, 'font-size': fontSize }">Styled
Text</div>

<!-- Example of ngClass -->

<div [ngClass]="{ 'highlight': isHighlighted, 'italic': isItalic }">Styled
Text</div>
```

- **Structural directives:** are responsible for changing the structure of the DOM by adding or removing elements based on conditions. They are typically prefixed with an asterisk (*) in their usage. Examples include `*ngIf` and `*ngFor`.

```html
<!-- Example of *ngIf -->
<div *ngIf="isVisible">Visible Content</div>

<!-- Example of *ngFor -->
<ul>
  <li *ngFor="let item of items">{{ item.name }}</li>
</ul>
```

Modules:

- Modules are used to organize the application into cohesive blocks.
- A module is a container for a set of components, services, directives, and pipes.
- Every Angular application has at least one module, the root module, conventionally named `AppModule`.
- An Angular module is a class with an `@NgModule()` decorator.

```typescript
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Services

- Services are used to encapsulate and share functionality across components.
- They are a way to keep components lean and focused on their specific tasks.
- A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

- Components often delegate fetching, validating, logging data to the services

Other use cases for services:

1. **Data Sharing:**
   - Services are often used to share data between components. They act as a centralized location to store and manage data that needs to be shared across different parts of an application.
2. **Business Logic:**
   - Services encapsulate business logic, allowing you to separate concerns and keep your components focused on their specific tasks. This promotes a modular and maintainable code structure.
3. **Dependency Injection:**
   - Angular's dependency injection system is used to inject services into components, making it easy to access their functionality. This promotes code reuse and testability.
4. **HTTP Communication:**
   - Services are commonly used to handle HTTP requests and responses. This includes making API calls to fetch or send data to a server. The Angular `HttpClient` module is often used within services for this purpose.

```
// user.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class UserService {
  private apiUrl = 'https://api.example.com/users';

  constructor(private http: HttpClient) {}

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>(this.apiUrl);
  }

  getUserById(userId: number): Observable<User> {
    const url = `${this.apiUrl}/${userId}`;
    return this.http.get<User>(url);
  }
  updateUser(user: User): Observable<User> {
    const url = `${this.apiUrl}/${user.id}`;
    return this.http.put<User>(url, user);
  }
}

interface User {
  id: number;
  name: string;
  email: string;
}
// user.component.ts
import { Component, OnInit } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-user',
  template: `
    <div *ngFor="let user of users">
      {{ user.name }} - {{ user.email }}
    </div>
  `,
})
export class UserComponent implements OnInit {
  users: User[];

  constructor(private userService: UserService) {}

  ngOnInit() {
    this.userService.getUsers().subscribe((users) => {
      this.users = users;
    });
  }
}
```

Pipes

- Pipes are used to transform the data before displaying it in the view.
- Angular provides built-in pipes for common transformations like date formatting, currency conversion, and more.
- You can create your own pipes

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: '<div>{{ currentDate | date: "short" }}</div>',
})
export class AppComponent {
  currentDate = new Date();
}
```

2. Explain how dependency injection is used in Angular

Dependency Injection (DI) is a fundamental concept in Angular that allows for a more modular and testable codebase. It is a design pattern in which a class requests dependencies from external sources rather than creating them.

The way it works in Angular is through a **hierarchy of injectors**. An injector is responsible for creating instances of dependencies and injecting them into classes that request them. When a class requests a dependency, the injector checks if it has already created an instance of that dependency. If not, it creates a new one, returns it to the class, and reserves a copy for further use. So, the next time the same dependency is requested, it returns the reserved dependency rather than creating a new one.

Here's an example of how dependency injection works in Angular:

```
// Product Service
import { Injectable } from '@angular/core';

@Injectable({
 providedIn: 'root'
})
export class ProductService {
 getProducts() {
   return ['Product 1', 'Product 2'];
 }
}

// Product List Component
import { Component } from '@angular/core';
import { ProductService } from './product.service';

@Component({
 selector: 'app-product-list',
 templateUrl: './product-list.component.html',
 styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
 products: string[];

 constructor(private productService: ProductService) {
   this.products = this.productService.getProducts();
 }
}
```

In this example, the ProductService is marked as @Injectable(), which means it can be injected into other classes. The ProductListComponent requests the ProductService in its constructor. Angular's DI system automatically provides an instance of ProductService when ProductListComponent is created. The ProductService is then used to fetch a list of products.

In summary, Dependency Injection in Angular provides a way to manage and organize dependencies in your application. It promotes loose coupling, easier unit testing, and improved code maintainability.

1. **Root Injector:** Created at application bootstrapping, provides global dependencies shared across the entire app.
2. **Module Injector:** Each Angular module has its own injector, providing dependencies specific to that module.
3. **Component Injector:** Every Angular component has its injector, providing dependencies specific to that component instance.

When a dependency is requested, Angular looks in the component injector first, then the module injector, and finally the root injector. This hierarchy ensures scoping and proper sharing of dependencies in Angular applications.

3. Explain how to component-to-component communication can be implemented

Component-to-component communication in Angular can be achieved in various ways, depending on the relationship between the components. Here are some common methods:

Parent to Child Communication:

If there is a parent-child relationship between components, data can be passed from the parent to the child using the @Input decorator. The parent component passes data to the child component via property binding.

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

@Component({
  selector: 'app-parent',
  template: `
    <app-child [dataFromParent]="parentData"></app-child>
  `,
})
export class ParentComponent {
  parentData = 'Data from Parent';
}


------------------------------------------------------------

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

@Component({
  selector: 'app-child',
  template: `
    <p>{{ dataFromParent }}</p>
  `,
})
export class ChildComponent {
  @Input() dataFromParent: string;
}
```

Child to Parent Communication:

For child-to-parent communication, the child component emits an event that the parent component can listen to. This is done using the @Output decorator and EventEmitter.

1. Use @Output decorator in child component and create EventEmitter
2. Create function in child .ts file and bind this function to a button in child template
3. In this function use emitter and pass received template data to emitter
4. In parent component create receiveData function, and pass it to child component like that: `(messageEvent)="receiveData($event)”`

```


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

@Component({
  selector: 'app-child',
  template: '<button (click)="sendMessage()">Send Message</button>',
})
export class ChildComponent {
  @Output() messageEvent = new EventEmitter<string>();

  sendMessage() {
    this.messageEvent.emit('Hello from Child!');
  }
}

---------------------------------------------------------------

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

@Component({
  selector: 'app-parent',
  template: `
      <app-child (messageEvent)="receiveMessage($event)"></app-child>`,
})
export class ParentComponent {
  receiveMessage(message: string) {
    console.log(message); // Outputs: Hello from Child!
  }
}
```

In this example, when the button in ChildComponent is clicked, it emits an event with the message 'Data from child'. ParentComponent listens to this event and logs the received data.

Communication Without Relationship:

For components that do not have a direct parent-child relationship, communication can be achieved using a shared service. This service can use BehaviorSubject or Subject from RxJS to create an observable that components can subscribe to.

1. You create a shared service with @Injectable annotation. There you create a Subject other components might subscribe to, and you declare some variable as observable.
2. You declare the SenderComponent and inject shared service through the constructor. There you use method from sharedService to send message you want to send to the subscribers

3. You declare a ConsumerComponent that has also injected sharedService. You use an observable variable from service and pass it a lambda that will get invoked when receiving the message.

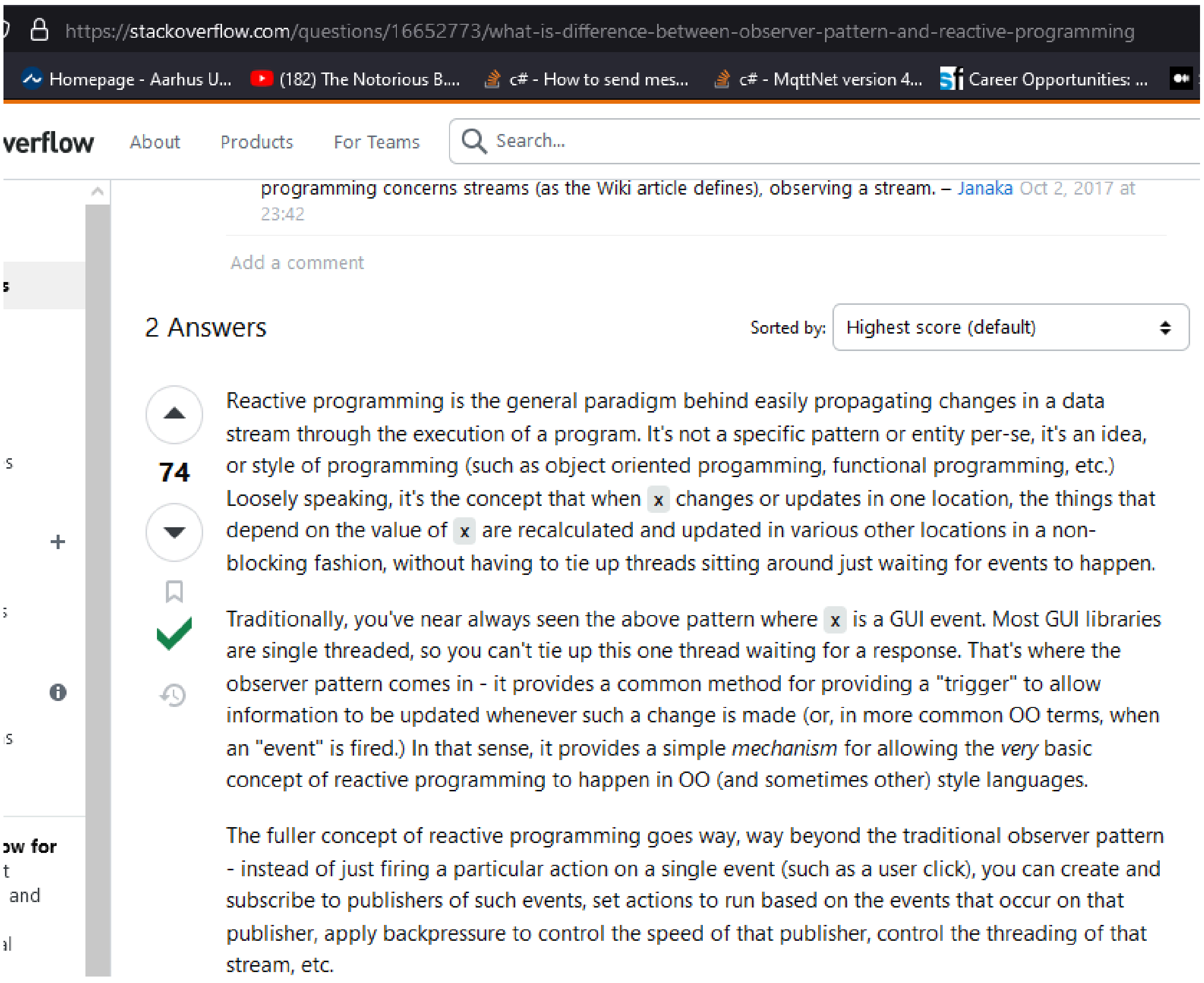
```
// message.service.ts
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class MessageService {
  messageSource = new Subject<string>();

  sendMessage(message: string) {
    this.messageSource.next(message);
  }
}

// sender.component.ts
import { Component } from '@angular/core';
import { MessageService } from './message.service';

@Component({
  selector: 'app-sender',
  template: `
    <input [(ngModel)]="message" placeholder="Type a message" />
    <button (click)="sendMessage()">Send Message</button>
  `,
})
export class SenderComponent {
  message: string = '';

  constructor(private messageService: MessageService) {}

  sendMessage() {
    this.messageService.sendMessage(this.message);
  }
}

// receiver.component.ts
import { Component, OnDestroy } from '@angular/core';
import { MessageService } from './message.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-receiver',
  template: '<p>{{ receivedMessage }}</p>',
})
export class ReceiverComponent implements OnDestroy {
  receivedMessage: string = '';
  private subscription: Subscription;

  constructor(private messageService: MessageService) {
    this.subscription = this.messageService.messageSource.subscribe((message)
=> {
      this.receivedMessage = message;
    });
  }

  ngOnDestroy() {
    // Unsubscribe to avoid memory leaks
    this.subscription.unsubscribe();
  }
}
```

Question 2

1. Explain reactive programming using RxJS

https://www.baeldung.com/cs/reactive-programming

Reactive programming is a declarative programming paradigm that is based on the idea of **asynchronous event processing and data streams**. Today, reactive programming is used in many different areas, such as GUI programming, web programming, microservices, or reactive systems in general.


https://stackoverflow.com/questions/16652773/what-is-difference-between-observer-pattern-and-reactive-programming
Homepage - Aarhus U...
(182) The Notorious B....
c# - How to send mes...
c# - MqttNet version 4...
Career Opportunities: ...
verflow
About
Products
For Teams
Search...
Add a comment
Sorted by:
Highest score (default)

programming concerns streams (as the Wiki article defines), observing a stream. – Janaka Oct 2, 2017 at 23:42

2 Answers

74

Reactive programming is the general paradigm behind easily propagating changes in a data stream through the execution of a program. It's not a specific pattern or entity per-se, it's an idea, or style of programming (such as object oriented progamming, functional programming, etc.) Loosely speaking, it's the concept that when `x` changes or updates in one location, the things that depend on the value of `x` are recalculated and updated in various other locations in a non-blocking fashion, without having to tie up threads sitting around just waiting for events to happen.

Traditionally, you've near always seen the above pattern where `x` is a GUI event. Most GUI libraries are single threaded, so you can't tie up this one thread waiting for a response. That's where the observer pattern comes in - it provides a common method for providing a "trigger" to allow information to be updated whenever such a change is made (or, in more common OO terms, when an "event" is fired.) In that sense, it provides a simple *mechanism* for allowing the *very* basic concept of reactive programming to happen in OO (and sometimes other) style languages.

The fuller concept of reactive programming goes way, way beyond the traditional observer pattern - instead of just firing a particular action on a single event (such as a user click), you can create and subscribe to publishers of such events, set actions to run based on the events that occur on that publisher, apply backpressure to control the speed of that publisher, control the threading of that stream, etc.

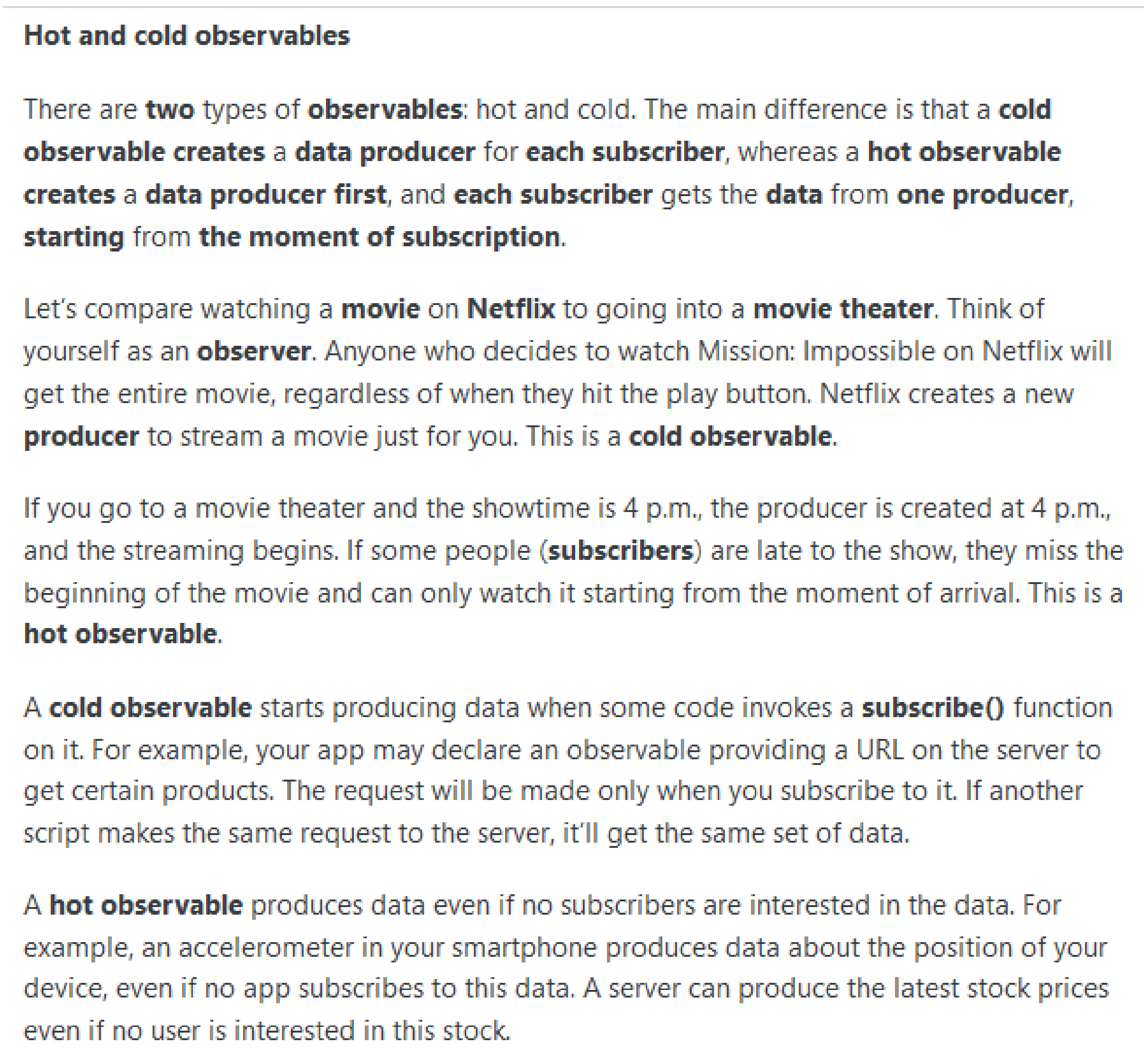
RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code. It allows developers to work with data streams and propagation of change in a declarative manner.

Reactive programming vs Event-driven architecture
Reactive programming and event-driven architecture are separate concepts but they share common ground in their approach to handling asynchronous events, promoting loose coupling, and providing real-time responsiveness. They are complementary paradigms that can be used together to build systems that are flexible, scalable, and responsive to changes.

From GPT:

Cold vs Hot observables

There are **two** types of **observables**: hot and cold. The main difference is that a **cold observable creates** a **data producer** for **each subscriber**, whereas a **hot observable creates** a **data producer first**, and **each subscriber** gets the **data** from **one producer**, **starting** from **the moment of subscription**.



Hot and cold observables

There are **two** types of **observables**: hot and cold. The main difference is that a **cold observable creates** a **data producer** for **each subscriber**, whereas a **hot observable creates** a **data producer first**, and **each subscriber** gets the **data** from **one producer**, **starting** from **the moment of subscription**.

Let's compare watching a **movie** on **Netflix** to going into a **movie theater**. Think of yourself as an **observer**. Anyone who decides to watch Mission: Impossible on Netflix will get the entire movie, regardless of when they hit the play button. Netflix creates a new **producer** to stream a movie just for you. This is a **cold observable**.

If you go to a movie theater and the showtime is 4 p.m., the producer is created at 4 p.m., and the streaming begins. If some people (**subscribers**) are late to the show, they miss the beginning of the movie and can only watch it starting from the moment of arrival. This is a **hot observable**.

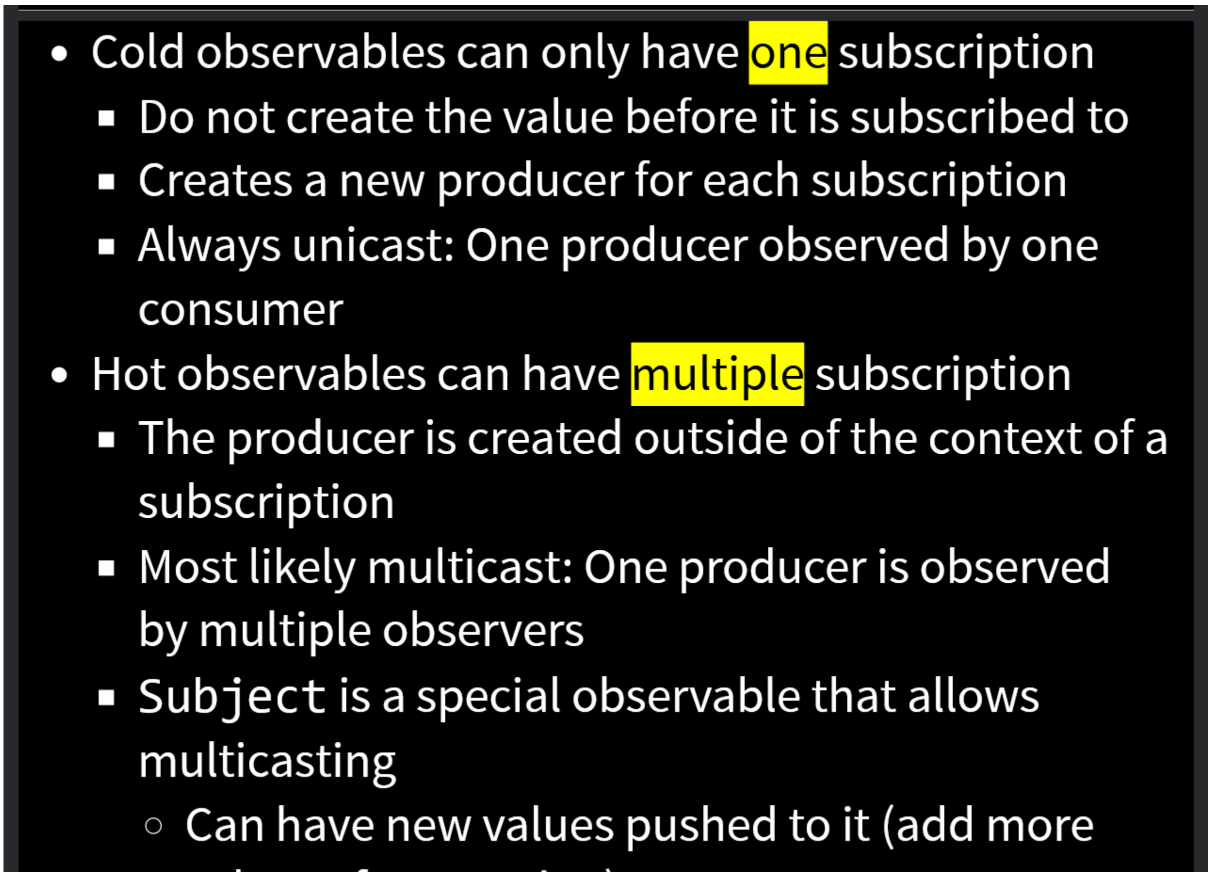
A **cold observable** starts producing data when some code invokes a **subscribe()** function on it. For example, your app may declare an observable providing a URL on the server to get certain products. The request will be made only when you subscribe to it. If another script makes the same request to the server, it'll get the same set of data.

A **hot observable** produces data even if no subscribers are interested in the data. For example, an accelerometer in your smartphone produces data about the position of your device, even if no app subscribes to this data. A server can produce the latest stock prices even if no user is interested in this stock.

Here are the key concepts in RxJS:

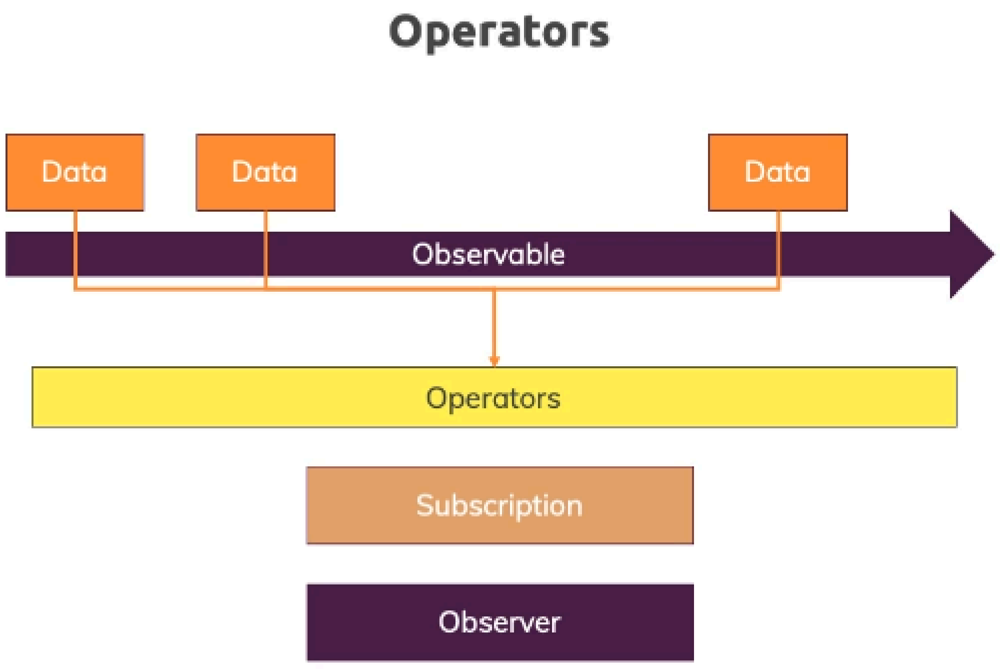
Observer pattern:

Consumer - The code that is subscribing to an Observable
Observer - The manifestation of a Consumer, that may have some (or all) handlers
Producer - Any entity that is the source of values
Subscription - A contract where a consumer is observing values pushed by a producer
Observable- A template for connecting an Observer, as a Consumer, to a Producer, via a subscribe action, resulting in a Subscription



- Cold observables can only have one subscription
  - Do not create the value before it is subscribed to
  - Creates a new producer for each subscription
  - Always unicast: One producer observed by one consumer
- Hot observables can have multiple subscription
  - The producer is created outside of the context of a subscription
  - Most likely multicast: One producer is observed by multiple observers
  - `Subject` is a special observable that allows multicasting
    - Can have new values pushed to it (add more

GPT key concepts RxJS:



RxJS (Reactive Extensions for JavaScript) is a library for reactive programming in JavaScript, and it plays a crucial role in the Angular framework. Here are some key concepts in RxJS:

Observable:

- An `Observable` represents a sequence of values or events that can be observed over time.. Observables can emit values, errors, and completion signals. Observable can emit: `next` (for values), `error` (for errors), and `complete` (for completion).

```
import { Observable } from 'rxjs';

const observable = new Observable<number>((observer) => {
  observer.next(1);
  observer.next(2);
  observer.complete();
});

observable.subscribe({
  next: (value) => console.log('Next:', value),
  complete: () => console.log('Complete'),
});
```

Observer:

- An `Observer` is an object that defines callbacks to handle the three types of notifications that an Observable can emit: `next` (for values), `error` (for errors), and `complete` (for completion).

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

@Component({
  selector: 'app-observable-example',
  template: '<div>{{ data }}</div>',

})

export class ObservableExampleComponent implements OnInit {

  data: number;

  ngOnInit() {
    const observable = new Observable<number>((observer) => {

      observer.next(42);

    });

    observable.subscribe((value) => {

      this.data = value;

    }); }}
```

Subscription:
Subscriptions in Angular represent the ongoing connection between Observables and Observers.

```
const observable = new Observable<number>((observer) => {
  const intervalId = setInterval(() => {
    observer.next(Math.random());
  }, 1000);

  return () => clearInterval(intervalId);
});

const subscription = observable.subscribe((value) => console.log('Random
Value:', value));

// Unsubscribe after 5 seconds to stop the interval
setTimeout(() => {
  subscription.unsubscribe();
}, 5000);
```

Operators:

Definition: Functions that enable the manipulation, transformation, and combination of Observables

```
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

const numbersObservable = new Observable<number>((observer) => {
  observer.next(1);
  observer.next(2);
  observer.complete();
});

const squaredNumbersObservable = numbersObservable.pipe(
  map((value) => value * value)
);

squaredNumbersObservable.subscribe((value) => console.log('Squared Value:',
value));
```

Subject:

Definition: A special type of Observable that allows multicasting events to multiple Observers.

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

@Injectable({

  providedIn: 'root',

})

export class DataService {

  private dataSubject = new Subject<number>();

  fetchData(): void {

    // Simulate fetching data asynchronously

    setTimeout(() => {

      const randomValue = Math.random();

      this.dataSubject.next(randomValue);

    }, 1000);

  }

  getDataObservable(): Observable<number> {

    return this.dataSubject.asObservable();
  }}
```

BehaviorSubject
A type of Subject that retains the latest emitted value. Subscribers receive the last emitted value immediately upon subscription.

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

@Injectable({
  providedIn: 'root',
})
export class DataService {
  private dataSubject = new BehaviorSubject<number>(0);

  fetchData(): void {
    // Simulate fetching data asynchronously
    setTimeout(() => {
      const randomValue = Math.random();
      this.dataSubject.next(randomValue);
    }, 1000);
  }

  getDataObservable(): Observable<number> {
    return this.dataSubject.asObservable();
  }
}
```

2. Explain how network communication is done in Angular according to best practices (use of `HttpClientModule`)

Network communication in Angular is primarily handled using the HttpClient module. Angular httpclient will always return an observable (RXjs) , promise is not available in Angular, it was available in AngularJs. In Angular you need to subscribe to the observable.

Here's how to use it according to best practices:

1. Import HttpClientModule

Import the HttpClientModule in your Angular module. This is typically done in the AppModule or in a feature module where you plan to use HTTP services.After importing HttpClientModule, you can inject HttpClient into any component or service where you need to make HTTP requests 1, 3.

```
import { HttpClientModule } from '@angular/common/http';

NgModule({

  imports: [HttpClientModule],

  // other module configurations

})

export class AppModule { }
```

2. **Create a Service for HTTP Requests:** It's a best practice to encapsulate HTTP communication logic within a service. Create a service dedicated to handling HTTP requests. Use Angular's `HttpClient` service within this service.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root', // providedIn: 'root' makes this service a singleton
and available throughout the app
})
export class DataService {
  private apiUrl = 'https://api.example.com';

  constructor(private http: HttpClient) {}

  getSomeData(): Observable<any> {
    return this.http.get(`${this.apiUrl}/some-endpoint`);
  }

  // Add other HTTP methods as needed (post, put, delete, etc.)
}
```

3. **Use Dependency Injection:** Inject the `DataService` into the components or services where you need to make HTTP requests. This ensures that Angular's dependency injection system manages the instances.

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-example',
  template: `
    <div>{{ responseData }}</div>
  `,
})
export class ExampleComponent implements OnInit {
  responseData: any;
  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.dataService.getSomeData().subscribe(
      (data) => {
        this.responseData = data;
      },
      (error) => {
        console.error('Error fetching data:', error);
      }
    );
  }
}
```

4. **Handle Errors and Loading:** Always handle errors and loading states appropriately. Use the `subscribe` method's second callback to handle errors, and consider incorporating loading spinners or messages during the HTTP request.

```
ngOnInit() {

  this.isLoading = true;

  this.dataService.getSomeData().subscribe(

    (data) => {     this.responseData = data;     },

    (error) => {
      console.error('Error fetching data:', error);
    },

    () => {     this.isLoading = false;     }

  );}
```

5. **RxJS and Observables:** Leverage the power of RxJS observables when working with HTTP requests. This enables you to compose and manipulate asynchronous data streams easily.

```
import { Observable } from 'rxjs';
import { catchError, map } from 'rxjs/operators';
```

```
getSomeData(): Observable<any> {

  return this.http.get(`${this.apiUrl}/some-endpoint`).pipe(

    map((response) => response),

    catchError((error) => {

      console.error('Error:', error);

      throw error; // rethrow the error for further handling

    })

  );

}
```

You can also use typed responses. TypeScript's static typing allows you to define the shape of the data you expect from an API, and Angular's HttpClient can be configured to return typed observables. Here's how you can achieve typed responses:

1. **Create TypeScript interfaces or types t**hat represent the structure of the data you expect from the API. This helps you catch potential errors during development.

```
// data.model.ts
export interface SomeData {
  id: number;
  name: string;
  // other properties
}
```

2. **Use Typed Observables in the Service:** Update your service to use the typed observables by specifying the type parameter in the `get` method.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { SomeData } from './data.model';
```

```
@Injectable({
  providedIn: 'root',
})
export class DataService {
  private apiUrl = 'https://api.example.com';

  constructor(private http: HttpClient) {}

  getSomeData(): Observable<SomeData> {
    return this.http.get<SomeData>(`${this.apiUrl}/some-endpoint`);
  }
}
```

3. **Handle Typed Responses in Components:** In your components, you can now work with the typed responses directly.

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';
import { SomeData } from './data.model';

@Component({
  selector: 'app-example',
  template: `
    <div>
      <div>{{ responseData?.id }}</div>
      <div>{{ responseData?.name }}</div>
      <!-- other properties -->
    </div>
  `,
})
export class ExampleComponent implements OnInit {
  responseData: SomeData;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.dataService.getSomeData().subscribe(
      (data) => {
        this.responseData = data;
      },
      (error) => {
        console.error('Error fetching data:', error);
      }
    );
  }
}
```

Question 3

1. Explain how routing works in Angular

Routing in Angular is managed by the Angular Router module, which enables navigation from one view to another as users perform application tasks. This is a critical aspect of building single-page applications (SPAs), as it allows for seamless transitions between views without requiring a full page reload 3.

Setting Up Routing

To set up routing in an Angular application, you first need to import the RouterModule and Routes from @angular/router in your module (*.module.ts) that is supposed to manage routing:

```
import { RouterModule, Routes } from '@angular/router';
```

Next, define your routes. Each route is an object with a path and a component. The path is a string that specifies the URL path for the route, and the component is the component that should be displayed when the route is activated:

```
const routes: Routes = [
 { path: '', redirectTo: '/home', pathMatch: 'full' },
 { path: 'home', component: HomeComponent },
 { path: 'about', component: AboutComponent },
];
```

Finally, register your routes with the RouterModule using method:

- forRoot() - if that's the root module
- forChild() - if that's the child module (routes will be relative to the component)

```
@NgModule({
 imports: [
   RouterModule.forRoot(routes)
 ],
 exports: [RouterModule]
})
export class AppRoutingModule { }
```

This sets up the basic routing structure for your application.

Navigating Between Routes

You can navigate between routes using the routerLink directive in your templates. This directive binds a clickable HTML element to a route. When the element is clicked, the router navigates to the linked route:

```
<a routerLink="/home">Home</a>
<a routerLink="/about">About</a>
```

In this example, clicking on "Home" navigates to the /home route, and clicking on "About" navigates to the /about route.

Displaying Route Components

To display the component associated with the current route, use the <router-outlet> directive in your templates. This directive acts as a placeholder that Angular dynamically fills based on the current router state:

```
<router-outlet></router-outlet>
```

When the route changes, the content inside <router-outlet> will change to reflect the component associated with the new route.

One can also have secondary named router outlets.

```
{ path: 'home', component: HomeComponent },
{ path: 'news', component: NewsComponent, outlet: 'sidebar' },
```

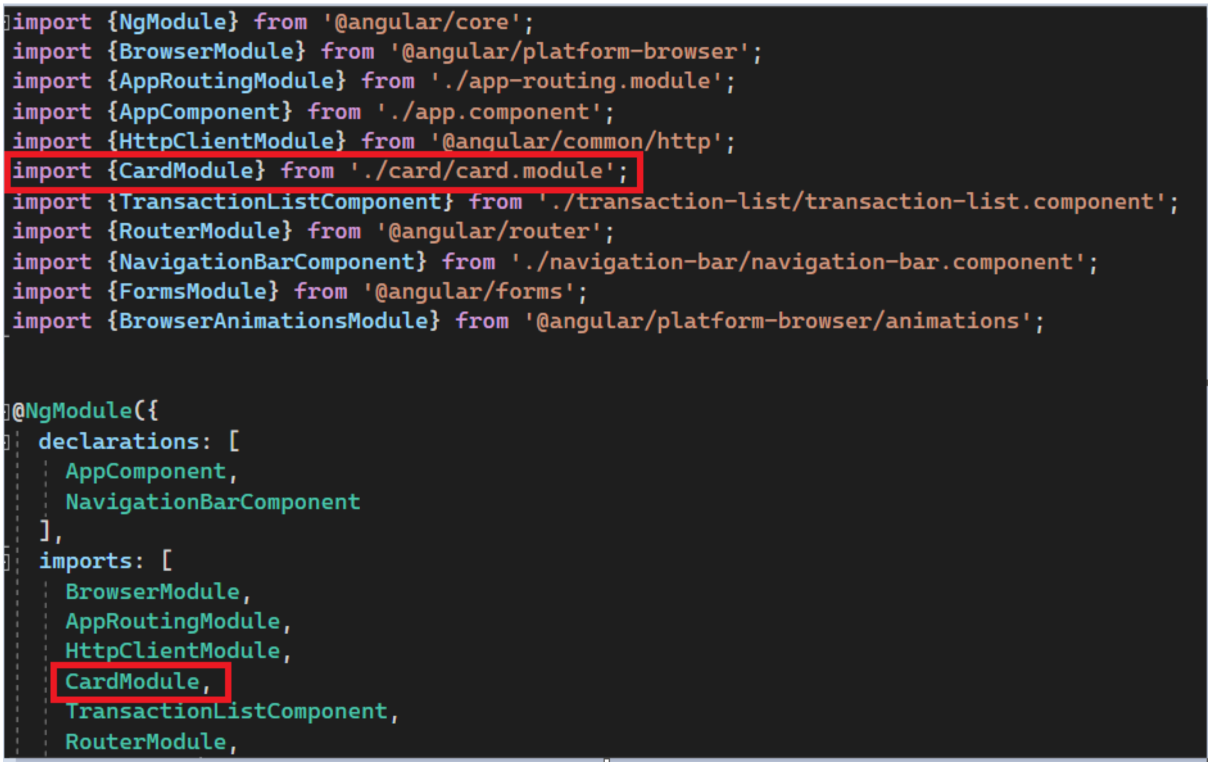
```
<div class="main-content">
  <h1>Main Content Area</h1>
  <router-outlet></router-outlet> // Primary “default” outlet
</div>

<div class="sidebar">
  <h2>Sidebar Content</h2>
  <router-outlet name="sidebar"></router-outlet> //Secondary named outlet
</div>
```

This functionality is useful ie. if we need two separate sections with updating content dependant on the route.

2. Explain module routing and lazy-loading

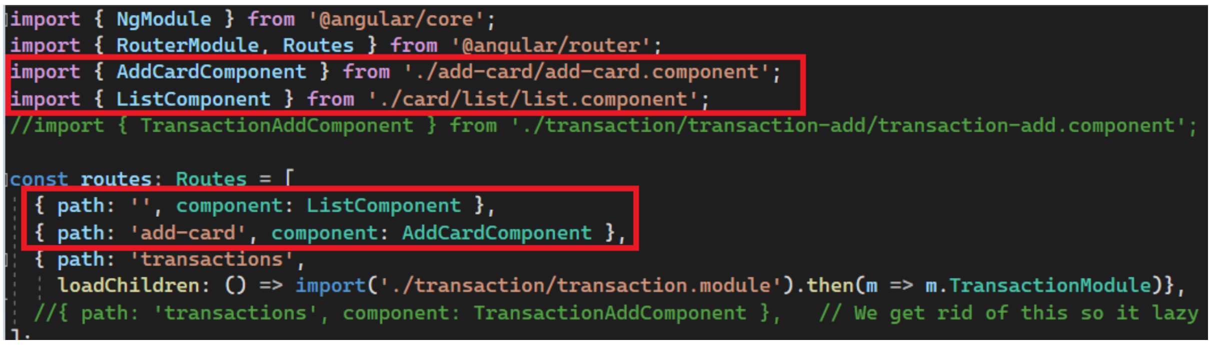
Regular module routing is usually performed by simply importing necessary modules in the main module (in @NgModule imports) and then importing necessary components (both to *.module.ts and *-routing.module.ts). Then we can perform routing without worrying what modules are invoked components in.

app.module.ts - importing CardModule (and some others necessary things for routing)



```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {AppRoutingModule} from './app-routing.module';
import {AppComponent} from './app.component';
import {HttpClientModule} from '@angular/common/http';
import {CardModule} from './card/card.module';
import {TransactionListComponent} from './transaction-list/transaction-list.component';
import {RouterModule} from '@angular/router';
import {NavigationBarComponent} from './navigation-bar/navigation-bar.component';
import {FormsModule} from '@angular/forms';
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';


@NgModule({
  declarations: [
    AppComponent,
    NavigationBarComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    CardModule,
    TransactionListComponent,
    RouterModule,
```

app-routing.module.ts - importing components from different modules



```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AddCardComponent } from './add-card/add-card.component';
import { ListComponent } from './card/list/list.component';
//import { TransactionAddComponent } from './transaction/transaction-add/transaction-add.component';

const routes: Routes = [
  { path: '', component: ListComponent },
  { path: 'add-card', component: AddCardComponent },
  { path: 'transactions',
    loadChildren: () => import('./transaction/transaction.module').then(m => m.TransactionModule)},
  //{ path: 'transactions', component: TransactionAddComponent },   // We get rid of this so it lazy
```

Lazy Loading

Lazy loading in Angular is a technique where you defer the initialization of an Angular module until it is actually needed. This can significantly improve the performance of your application, especially when you have large modules that aren't needed immediately when the application starts.

Setting Up Lazy Loading

To set up lazy loading, you first need to create a separate Angular module for the part of your application that you want to lazy load. You can do this using the Angular CLI:

```
ng generate module modulea --route a --module app.module
```

This command generates a new module named modulea with a route a and registers it with the app.module. The generated modulea contains its own module.ts, routing.ts, and component files.

Configuring Lazy Loading

Next, you need to configure your main routing module to lazy load the new module. You do this by using the loadChildren property in your routes configuration:

```
const routes: Routes = [
 { path: 'a', loadChildren: () => import('./modulea/modulea.module').then(m =>
m.ModuleaModule) },
];
```

In this example, when the /a route is visited, the ModuleaModule is loaded lazily.

Navigating to Lazy Loaded Module

You can navigate to the lazy loaded module using the routerLink directive in your templates:

```
<a routerLink="/a">Go to Module A</a>
```

When you click on "Go to Module A", the ModuleaModule is loaded lazily and the associated component is displayed.

Lazy loading can significantly improve the performance of your Angular application, especially when you have large parts of your application that aren't needed immediately when the application starts. However, it is not the best idea in small applications as it increases number of requests made - in such case it’s better to avoid it (especially if the user usually enters this module when using the app).

3. Explain how to get route information in the navigated-to component

To access route information in the navigated-to component, you can use the ActivatedRoute service provided by Angular. This service contains information about the route associated with the component that is loaded in an outlet.

Here's how you can use it:

Step 1: Import ActivatedRoute

First, import ActivatedRoute from @angular/router in your component:

```
import { ActivatedRoute } from '@angular/router';
```

Step 2: Inject ActivatedRoute

Next, inject ActivatedRoute in your component's constructor:

```
constructor(private route: ActivatedRoute) { }
```

Step 3: Access Route Information

You can now access various pieces of information about the current route using the properties of ActivatedRoute. Here are a few examples:

Path Parameters: You can access path parameters using the paramMap property:

```
this.route.paramMap.subscribe(params => {
 console.log(params.get('id')); // replace 'id' with the name of your path
parameter
});
```

Query Parameters: You can access query parameters using the queryParamMap property:

```
this.route.queryParamMap.subscribe(params => {
 console.log(params.get('queryParamName')); // replace 'queryParamName' with the name
of your query parameter
});
```

Route Data: You can access data associated with the route using the data property:

```
this.route.data.subscribe(data => {
 console.log(data);
});
```

In Angular, the `data` property of a route allows you to associate arbitrary data with a particular route. This can be useful for providing additional information or configuration for a component associated with that route. You can pass for example data regarding the role of the user.

```
const routes: Routes = [
  {
    path: 'user/:id',
    component: UserProfileComponent,
    data: { title: 'User Profile', isAdmin: true }
  }
];
```

4. Explain how to protect routes with guards

In Angular, route guards are interfaces that decide whether a route can be activated or not. They can be used to protect routes based on certain conditions, such as user authentication status or roles. There are several types of route guards, including CanActivate, CanActivateChild, CanDeactivate, Resolve, and CanLoad.

Creating a Route Guard

First, you need to create a service that implements one of the route guard interfaces. For example, to create a simple route guard that checks if a user is authenticated:

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree }
from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
 providedIn: 'root'
})
export class AuthGuard implements CanActivate {
 canActivate(
  next: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean |
UrlTree> | boolean | UrlTree {
 // Replace this with your actual authentication check
 const isAuthenticated = localStorage.getItem('authToken') !== null;
 return isAuthenticated;
 }
}
```

In this example, the canActivate method checks if the user is authenticated by checking if an auth token exists in local storage. If the user is authenticated, it returns true; otherwise, it returns false.

Applying the Route Guard

Next, apply the route guard to a route by using it as a provider in your routing module:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AuthGuard } from './auth.guard';

const routes: Routes = [
 { path: 'protected', component: ProtectedComponent, canActivate: [AuthGuard] },
];

@NgModule({
 imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```
In this example, the AuthGuard is applied to the /protected route. Whenever a user tries to navigate to this route, the AuthGuard's canActivate method is called. If the method returns true, the navigation proceeds; if it returns false, the navigation is canceled

Remember, while route guards can enhance the user experience by controlling access to routes, they should not be used as a security measure. Any data or functionality that should be restricted should also be secured on the server.

5. Explain why and how to use Server-Side Rendering (SSR)

Server-Side Rendering (SSR) is a popular technique for rendering a normally client-side single page app (SPA) on the server and sending a fully rendered page to the client. The client's JavaScript bundle can then take over and the SPA can operate as normal. SSR can provide a better user experience and improved SEO.

Why Use SSR?

SSR is particularly beneficial when you want to optimize for SEO or improve the initial page load performance. Since search engine crawlers usually don't perform any JS scripts when indexing pages (they only read static pages), SSR can provide them with a rendered page, making the site more discoverable on search engines. Similarly, since the server sends a fully rendered initial page to the client, the user sees the content faster, leading to a better user experience.

SSR benefits:
- Facilitated web crawlers through search engine optimization (SEO) due to serving some static content
- Improved performance on mobile and low-powered devices (some devices might not support JavaScript)
- Show the first page quickly (critical for user engagement) while loading the full application in the background

Performance concerns

Terms:
TTFB —*Time to First Byte*.
Seen as the time between clicking a link and the first bit of content coming in
FP —*First Paint*.
The first time any pixel gets becomes visible to the user
FCP —*First Contentful Paint*.
The time when requested content (article body, etc) becomes visible (FCP measures how long it takes the browser to render the first piece of DOM content after a user navigates to your page)

TTI —*Time To Interactive*.
The time at which a page becomes interactive (events wired up, etc)

LCP - The Largest Contentful Paint (LCP) metric reports the render time of the largest image or text block visible within the viewport, relative to when the user first navigated to the page.

Comparison - static, CSR, SSR

Static rendering
👍—Fast FP, FCP, TTI, and TTFB
👎—All HTML must be rendered, infeasable if unable to predict

SSR
👍—Fast FP, FCP, and TTI
👎—Slow TTFB

CSR
👍—Flexible, fast TTFB
👎—Slow TTI and FCP

How to Use SSR in Angular?

Angular Universal is a technology for Angular applications that allows you to render your Angular app on the server. It can be used for Server-Side Rendering (SSR) and Pre-Rendering.

To use Angular Universal, you would typically start by installing the Angular Universal package:

```
ng add @nguniversal/express-engine
```

Then, build your application and run it:

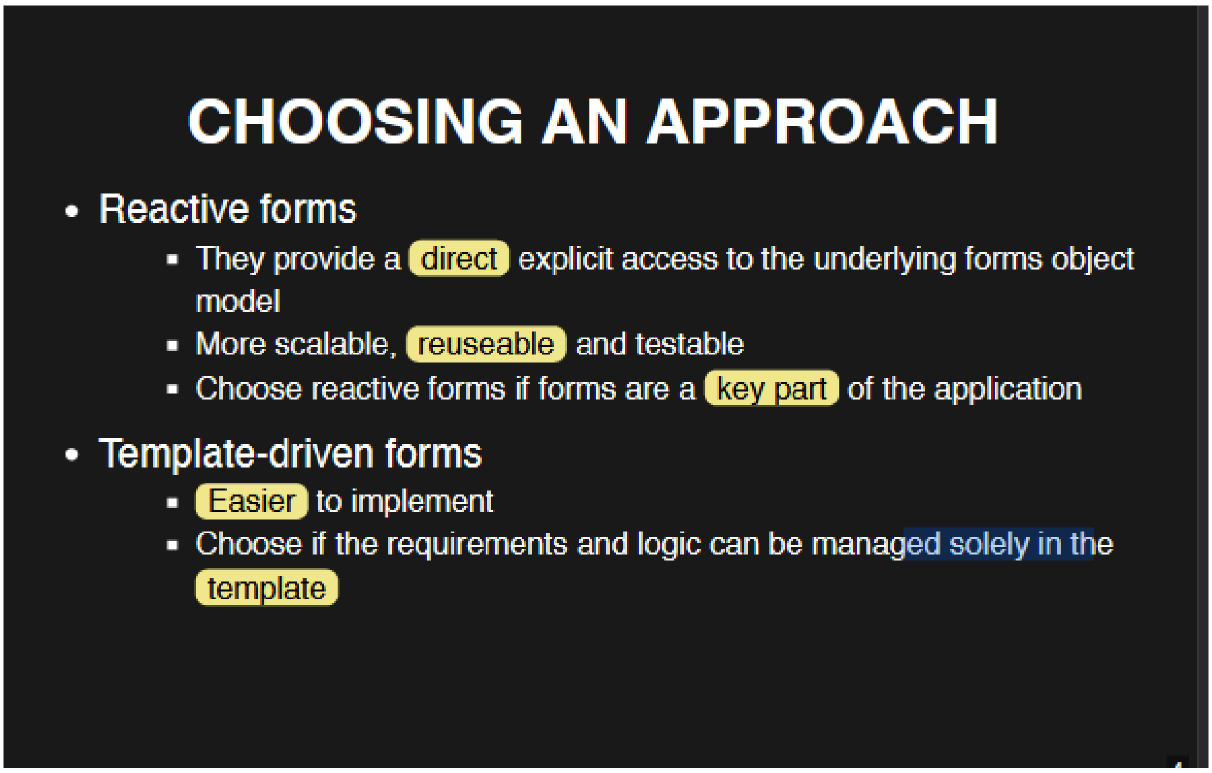
```
npm run build:ssr && npm run serve:ssr
```

This will start a Node Express server that serves your Angular application.

Challenges with SSR

While SSR provides many benefits, it does come with some challenges. One of the main challenges is managing the complexity of merging server-rendered and client-rendered content. Another challenge is handling dynamic components and directives that rely on client-side APIs, such as window or document objects. These components cannot be rendered on the server because these APIs do not exist there.

Question 4


CHOOSING AN APPROACH

- Reactive forms
  - They provide a direct explicit access to the underlying forms object model
  - More scalable, reuseable and testable
  - Choose reactive forms if forms are a key part of the application
- Template-driven forms
  - Easier to implement
  - Choose if the requirements and logic can be managed solely in the template

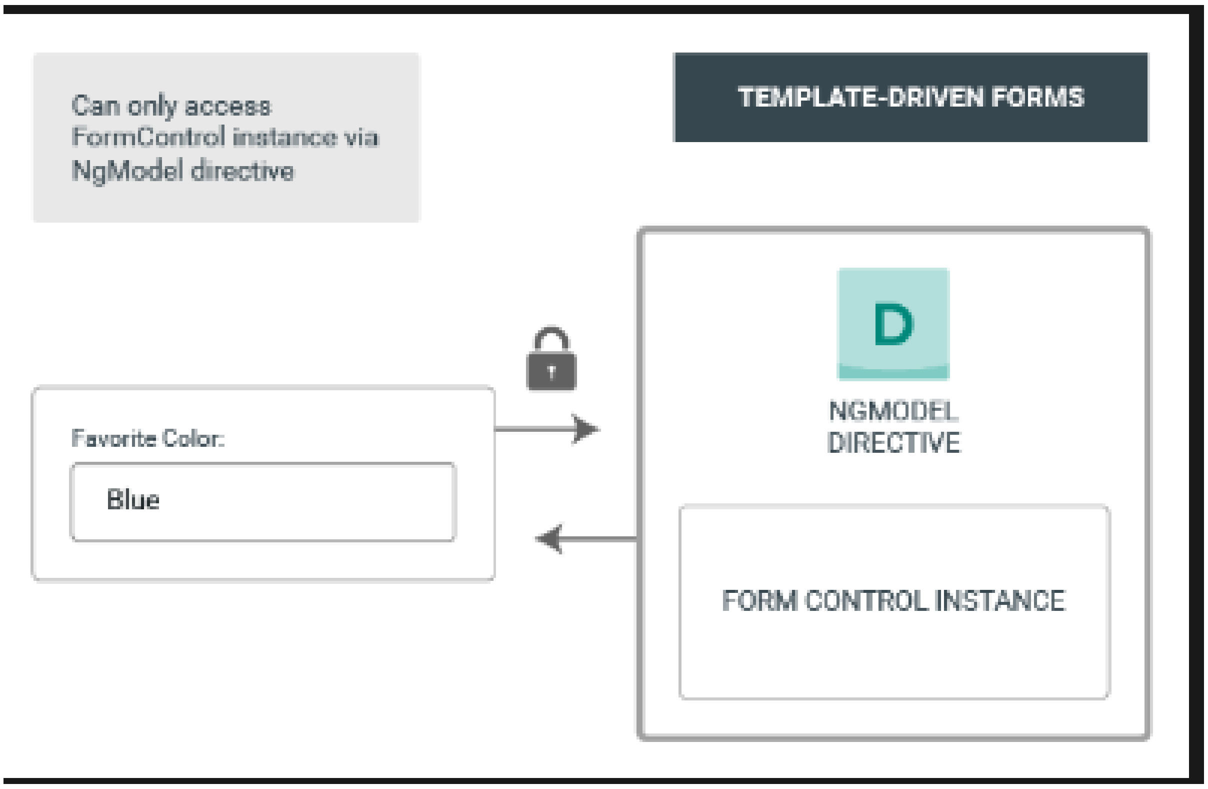
NgModel —used to mark HTML elements as part of the data model

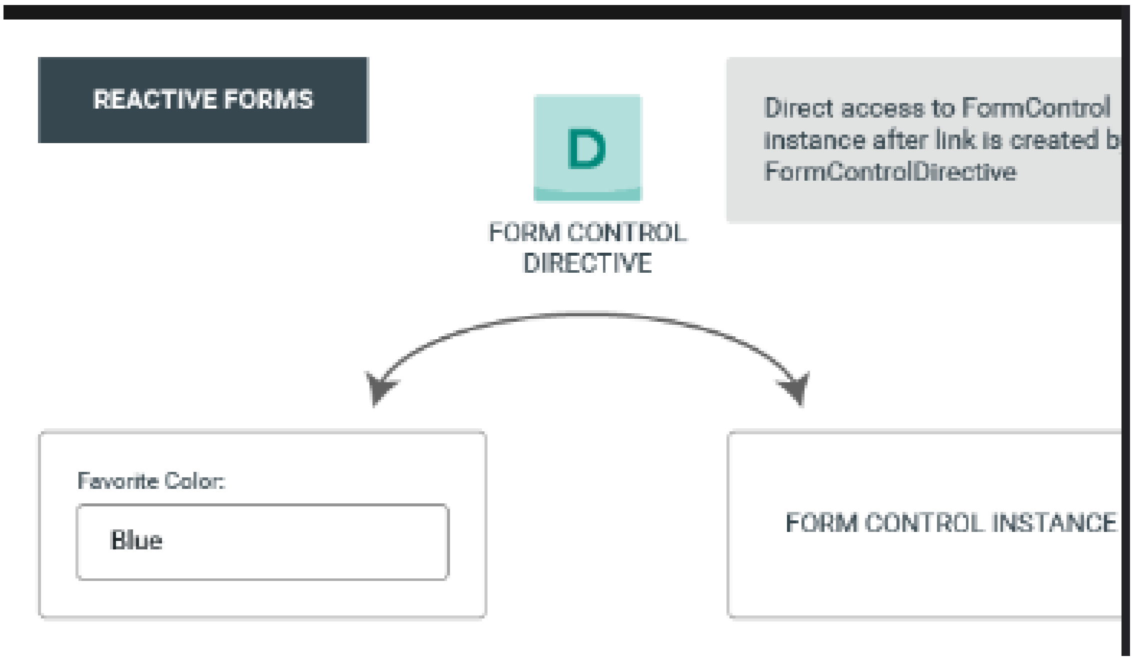
1. Explain template-driven forms in Angular

Reactive and template-driven forms process and manage data differently

Template driven forms are:
- good for simple forms
- simple validation
- easier to create
- require less code

Templates in template-driven approach are implicitly transferred to a model. In reactive forms approach you can directly access FormGroup members (FormControl field) and you explicitly declare model.





Template-driven forms in Angular provide a straightforward approach to handle user input and perform validation. They are simpler to implement and require less code than reactive forms, making them ideal for simple scenarios.

Setting Up a Template-Driven Form

To create a template-driven form, you need to import FormsModule from @angular/forms in your module:

```
import { FormsModule } from '@angular/forms';

@NgModule({
 imports: [
  FormsModule
 ],
 ...
})
export class AppModule { }
```

Next, you can create an HTML form in your component's template. Use the ngModel directive to create a FormControl instance for each form control:

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <div id="user-data">
    <div class="form-group">
      <label for="username">Username</label>
      <input
        type="text"
        id="username"
        class="form-control"
        ngModel
        name="username"
      />
    </div>
  <button class="btn btn-primary" type="submit">Submit</button>
</form>
```

In this example, #f="ngForm" creates a local template variable ngForm that holds a reference to the NgForm directive. (ngSubmit)="onSubmit(ngForm)" handles the form submission event. The ngModel directive binds the input field to a FormControl.

Handling Form Submission

To handle form submission, you can define a method in your component class:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  onSubmit(form: NgForm) {
    console.log(form);
  }
}
```

In this example, onSubmit() logs the form values when the form is submitted.

Also, you don’t have to pass the reference through the onSubmit function parameter. You can also use @ViewChild annotation to get access to the Form reference

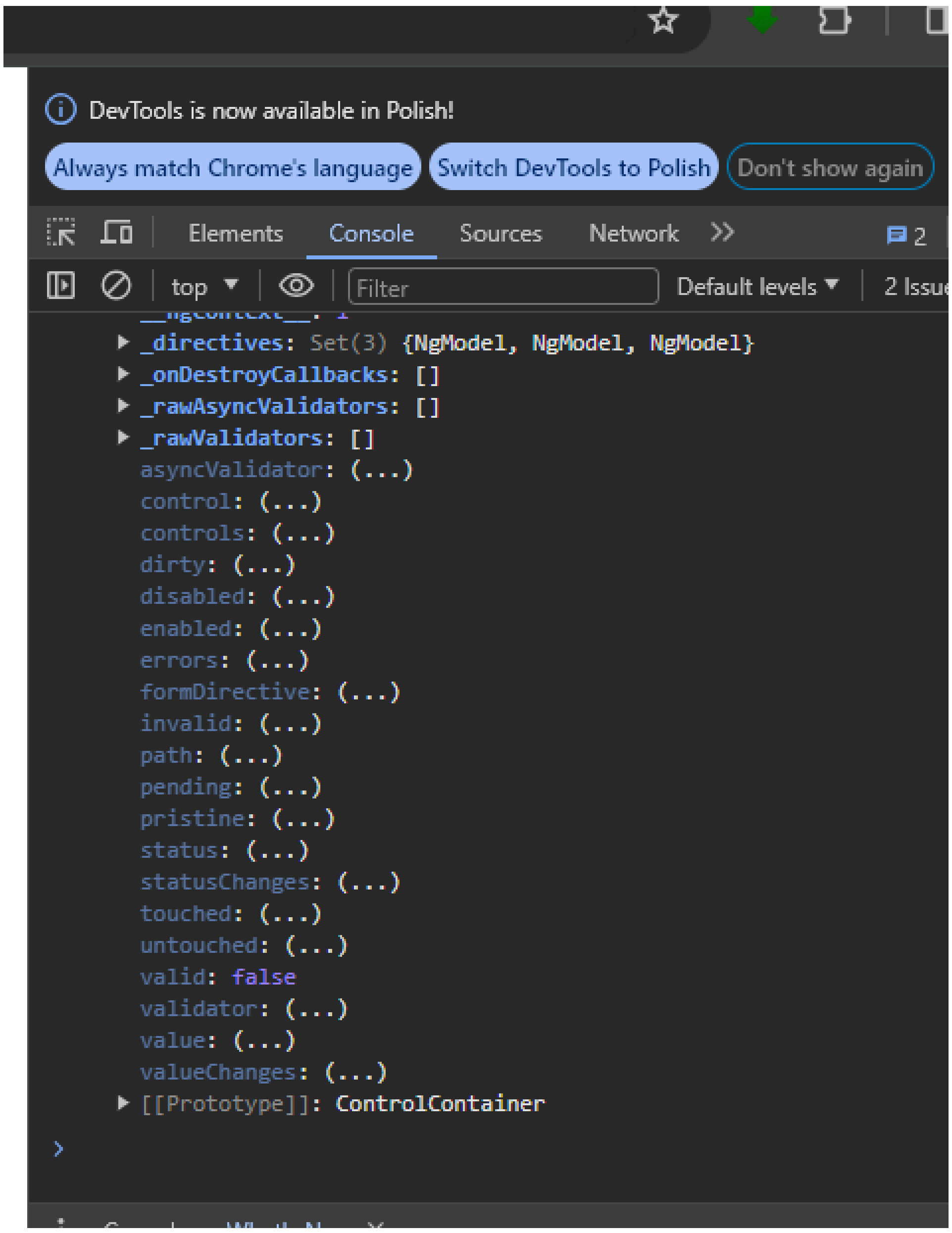
```
import { Component, ViewChild } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  @ViewChild('f') form: NgForm;

  onSubmit() {
    console.log(this.form);
  }
}

<form (ngSubmit)="onSubmit()" #f="ngForm">
  <div id="user-data">
    <div class="form-group">
      <label for="username">Username</label>
      <input
        type="text"
        id="username"
        class="form-control"
        ngModel
        name="username"
      />
    </div>
  <button class="btn btn-primary" type="submit">Submit</button>
```

```
</form>
```

Also there are different ways of getting form values. You can get them by accessing the passed form object (either with ViewChild or parameter approach)

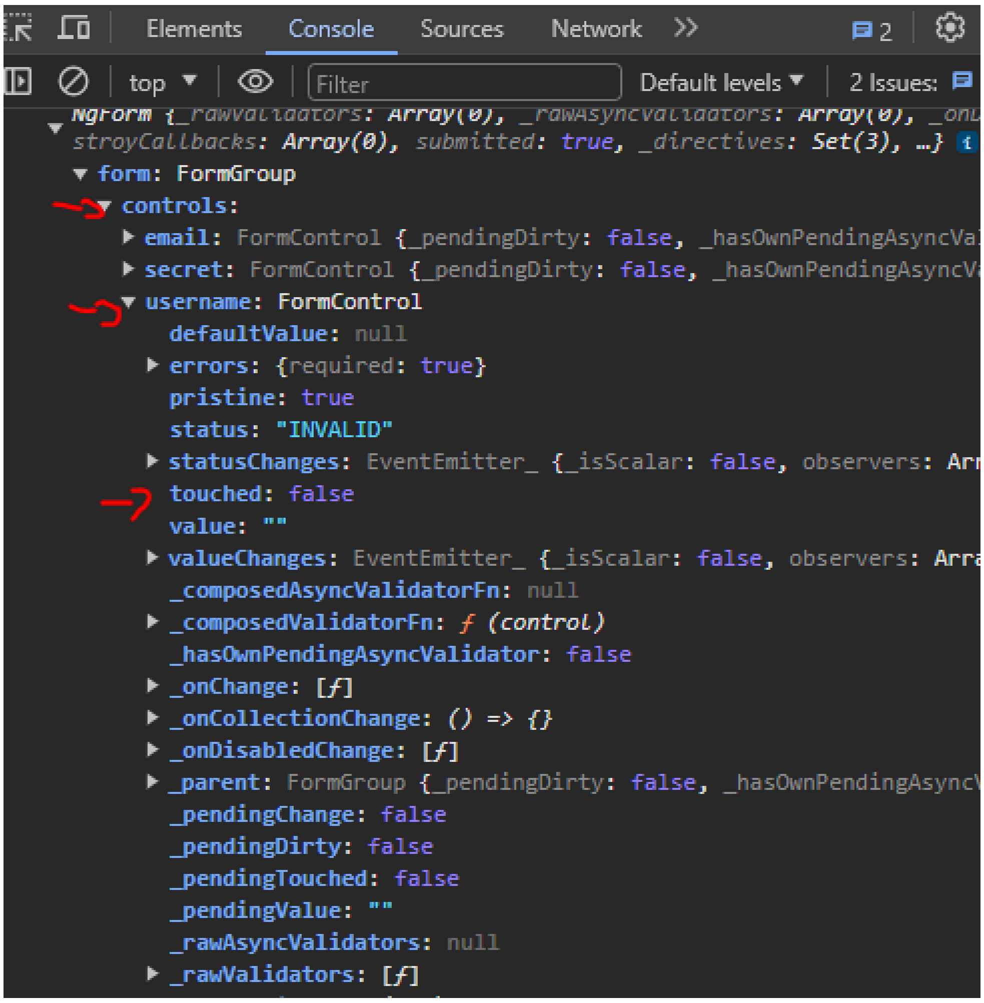
Adding Validation

Template-driven forms support simple validation out of the box. You can use built-in validators like required, minlength, maxlength, etc.:


DevTools is now available in Polish!
Always match Chrome's language
Switch DevTools to Polish
Don't show again
Elements
Console
Sources
Network
top
Filter
Default levels

```
▶ _directives: Set(3) {NgModel, NgModel, NgModel}
▶ _onDestroyCallbacks: []
▶ _rawAsyncValidators: []
▶ _rawValidators: []
  asyncValidator: (...)
  control: (...)
  controls: (...)
  dirty: (...)
  disabled: (...)
  enabled: (...)
  errors: (...)
  formDirective: (...)
  invalid: (...)
  path: (...)
  pending: (...)
  pristine: (...)
  status: (...)
  statusChanges: (...)
  touched: (...)
  untouched: (...)
  valid: false
  validator: (...)
  value: (...)
  valueChanges: (...)
▶ [[Prototype]]: ControlContainer
```

NgForm passes multiple states of the form like touched, valid, pristine which tell you what is the state of the form. For each of the controls you also have their state:



```
▼ NgForm {_rawValidators: Array(0), _rawAsyncValidators: Array(0), _onD
  stroyCallbacks: Array(0), submitted: true, _directives: Set(3), …}
  ▼ form: FormGroup
    ▼ controls:
      ▶ email: FormControl {_pendingDirty: false, _hasOwnPendingAsyncVa
      ▶ secret: FormControl {_pendingDirty: false, _hasOwnPendingAsyncV
      ▼ username: FormControl
          defaultValue: null
        ▶ errors: {required: true}
          pristine: true
          status: "INVALID"
        ▶ statusChanges: EventEmitter_ {_isScalar: false, observers: Arr
          touched: false
          value: ""
        ▶ valueChanges: EventEmitter_ {_isScalar: false, observers: Arra
          _composedAsyncValidatorFn: null
        ▶ _composedValidatorFn: ƒ (control)
          _hasOwnPendingAsyncValidator: false
        ▶ _onChange: [ƒ]
        ▶ _onCollectionChange: () => {}
        ▶ _onDisabledChange: [ƒ]
        ▶ _parent: FormGroup {_pendingDirty: false, _hasOwnPendingAsync
          _pendingChange: false
          _pendingDirty: false
          _pendingTouched: false
          _pendingValue: ""
          _rawAsyncValidators: null
        ▶ _rawValidators: [ƒ]
```

Using ngModel you can also access these values in the template to conditionally display some information.

```
 <div class="form-group">
          <label for="email">Mail</label>
          <input
            type="email"
            id="email"
            class="form-control"
            ngModel
            name="email"
            required
            email
            #email="ngModel"
          />
          <p *ngIf="!email.valid && email.touched">Email not valid</p>
        </div>
```

The message about the email will be displayed only if user touched the control field and the email is not valid.

Accessing form values

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  @ViewChild('f') form: NgForm;

  user = {
    username: '',
    email: '',
  };

  onSubmit() {
    this.user.username = this.form.value.username;
    this.user.email = this.form.value.email;
  }
}
```

Despite being simpler, template-driven forms are flexible and powerful enough for many use cases. However, for complex forms, reactive forms may offer more control and predictability.

2. Explain reactive forms in Angular

Reactive forms are:
- good for complex forms
- unit testable
- give you more control
-

Templates in template-driven approach are implicitly transferred to a model. In reactive forms approach you can directly access FormGroup members (FormControl field) and you explicitly declare model.

Reactive forms in Angular are a model-driven approach to handling form inputs whose values change over time. Unlike template-driven forms, which are more suitable for static forms, reactive forms provide more predictability and scalability, especially for complex forms.

Setting Up Reactive Forms

To use reactive forms, you need to import ReactiveFormsModule from @angular/forms in your module:

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
 imports: [
 ReactiveFormsModule
 ],
 ...
})
export class AppModule { }
```

Next, you can create a form group in your component class. FormGroup represents a group of FormControl instances, FormControl tracks the value and validation status of an individual form control.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  genders = ['male', 'female'];
  signupForm: FormGroup;

  ngOnInit(): void {
    this.signupForm = new FormGroup({
      username: new FormControl(null),
      email: new FormControl(null),
      gender: new FormControl('male'),
    });
  }
}
```

Now we need to bind it to the template. You have to use property binding in the form which is called 'formGroup'. Then for each of the inputs you need to pass corresponding FormControl name like this -> formControlName="username"

```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
      <form [formGroup]="signupForm">
```

```
        <div class="form-group">
          <label for="username">Username</label>
          <input
            formControlName="username"
            type="text"
            id="username"
            class="form-control"
          />
        </div>
            ...
            Other inputs
            ...
            ...
            ...
      </form>
    </div>
  </div>
</div>
```

Handling Form Submission

To handle form submission, you can define a method in your component class and bind it to the form. You cannot use some references like in the template approach. And you don’t need to because you created all of the controls in the component class. You just need to bind the onsubmit method to onSubmit event in the form in the template.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  genders = ['male', 'female'];
  signupForm: FormGroup;

  ngOnInit(): void {
    this.signupForm = new FormGroup({
      username: new FormControl(null),
      email: new FormControl(null),
      gender: new FormControl('male'),
    });
  }

  onSubmit(): void {
    console.log(this.signupForm);
  }
}
```

```
<div class="container">
  <div class="row">
    <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
      <form (ngSubmit)="onSubmit()" [formGroup]="signupForm">
        <div class="form-group">
          <label for="username">Username</label>
          <input
            formControlName="username"
            type="text"
            id="username"
            class="form-control"
          />
        </div>
            ...
            Other inputs
            ...
            ...
            ...
      </form>
    </div>
  </div>
</div>
```

Adding Validation

Reactive forms support complex validation through the use of Validators. You only do this in the class component and not in the template and you just pass a Validator or an array of Validators to the FormControl.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent implements OnInit {
  genders = ['male', 'female'];
  signupForm: FormGroup;

  ngOnInit(): void {
    this.signupForm = new FormGroup({
      username: new FormControl(null, Validators.required),
      email: new FormControl(null, [Validators.required, Validators.email]),
      gender: new FormControl('male'),
    });
  }

  onSubmit(): void {
    console.log(this.signupForm);
  }
}
```

When it comes to printing messages when the state of the form changes you have to use *ngIf and then access the form you declared in the class component. You cannot use reference like you did in the template approach.

```
  <form (ngSubmit)="onSubmit()" [formGroup]="signupForm">
        <div class="form-group">
          <label for="username">Username</label>
          <input
            formControlName="username"
            type="text"
            id="username"
            class="form-control"
          />
          <p
            *ngIf="
              !signupForm.get('username').valid &&
              signupForm.get('username').touched
            "
          >
            Please enter a valid username
          </p>
        </div>
```

You can also create your custom validators. You just need to edit the class component like that: (look at forbiddenNamesValidator and username FormControl)

```
export class AppComponent implements OnInit {
  genders = ['male', 'female'];
  signupForm: FormGroup;
  forbiddenUsernames = ['Alex', 'David'];

  ngOnInit(): void {
    this.signupForm = new FormGroup({
      username: new FormControl(null, [
        Validators.required,
        this.forbiddenNamesValidator.bind(this),
      ]),
      email: new FormControl(null, [Validators.required, Validators.email]),
      gender: new FormControl('male'),
    });
  }

  onSubmit(): void {
    console.log(this.signupForm);
  }

  forbiddenNamesValidator(control: FormControl): { [s: string]: boolean } {
    if (this.forbiddenUsernames.indexOf(control.value) !== -1) {
      return { nameIsForbidden: true };
    }
    return null;
  }
}
```

3. Discuss the pros and cons on the different approaches

Both template-driven and reactive forms in Angular have their own advantages and disadvantages, and the choice between them depends on the specific requirements of your application.

Template-Driven Forms

Pros:

- Simpler and quicker to set up, especially for simple forms.
- Ideal for beginners due to their simplicity.
- Familiar to AngularJS developers, making migration easier.

Cons:

- Limited in terms of flexibility, scalability and control.
- Validation logic is mixed with the template, making the template messy.
- Not suitable for complex forms.

Reactive Forms

Pros:

- More robust and scalable, making them suitable for complex forms.
- Provide a clear separation between business logic and presentation logic, leading to cleaner and more maintainable HTML templates.
- Easier to create custom validators.
- Allows for better testing since the form model is defined in the component class.
-

Cons:

- Requires more coding compared to template-driven forms.
- Might be more difficult to understand for beginners due to its complexity.

In summary, if you're dealing with simple forms or you're new to Angular, template-driven forms might be a good starting point. But for complex forms or when you need more control and scalability, reactive forms are generally the better choice.

4. Explain how to test Angular apps

Testing is an integral part of Angular development, ensuring the functionality, performance, and behavior of your application. Angular provides built-in support for testing, making it easy to script and run tests for your apps.

Types of Testing in Angular

There are three main types of tests you can perform in Angular:

Unit Tests: These validate individual units of code. They are usually written for components, services, and pipes.
Integration Tests: These validate the interaction between different units of code.

End-to-End (E2E) Tests: These validate the application as a whole, simulating real user scenarios.

Writing Tests

To write a test, you typically create a .spec.ts file corresponding to the file you want to test. For example, if you want to test a component, you would create a my-component.component.spec.ts file. Inside this file, you can write tests using Jasmine or Jest.

Here's an example of a simple unit test for a component:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { MyComponent } from './my-component.component';

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

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

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

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

In this example, beforeEach is used to set up the testing environment before each test is run. The it function defines a test, and the expect function is used to make assertions about the component.

Running Tests

Tests can be run using the Angular CLI with the ng test command. This command uses Karma as the test runner and Jasmine as the testing framework. The results of the tests are displayed in the terminal and in the browser 2.

Best Practices

Some best practices for Angular testing include:

Writing tests that are independent and isolated.
Using beforeEach and afterEach to set up and tear down test data.
Using the expect function to make assertions.
Regularly running tests as part of the development process.

Conclusion
Testing is an essential part of software development that helps catch bugs early and ensure the quality of the application. Whether you're writing unit tests, integration tests, or end-to-end tests, the goal is always to ensure that your application behaves as expected 4.

Question 5

1. Explain and discuss responsive web design (Flexbox, Grid and media queries)

Responsive web design is crucial in today's multi-device world. It ensures that websites work well on any device, from smartphones to large desktop monitors, by designing websites so that they automatically adjust their layout, images, and content to fit the screen size and resolution of the device being used.

Flexbox

Flexbox is a CSS layout module that makes it easy to design flexible responsive layout structure without using float or positioning. With Flexbox, you can easily manage space distribution between items in a container and align items vertically, horizontally, or along both axes [Source 0](https://www.w3schools.com/css/css3_flexbox_responsive.asp).

Here's an example of how you can use Flexbox to create a responsive layout:

```css
.container {
 display: flex;
 flex-wrap: wrap;
}

.item {
 flex: 1 0 20%; /* grow, shrink, basis */
}
```

In this example, the container becomes a flex container, and the items become flex items. The `flex` property is a shorthand for `flex-grow`, `flex-shrink`, and `flex-basis` combined. The second and third parameters (`flex-shrink` and `flex-basis`) are optional. Default is `0 1 auto`.

Grid

CSS Grid is another powerful tool for creating responsive designs. It allows you to create complex layouts with ease. With Grid, you can specify the size and position of elements in a two-dimensional layout system.
Here's an example of how you can use CSS Grid to create a responsive layout:

```css
.container {
 display: grid;
 grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
}
```

In this example, the container becomes a grid container, and the `grid-template-columns` property defines the number and size of columns in the grid. The `repeat()` function is used to create a responsive number of columns, and the `minmax()` function is used to specify the minimum and maximum size of each column.

Media Queries

Media queries are a fundamental part of responsive design. They allow you to apply different styles depending on the characteristics of the device displaying the webpage, such as its screen width, orientation, and resolution.

Here's an example of how you can use media queries to create a responsive design:

```css
body {
 background-color: lightblue;
}

@media (min-width: 600px) {
 body {
 background-color: lightgreen;
 }
}
```

In this example, the background color of the body is `lightblue` by default. However, when the viewport is at least `600px` wide, the background color changes to `lightgreen`.

Conclusion

Flexbox, Grid, and media queries are powerful tools for creating responsive designs in CSS. While Flexbox is great for layouts in one dimension, Grid excels in two dimensions. Media queries are necessary for applying different styles for different devices. Together, they enable you to create websites that look great on any device.

2. Explain how third-party libraries can be used to develop web applications (examples: NgRx, Tailwind, and Angular Material)

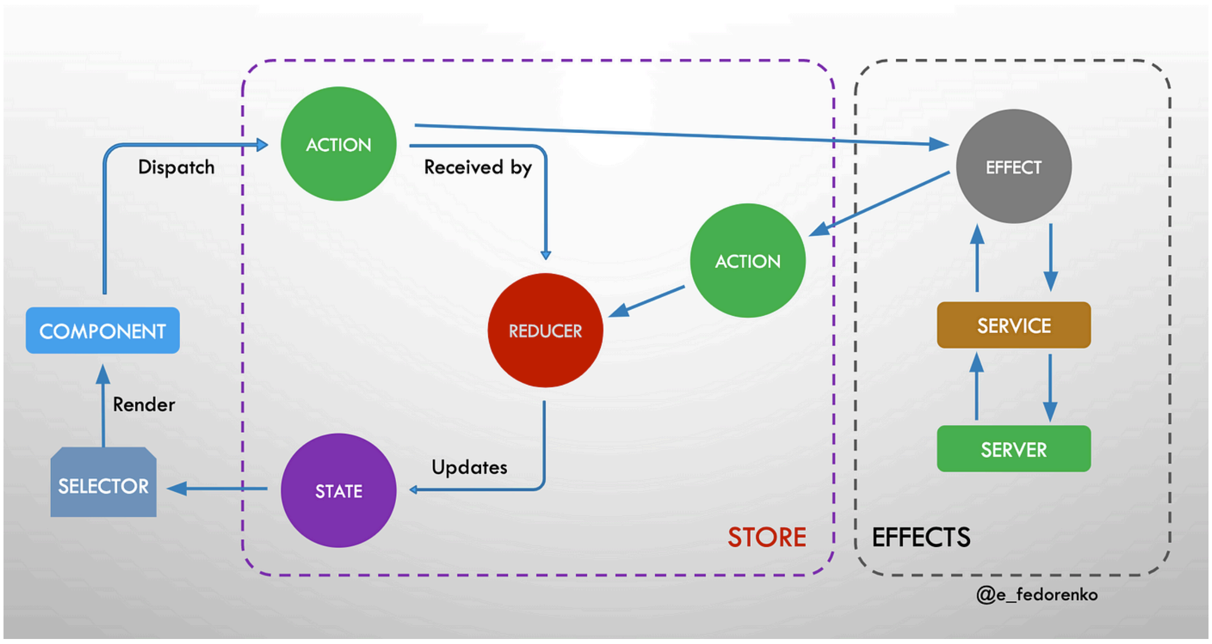
Third-party libraries in Angular can greatly enhance the functionality and development speed of your applications. They provide pre-built components, patterns, and tools that can be easily integrated into your projects, allowing you to focus on writing your application logic. Here are examples of three popular third-party libraries: NgRx, Tailwind, and Angular Material.

NgRx

NgRx is a library for managing state in Angular applications. It provides a Redux-style architecture using RxJS Observables. With NgRx, you can manage global state, handle side effects, and isolate state management logic from your components. Here's an example of how you can use NgRx to manage state in an Angular application:

```
import { StoreModule } from '@ngrx/store';
import { reducers } from './reducers';

@NgModule({
 imports: [StoreModule.forRoot(reducers)]
})
export class AppModule { }
```

In this example, the StoreModule.forRoot() method is used to initialize the store with the root reducer.



Tailwind

Tailwind CSS is a utility-first CSS framework that provides low-level utility classes to build custom designs. It doesn't provide pre-designed components like Bootstrap or Angular Material, but instead gives you the building blocks to create your own unique design. Here's an example of how you can use Tailwind to style an Angular component:

```
<div class="flex justify-center items-center h-screen bg-gray-200">
 <h1 class="text-4xl text-blue-500">Hello, Tailwind!</h1>
</div>
```

In this example, the flex, justify-center, items-center, h-screen, bg-gray-200, text-4xl, and text-blue-500 classes are used to style the div and the heading 2.

Angular Material

Angular Material is a UI component library that implements Material Design principles. It provides a wide range of pre-styled components, such as buttons, cards, checkboxes, dialogs, and sliders. Here's an example of how you can use Angular Material to add a button to an Angular component:

```
<button mat-raised-button color="primary">Primary Button</button>
```

In this example, the mat-raised-button directive is used to create a raised button, and the color attribute is used to set the button's color 5.

In conclusion, third-party libraries can significantly enhance your Angular development process. They can save you time and effort, provide pre-built components and patterns, and help you write cleaner and more maintainable code.

Question 6

1. Explain the concept Progressive Web Apps.

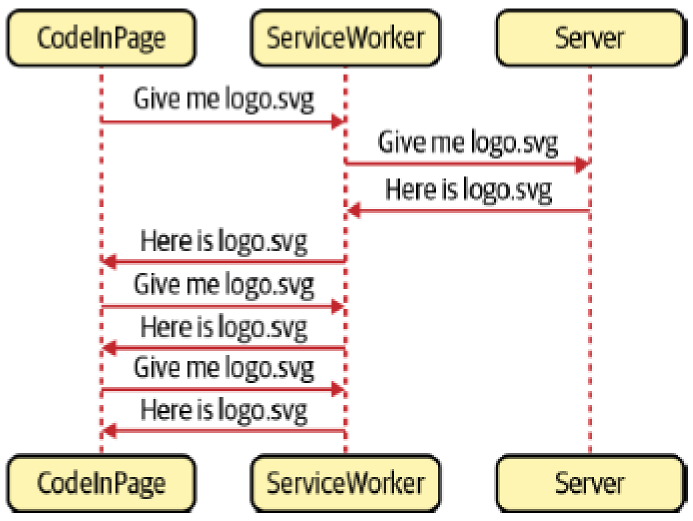
Progressive Web Apps (PWAs) are a type of application software that's designed to work on standard web browsers like Chrome, Safari, Firefox, and Edge. They are essentially web applications that utilize modern web capabilities to provide an app-like experience to users. This means they can work offline, send push notifications, and even be installed directly onto the user's device.

Working offline:

Service workers: service workers in the browser are like a proxy between the browser and the server of the application. They are responsible for managing caching, loading data (even offline, using cached data and so-called app *manifest.json* file) using a promise-based approach. Service workers can intercept network requests, check if the requested resources are available in the cache, and serve them if they are. If the resources are not in the cache, the service worker can fetch them from the network and store them for future use. They run on different thread than the main javascript of the app, so they can perform even heavy calculations, not disturbing the application UX.
Most browsers nowadays support the usage of service workers.
If we want out app to use a service worker, we need to put a code to register it into a browser (Angular has a package for that).

“A service worker is a script written in javascript that allows intercepting and control of network requests and asset caching from the web browser. With service workers, web developers can create reliably fast web pages and offline experiences. It’s a browser feature but it doesn't have access to the DOM, its role is different than a normal JS script.”

Service worker in operation



IndexedDB: IndexedDB is a low-level API for client-side storage of structured data. It enables PWAs to store and manage large amounts of data offline, such as product catalogs, user profiles, or shopping carts. Developers can use IndexedDB to create, read, update, and delete records in the database, even when the user is offline. When the user regains connectivity, the PWA can synchronize the offline data with the server.

There are 3 strategies to fetching data in PWA:
https://blog.bitsrc.io/5-service-worker-caching-strategies-for-your-next-pwa-app-58539f156f52
Cache-first: this strategy prioritizes serving content from the cache, only falling back to the network when the requested content is not available in the cache. This strategy is suitable for static assets, such as images, stylesheets, and scripts, which don’t change frequently. It ensures faster load times and a smooth offline experience.

Network-first: this strategy attempts to fetch content from the network first, and if that fails, it falls back to the cache. This strategy is ideal for dynamic content that changes frequently, such as news articles, user-generated content, or real-time data. It ensures that users always get the most up-to-date content when they have connectivity.

Stale-while-revalidate: this strategy serves content from the cache while simultaneously updating the cache with the latest content from the network. This approach provides a balance between speed and freshness, making it suitable for content that is relatively static but occasionally updated, such as product listings or blog posts.

Installation:

A PWA should have either a pop-up prompting user to install the app (like reddit has on mobiles) or it may be "added to home screen" through browser settings, either on mobile or desktop. When a user chooses this option, the PWA should be installed without the need to use any app store - just directly from the browser.
For instance: on Android, after clicking "install", an icon should appear in the home screen (with a logo + a title). When clicked, there is a loading screen with the logo and name, then the app content is loaded (either online or offline). There may optionally be no browser searchbar there.

So, how does the application know what logos, names, colors etc to use when installing?
A PWA needs a manifest.json file. It may look like this:

```
{
 "name": "My PWA",
 "short_name": "PWA",
 "theme_color": "#1976d2",
 "background_color": "#fafafa",
 "display": "standalone",
 "scope": "/",
 "start_url": "/",
 "icons": [
   {
     "src": "assets/icons/icon-72x72.png",
     "sizes": "72x72",
     "type": "image/png"
   },
  . . .
 ]
}
```

When installing the app, an appropriate to the screen size icon is chosen (if many provided - there should be at least 1 192x192 px icon there) and the "Short name" is put as an app title. Background color for loading screen is presented in the json as well. Indicated is also the starting page for the app and the way of displaying it (whether in a browser window, or as standalone app with no searchbar). Some additional properties may be set in this file (like screen orientation).

If you want to see if the page is PWA, you may go to Developer Tools -> Application -> (Manifest, Service Workers, Storage, IndexedDB, …)
You may also generate a Lighthouse report: Developer Tools -> Lighthouse -> Analyze page load. This will tell you much about all PWA aspects and features in the app.

A PWA does not have to meet 100% of the above requirements - it may be tailored to the company's needs, but that's the general idea.

Key Characteristics of PWAs **(kinda buzz words)**

1. **Progressive:** PWAs work for every user, regardless of browser choice, because they're built with progressive enhancement as a core tenet. This means they enhance the user

experience progressively, providing basic functionality to all users and adding advanced features as the browser supports them.

2. **Responsive:** PWAs fit any form factor, desktop, mobile, tablet, or whatever is next. They are designed to provide a consistent user experience across all devices.

3. **Connectivity Independent:** Thanks to service workers, PWAs can work offline or on low-quality networks. Service workers are scripts that your browser runs in the background, separate from a web page, opening the door to features that don't need a web page or user interaction.

4. **App-like:** PWAs use the app shell model to provide app-style navigations and interactions. An app shell is a minimal HTML, CSS, and JavaScript representation of the app UI, providing a consistent user experience across different pages.

5. **Fresh:** PWAs are always up-to-date thanks to the service worker update process. Even when the user is offline, the service worker can sync data when the network is available again.

6. **Safe:** PWAs are served via HTTPS to prevent snooping and ensure content hasn’t been tampered with.

7. **Discoverable:** PWAs are identifiable as “applications” thanks to W3C manifests and service worker registration scope, allowing search engines to find them.

8. **Re-engageable:** PWAs make re-engagement easy through features like push notifications.

9. **Installable:** Users can “keep” PWAs they find most useful on their home screen without the hassle of an app store

10. **Linkable:** PWAs can be easily shared via URL and not require complex installation.

Benefits of PWAs

The key advantage of PWAs is that they can provide a near-native app experience on the web, without the need to install a separate app on the user's device. This makes them fast, cost-effective, and easily discoverable. PWAs also benefit from continuous updates, enabling them to remain fresh and relevant. Furthermore, since PWAs are built with standard web technologies, they can be developed once and run on any device or channel.

2. Show how to implement a Progressive Web App with Angular or React (the student is free to chose which framework to use for explanation and demonstration of how to build a PWA)

Angular:

Let's discuss how to implement a Progressive Web App (PWA) using Angular. We'll use Angular CLI to create a new project, and then configure it to become a PWA.

Step 1: Create a New Angular Project

First, install Angular CLI globally on your machine if you haven't done so already:

```
npm install -g @angular/cli
```

Then, create a new Angular project:

```
ng new my-pwa
```

Step 2: Install Angular PWA Package

Next, navigate into your new project directory and install the @angular/pwa package:

```
cd my-pwa
npm install @angular/pwa --save
```

Step 3: Configure the Angular PWA Package

```
Import the ServiceWorkerModule in your app.module.ts:

import { ServiceWorkerModule } from '@angular/service-worker';

@NgModule({
 ...
 imports: [
   ...
   ServiceWorkerModule.register('ngsw-worker.js', { enabled:
environment.production })
```

```
  ],
  ...
})
export class AppModule { }
```

In this example, the service worker is registered only in production mode.

Step 4: Add a Web App Manifest

Create a manifest.json file in the src folder with the following content:

```
{
  "name": "My PWA",
  "short_name": "PWA",
  "theme_color": "#1976d2",
  "background_color": "#fafafa",
  "display": "standalone",
  "scope": "/",
  "start_url": "/",
  "icons": [
    {
      "src": "assets/icons/icon-72x72.png",
      "sizes": "72x72",
      "type": "image/png"
    },
   . . .
  ]
}
```

Then, link this manifest file in your index.html:

```
<link rel="manifest" href="/manifest.json">
```

Step 5: Generate Assets for Icons

Generate the icons for your PWA using the ng generate @angular/pwa:icons command. This will generate a variety of icon sizes for different devices and platforms.

Step 6: Build Your Application

Finally, build your application for production using the ng build --prod command. This will generate a dist/ folder with the built files, including the service worker.

Your Angular application is now configured as a PWA. It can be installed on the user's device, work offline, and provide a native app-like experience.

React:

Step 1: Create a New React Project

To create a progressive web application, we can do it by typing the following

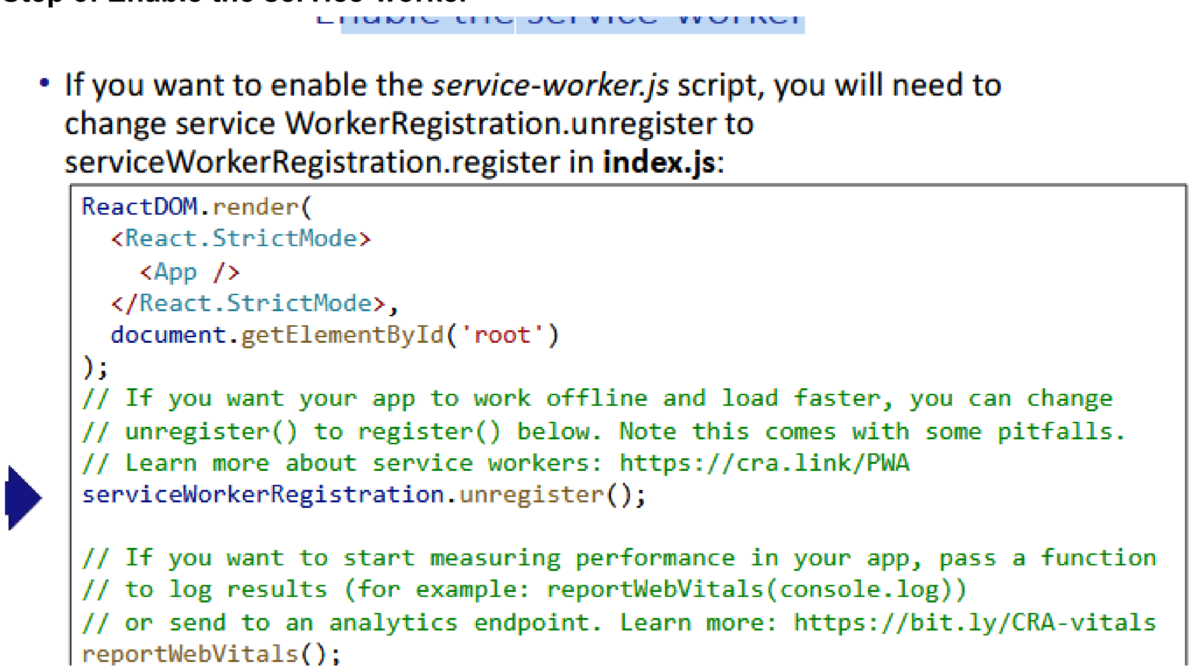
```
npx create-react-app appname --template cra-template-pwa
```

The application will look virtually the same as any other CRA application, but it will install several Workbox libraries.

Step 2: And add two source files

```
- service-worker.js.
- serviceWorkerRegistration.js
```

Step 3: Enable the service-worker



- If you want to enable the *service-worker.js* script, you will need to change service WorkerRegistration.unregister to serviceWorkerRegistration.register in **index.js**:

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://cra.link/PWA
serviceWorkerRegistration.unregister();

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Step 4: Customize serviceWorkerRegistration

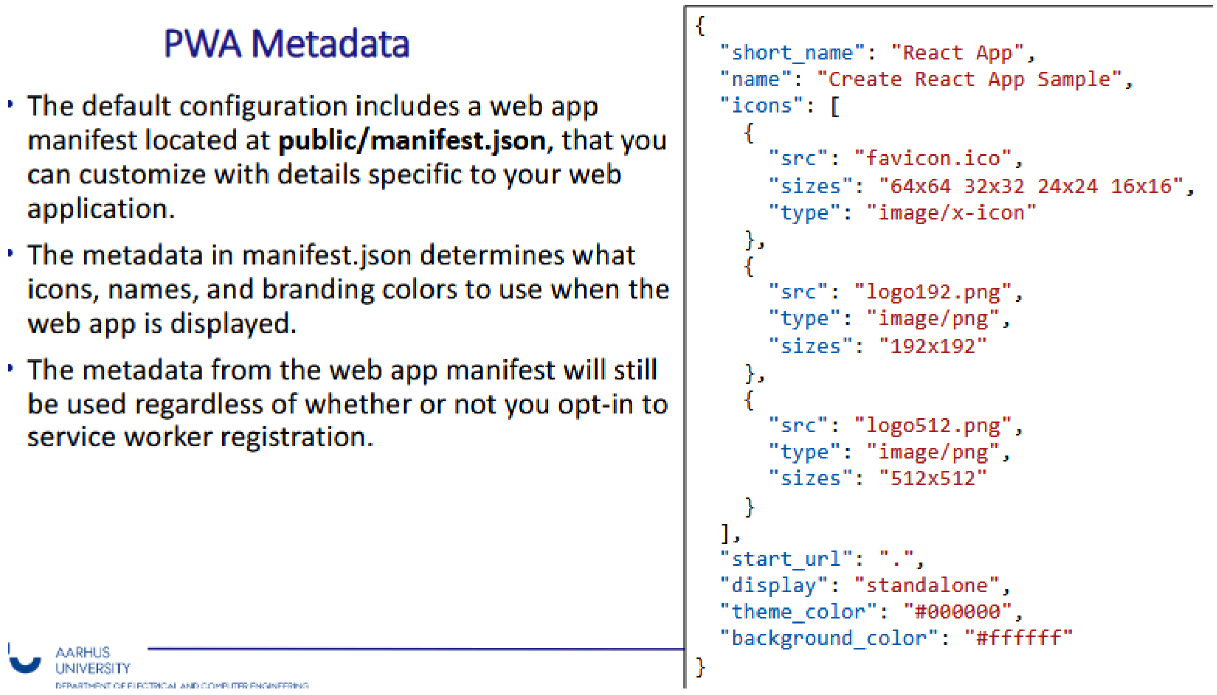
```
serviceWorkerRegistration.register({
  onInstall: (registration) => {
    console.log('Service worker installed')
  },
  onUpdate: (registration) => {
    console.log('Service worker updated')
  },
})
```

Step 5: Set caching strategy

- This strategy is helpful if you make frequent requests for a resource but don't care if you have the latest version.

```
import { registerRoute } from 'workbox-routing';
import { CacheFirst, StaleWhileRevalidate } from 'workbox-strategies';

// code missing

registerRoute(
    ({url}) => url.origin === 'https://fonts.googleapis.com',
    new StaleWhileRevalidate({
        cacheName: 'stylesheets',
    })
);
```

Step 6 - Complete manifest.json



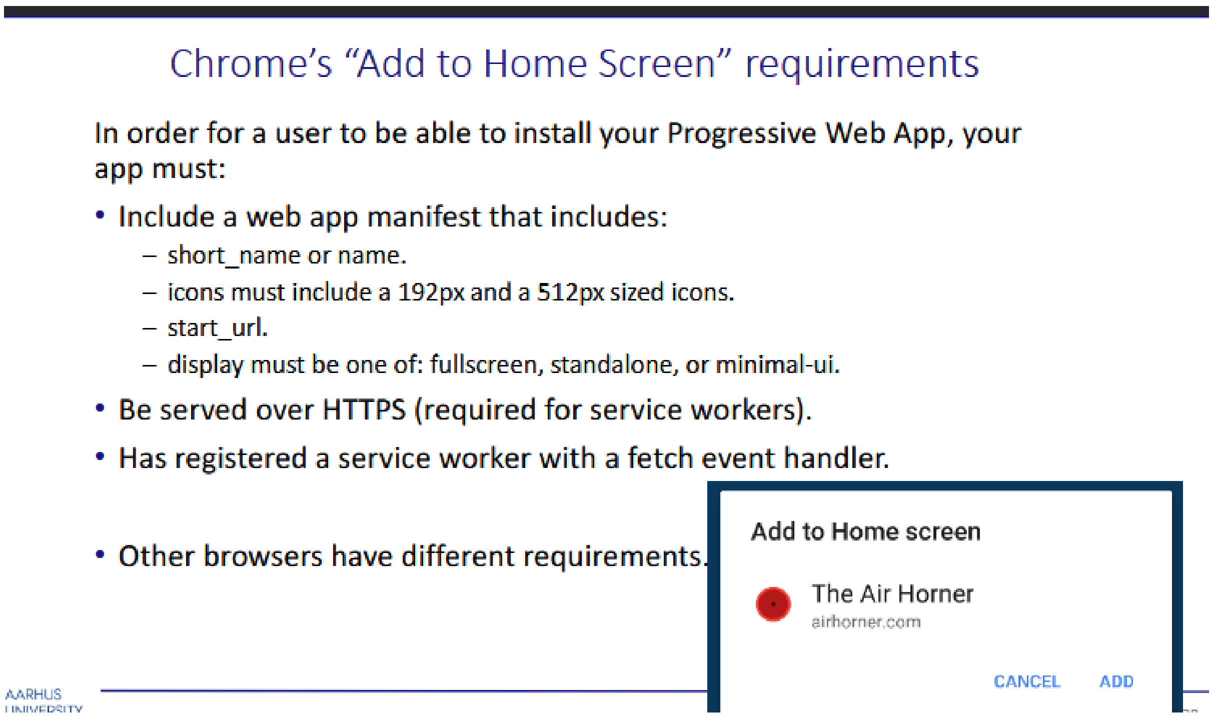
PWA Metadata

- The default configuration includes a web app manifest located at **public/manifest.json**, that you can customize with details specific to your web application.
- The metadata in manifest.json determines what icons, names, and branding colors to use when the web app is displayed.
- The metadata from the web app manifest will still be used regardless of whether or not you opt-in to service worker registration.

```
{
  "short_name": "React App",
  "name": "Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    },
    {
      "src": "logo192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "logo512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Step 7 - Make it compliant with the browser


Chrome's "Add to Home Screen" requirements

In order for a user to be able to install your Progressive Web App, your app must:

- Include a web app manifest that includes:
  - short_name or name.
  - icons must include a 192px and a 512px sized icons.
  - start_url.
  - display must be one of: fullscreen, standalone, or minimal-ui.
- Be served over HTTPS (required for service workers).
- Has registered a service worker with a fetch event handler.
- Other browsers have different requirements.

Question 7

1. Give an overview of Next.js.

Overview of Next.js

- Good for both developers, who can write in TypeScript, with linters and in a declarative way and for user experience, because Next is compiled to efficient bundle
- Next.js supports server-side rendering (SSR) and static site generation (SSG). In SSR, pages can be generated on each request, while in SSG, pages are generated at build time
- In Next.js in order to move an application from development to production the application code needs to be:
  - Compiled - *When preparing the application for production, Next.js performs a more comprehensive compilation. This involves transforming the source code (JavaScript and TypeScript) into a format that is optimized for execution by web browsers. Compilation might include transpilation from modern JavaScript features to versions compatible with a broader range of browsers*
  - Bundled - *Bundling is the process of resolving the web of dependencies and merging (or 'packaging') the files (or modules) into optimized bundles for the browser, with the goal of reducing the number of requests for files when a user visits a web page*
  - Minified - *Minification is the process of removing unnecessary code formatting and comments without changing the code's functionality. The goal is to improve the application's performance by decreasing* file sizes
  - Code split - *Next.js supports automatic code splitting in production. This means that different parts of the application (e.g., pages or components) are*

split into separate chunks. Only the necessary code for the initial page load is sent to the client. As users navigate through the application, additional code is loaded dynamically. Code splitting improves the initial loading time and reduces the amount of code that needs to be downloaded before the user can start interacting with the application.

- Rendering - Is the process of converting the code you write in React into the HTML representation of your UI. Rendering can take place on the server or on the client. It can happen either ahead of time at build time, or on every request at runtime.
    - With Next.js, three types of rendering methods are available:
        - Server-Side Rendering
        - Static Site Generation
        - Client-Side Rendering

Next.js offers three rendering methods: Server-Side Rendering (SSR) generates HTML on each request, shows it to the user and then uses a hydration ***process*** *to make it interactive, ideal for dynamic and personalized content. Static Site Generation (SSG) pre-renders HTML at build time, suitable for content with infrequent changes like blogs. Client-Side Rendering (CSR) delivers minimal content initially, with client-side JavaScript fetching and rendering additional content at runtime, making it great for interactive single-page applications. Developers can use a hybrid approach, employing different rendering methods for various pages in the same application.*

2. Explain the different types of components in Next, and how mix them.

Next.js is a React framework that offers several types of components, each serving a different purpose. Let's discuss the different types of components and how to mix them:

Server Components

Server Components in Next.js allow you to write UI that can be rendered and optionally cached on the server. This enables streaming and partial rendering, and is particularly useful for parts of your application that need to fetch data, access your database, or interact with backend services.

Client Components

Client Components are the traditional React components that run in the browser. They can contain state, effects, and other React features. By default, Next.js uses Server Components, but you can opt into using Client Components when needed.

Mixing Components

You can mix Server and Client Components in various ways. One common pattern is passing a Server Component as a child or prop of a Client Component. This allows the

Server Component to be rendered on the server, well before the Client Component is rendered on the client.

Here's an example:

```
import ClientComponent from './client-component'
import ServerComponent from './server-component'

// Pages in Next.js are Server Components by default
export default function Page() {
 return (
  <ClientComponent>
    <ServerComponent />
  </ClientComponent>
 )
}
```

In this example, `<ClientComponent>` and `<ServerComponent>` are decoupled and can be rendered independently. The child `<ServerComponent>` can be rendered on the server, well before `<ClientComponent>` is rendered on the client.

Note: The pattern of "lifting content up" has been used to avoid re-rendering a nested child component when a parent component re-renders. You're not limited to the `children` prop. You can use any prop to pass JSX

In **conclusion**, Next.js provides flexibility in choosing what parts of your application should be rendered on the server or the client, and how to combine them. This allows you to optimize performance, reduce loading times, and create a seamless user experience.

3. Explain how to use the App router, and how it is different from the Pages router.

In Next.js, there are two primary approaches: the App Router - newer one and the Pages Router - the older one.

Pages router	**App router**
File & folder based - you can define your paths by both folder names and file names inside those folders	Folder names define paths and filenames are pre-defined. You have to for instance use ‘page.js’ reserved name.
For server-side/static rendering you use getStaticProps() and getServerSideProps() methods	You just use the fetch function with the await keyword and because all of the components are by default server-side.
SEO meta tags must be manually inserted into the Head component	you can use generateMetadata() built-in function to specify title, openGraph, description etc..

you can define backend API with handler function:
 	
export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>
) {
	you can also define backend API with functions for each HTTP verb:
	
	export async function GET() {
	  const res = await fetch('https://data.mongodb-api.com/...', {
	        headers: {
	        'Content-Type': 'application/json',
	        'API-Key': process.env.DATA_API_KEY,
	        },
	  })
	  const data = await res.json()
	
	  return Response.json({ data })
	}

HTML structure is pre-rendered on the server, but to add data to it you must use getServerSideProps() or getStaticProps()	Components are by default rendered and processed on the server until you use ‘use client’ directive.
Server components are only top-level ones.	Client components can be nested inside of server components, but not the other way around. If you want to pass the server component into the client component you have to pass it as props.

Question 8

1. Explain what Search engine optimization is. Why and when it is important.

The goal of SEO is to create a strategy that will increase your rankings position in search engine results. The higher the ranking, the more organic traffic to your site, which ultimately leads to more business for you. SEO is the key to increased conversion and confidence in your brand. It’s s key to many businesses for three reasons:

• Qualitative

– Increased chance that visitors turn into customers

• Trustable

– Higher confidence in your brand or mission

• Low-Cost

– Aside from the time and effort spent, having good SEO practices that result in higher search engine ranking is free

– There is no direct cost to appear in top organic search results positions

Three Pillars of Optimization

1. Technical

– Optimize your website for crawling and web performance

2. Creation

– Create a content strategy to target specific keywords

3. Popularity

– Boost your site's presence online so search engines know you are a trusted

– This is done using backlinks – third-party sites that link back to your site

2. What affects SEO ranking?

- **The most important thing for SEO is that page data and metadata is available on page load without JavaScript**. Static Site Generation, SSG or Server-Side Rendering, SSR are your best options

- **URL Structure is an important part of an SEO strategy**
  – It's best to use URLs that are semantic, meaning that they use words instead of IDs or random numbers – /learn/basics/create-nextjs-app is better than /learn/course-1/lesson-1

- **Metadata - Is the abstract of the website's content and is used to attach a title, a description, and an image to the site**
  - Title - The title tag is one of the most important SEO elements:
    It's what users see when they click to enter your website from search results

    - Description - is not taken into account for ranking purposes, but it can affect your click-through-rate on search results

- **On Page SEO** - Refers to the headings and links that make up the overall structure of the page– Headings indicate importance in the document. Links connect the web together. It's recommended to use the H1 heading tag in each page. H1 should represent what the page is about and be similar to your title tag.

- **Off-Page SEO:** Building external signals to enhance a website's authority, such as backlinks from reputable websites and social media signals.

- **Links** – Websites that receive more links tend to represent websites that are more trusted by users– Google started this principle with the invention of the PageRank Algorithm - The PageRank algorithm, at a high level, is an algorithm that goes through
every link on a database and scores domains based on how many links they receive (quantity) and from which domains (quality)

- **Content Quality:** Creating high-quality, valuable, and engaging content that meets the needs of users.
- **The Open Graph Protocol (OGP**) is a meta tag-based protocol that enhances how a website's content appears on social media platforms. By providing structured data, OGP allows for a visually appealing and consistent presentation of shared links, improving user engagement and maintaining brand consistency. Implementation involves adding specific meta tags to the HTML head section of web pages.

Here are some of the common Open Graph Protocol meta tags:
- og:title: The title of the content.
- og:type: The type of content (e.g., article, video, website).
- og:image: An image representing the content.
- og:url: The canonical URL of the content.
- og:description: A brief description of the content.

3. What are Core Web Vitals? And why are they important?

Core Web Vitals are a set of three metrics that measure the speed, interactivity, and visual stability of a webpage. They are considered to be important ranking factors for search engine optimization (SEO), and improving them can help your website rank higher in search results and provide a better user experience.

The three Core Web Vitals are:

Largest Contentful Paint (LCP): This measures how long it takes for the largest content element on a page to load. LCP should ideally be less than 2.5 seconds to provide a good user experience.
Cumulative Layout Shift (CLS): This measures how much a page’s layout shifts unexpectedly as it loads.Is a measure of your site’s overall layout stability. CLS should ideally be less than 0.1 for a good user experience.

First Input Delay (FID): This measures how long it takes for the browser to respond to a user’s first interaction with a page, such as a click or a tap. FID should ideally be less than 100 milliseconds for a good user experience. Unlike FID, Time to Interactive only tells us how long it took for the page to become interactive, not the time to respond to user interaction.

Core Web Vitals are important because they contribute to the overall user experience of your website, which is a key factor in Google's ranking algorithm. Google plans to make page experience an official Google ranking factor. Page experience will be a combination of factors that Google considers important for user experience, including Core Web Vitals. Although a great page experience score won't magically push you to the top spot in Google, it is still an important consideration. There's no need to panic if your site's Core Web Vitals score isn't perfect. Google has given site owners until next year to improve their site’s Core Web Vital scores.

However, it's worth noting that while Core Web Vitals are important, they are not a significant ranking factor. Google has over 200 ranking factors, many of which don’t carry much weight. When talking about Core Web Vitals, Google reps have referred to these as tiny ranking factors or even tiebreakers. That said, there have been studies that found some positive correlation between passing Core Web Vitals and better rankings, but these results should be taken with skepticism.

In **conclusion**, while Core Web Vitals are not the only factor Google considers when ranking websites, they are an important aspect of the user experience that can contribute to improved visibility in search results.

4. Give an overview of Next.js and explain how to improve SEO in a Next Web app.

Overview of Next.js

- Good for both developers, who can write in TypeScript, with linters and in a declarative way and for user experience, because Next is compiled to efficient bundle
- Next.js supports server-side rendering (SSR) and static site generation (SSG). In SSR, pages can be generated on each request, while in SSG, pages are generated at build time
- In Next.js in order to move an application from development to production the application code needs to be:
    - Compiled - *When preparing the application for production, Next.js performs a more comprehensive compilation. This involves transforming the source code (JavaScript and TypeScript) into a format that is optimized for execution by web browsers. Compilation might include transpilation from modern JavaScript features to versions compatible with a broader range of browsers*
    - Bundled - *Bundling is the process of resolving the web of dependencies and merging (or ‘packaging’) the files (or modules) into optimized bundles for the*

    browser, with the goal of reducing the number of requests for files when a user visits a web page
    - Minified - *Minification is the process of removing unnecessary code formatting and comments without changing the code's functionality. The goal is to improve the application's performance by decreasing* file sizes
    - Code split - *Next.js supports automatic code splitting in production. This means that different parts of the application (e.g., pages or components) are split into separate chunks. Only the necessary code for the initial page load is sent to the client. As users navigate through the application, additional code is loaded dynamically. Code splitting improves the initial loading time and reduces the amount of code that needs to be downloaded before the user can start interacting with the application.*
- Rendering - Is the process of converting the code you write in React into the HTML representation of your UI. Rendering can take place on the server or on the client. It can happen either ahead of time at build time, or on every request at runtime.
    - With Next.js, three types of rendering methods are available:
        - Server-Side Rendering
        - Static Site Generation
        - Client-Side Rendering

Next.js offers three rendering methods: Server-Side Rendering (SSR) generates HTML on each request, shows it to the user and then uses a hydration ***process*** *to make it interactive, ideal for dynamic and personalized content. Static Site Generation (SSG) pre-renders HTML at build time, suitable for content with infrequent changes like blogs. Client-Side Rendering (CSR) delivers minimal content initially, with client-side JavaScript fetching and rendering additional content at runtime, making it great for interactive single-page applications. Developers can use a hybrid approach, employing different rendering methods for various pages in the same application.*

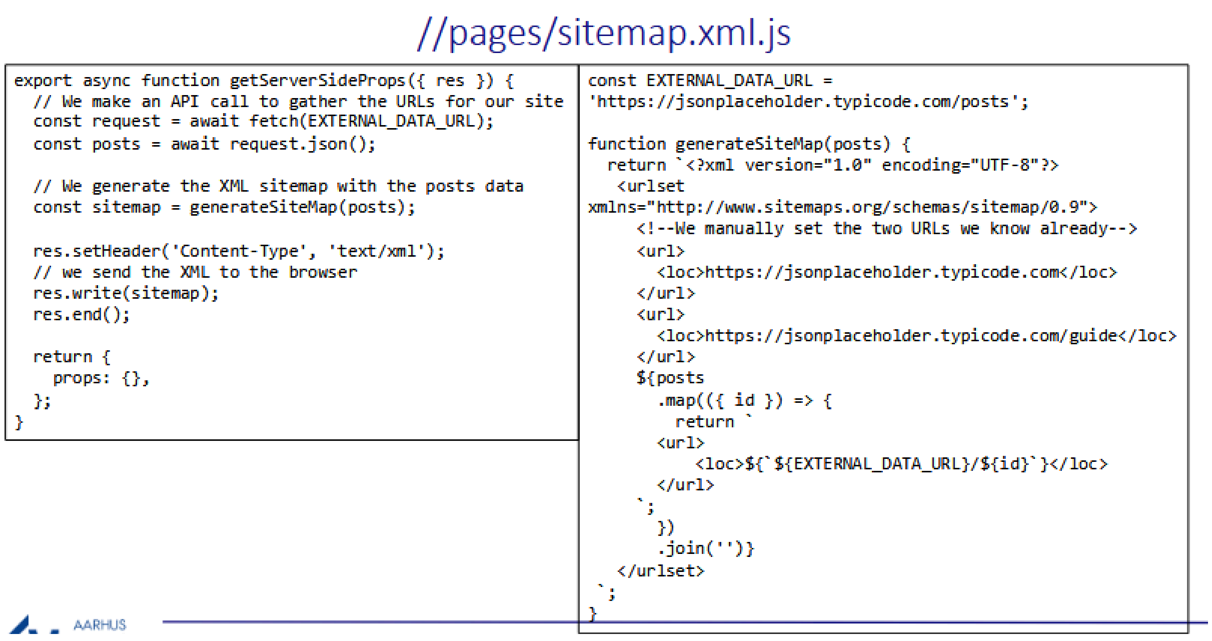
Improving SEO in Next.js

- Use SSR/SSG
- Use meta tags in the header

```
import Head from 'next/head';

function IndexPage() {
  return (
    <div>
      <Head>
        <title>Meta Tag Example</title>
        <meta name="google" content="nositelinkssearchbox" key="sitelinks" />
        <meta name="google" content="notranslate" key="notranslate" />
      </Head>
      <p>Here we show some meta tags off!</p>
    </div>
  );
}

export default IndexPage;
```

- Create a new file named `robots.txt` the public folder

- If site is large use XML Sitemap (You can do it manually by creating xml file or we fetch it from backend and use getServerSiteProps to generate it during next build phase)

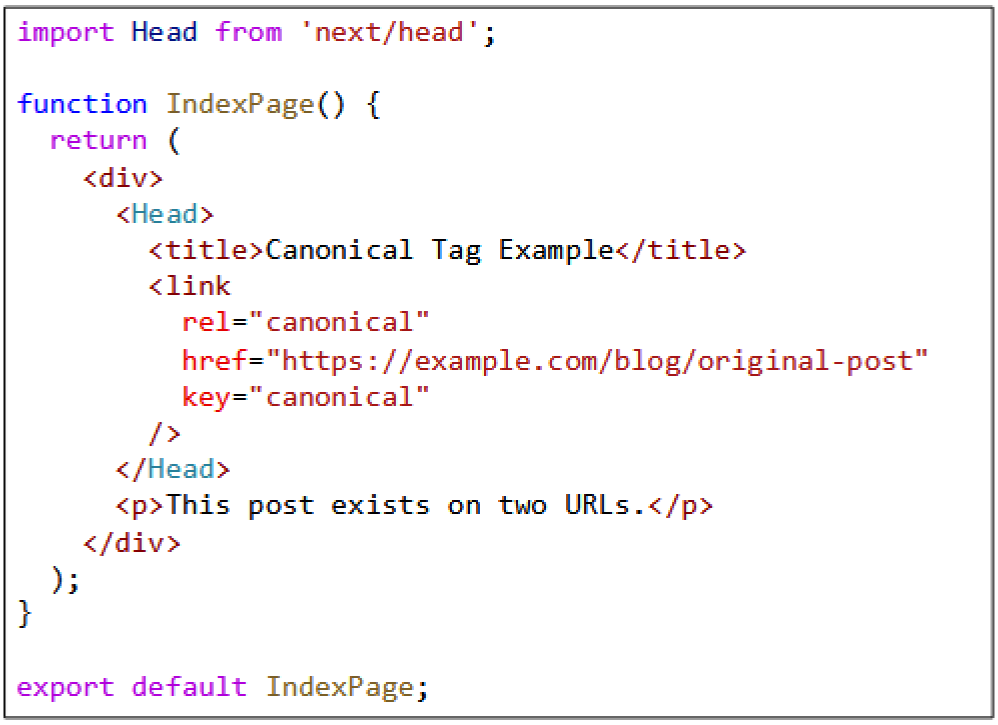


//pages/sitemap.xml.js

```
export async function getServerSideProps({ res }) {
  // We make an API call to gather the URLs for our site
  const request = await fetch(EXTERNAL_DATA_URL);
  const posts = await request.json();

  // We generate the XML sitemap with the posts data
  const sitemap = generateSiteMap(posts);

  res.setHeader('Content-Type', 'text/xml');
  // we send the XML to the browser
  res.write(sitemap);
  res.end();

  return {
    props: {},
  };
}
```

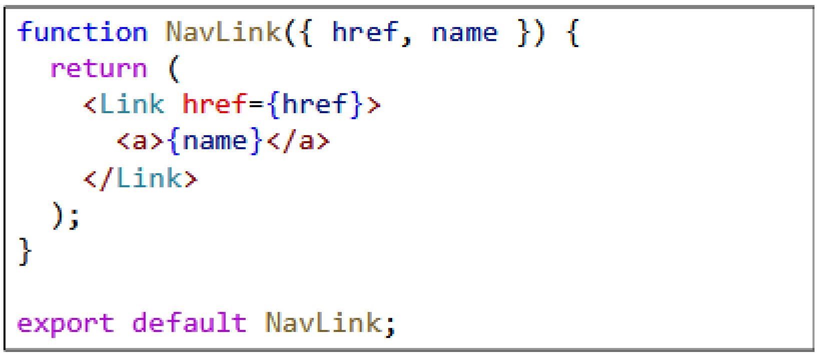
```
const EXTERNAL_DATA_URL =
'https://jsonplaceholder.typicode.com/posts';

function generateSiteMap(posts) {
  return `<?xml version="1.0" encoding="UTF-8"?>
   <urlset
xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
     <!--We manually set the two URLs we know already-->
     <url>
       <loc>https://jsonplaceholder.typicode.com</loc>
     </url>
     <url>
       <loc>https://jsonplaceholder.typicode.com/guide</loc>
     </url>
     ${posts
       .map(({ id }) => {
         return `
       <url>
           <loc>${`${EXTERNAL_DATA_URL}/${id}`}</loc>
       </url>
     `;
       })
       .join('')}
   </urlset>
 `;
}
```

- use h1 tag on pages
- use canonical url

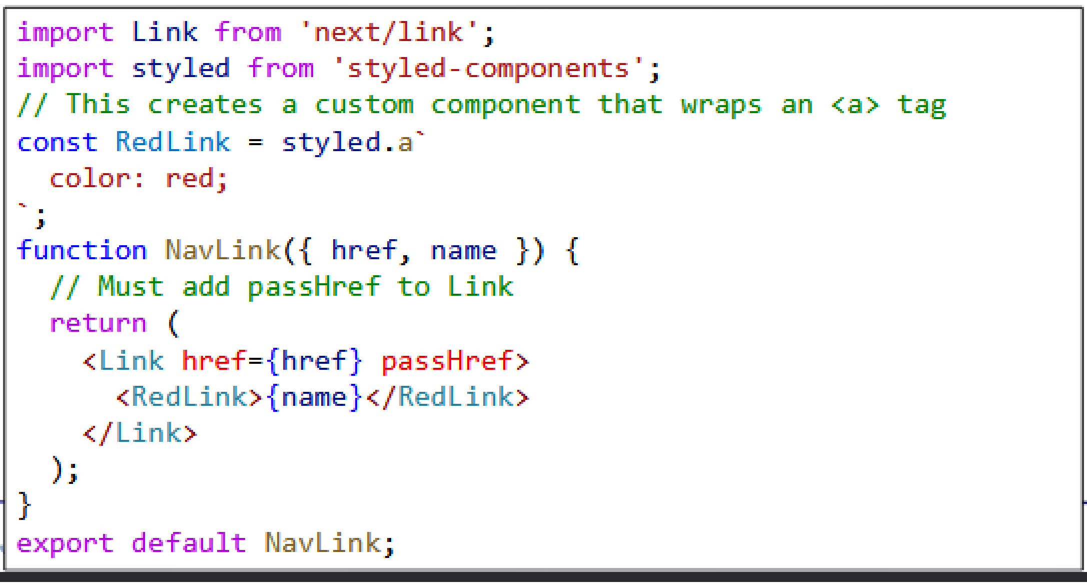


```
import Head from 'next/head';

function IndexPage() {
  return (
    <div>
      <Head>
        <title>Canonical Tag Example</title>
        <link
          rel="canonical"
          href="https://example.com/blog/original-post"
          key="canonical"
        />
      </Head>
      <p>This post exists on two URLs.</p>
    </div>
  );
}

export default IndexPage;
```

- use href and passHref tags to make sure tags have href attribute



```
function NavLink({ href, name }) {
  return (
    <Link href={href}>
      <a>{name}</a>
    </Link>
  );
}

export default NavLink;
```



```
import Link from 'next/link';
import styled from 'styled-components';
// This creates a custom component that wraps an <a> tag
const RedLink = styled.a`
  color: red;
`;
function NavLink({ href, name }) {
  // Must add passHref to Link
  return (
    <Link href={href} passHref>
      <RedLink>{name}</RedLink>
    </Link>
  );
}
export default NavLink;
```

Question 9

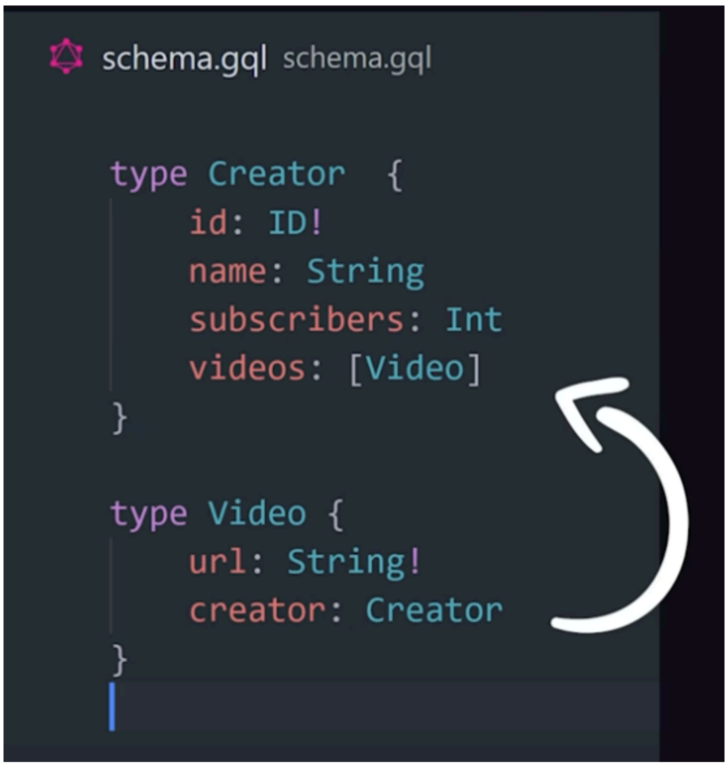
1. Explain the principles of GraphQL.

- It’s a specification published by Facebook which can be implemented in any programming language
- It’s best to use it in large applications, when you struggle with REST API, and you make multiple requests to backend in order to fetch relevant data
- GraphQL maps the relationships between objects in your database creating a graph. It’s a query language for traversing that map of relationships in your database
- By sending a query you are able to ask for specific data instead of making multiple requests. Consequently it’s more flexible.
- There is only one endpoint, not like in REST, where there are multiple endpoints
- Building a GraphQL API can be more intensive than building a REST API
- The flexibility and richness of the query language also adds complexity that may not be worthwhile for simple APIs
- Both REST and GraphQL use HTTP POST requests

Types of operations GraphQL

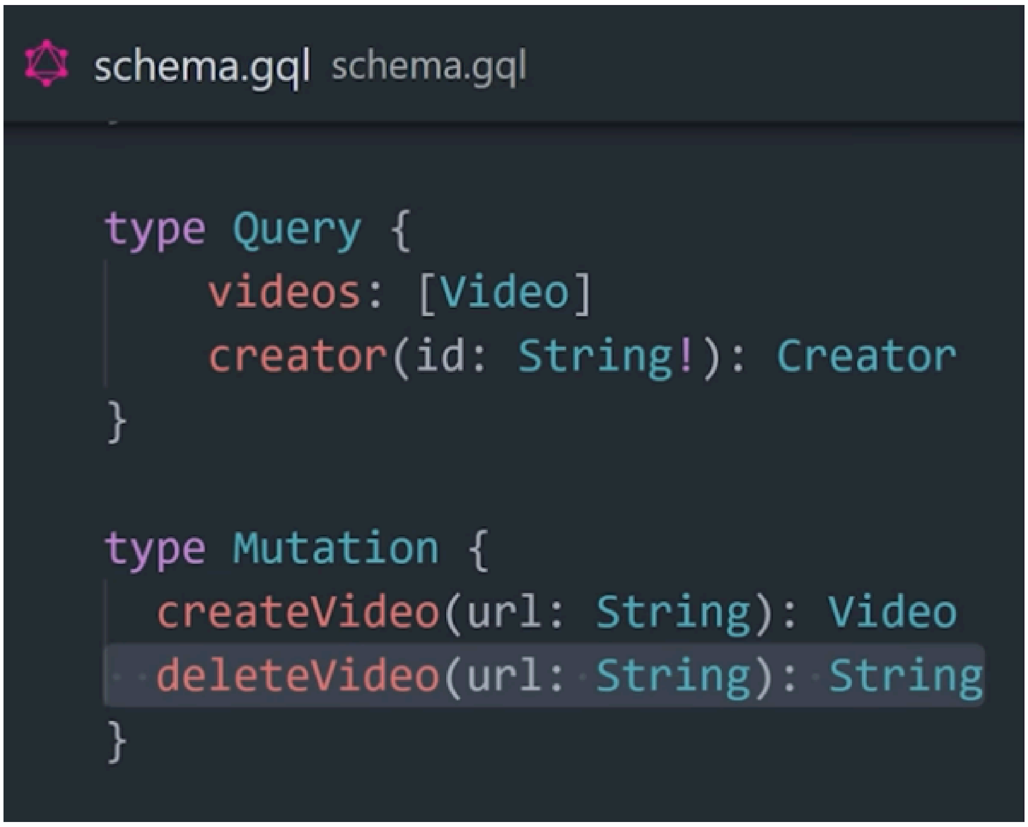
• Query operations
– A read-only fetch
• Mutation operations
– A write followed by a read
• Subscription operations
– A request for real-time data updates

To define GraphQL API you need to first define entities (models):



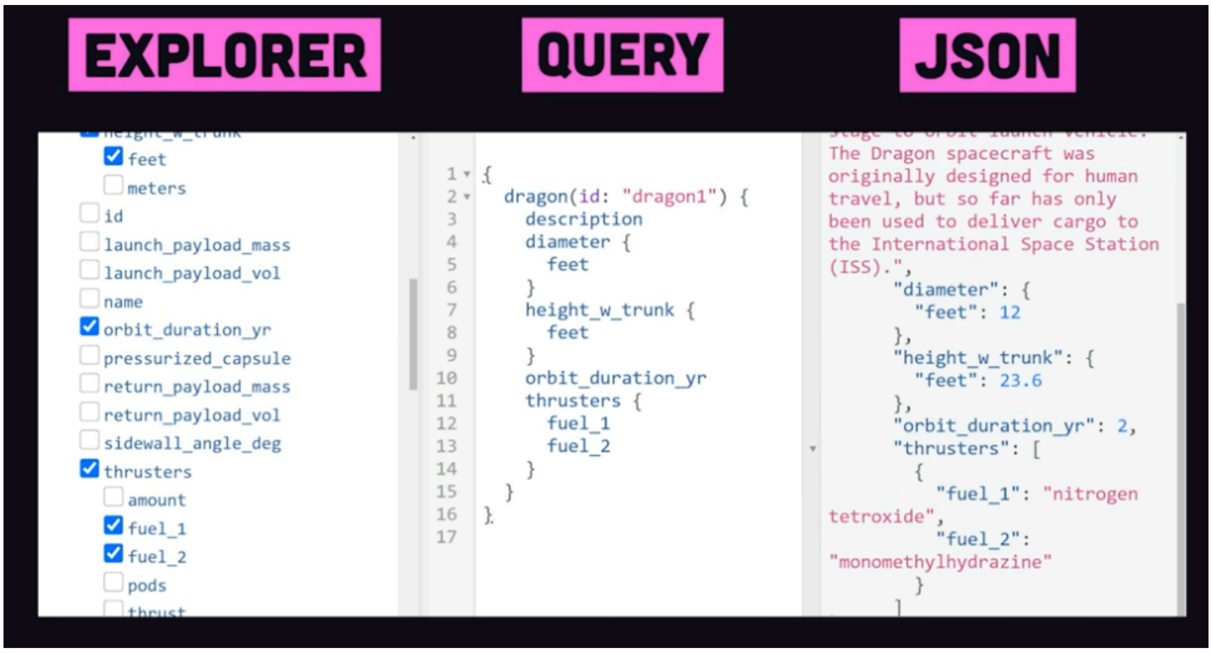
```
type Creator  {
    id: ID!
    name: String
    subscribers: Int
    videos: [Video]
}

type Video {
    url: String!
    creator: Creator
}
```

And then add Query type for read and Mutation type for data modification



schema.gql schema.gql

```
type Query {
    videos: [Video]
    creator(id: String!): Creator
}

type Mutation {
  createVideo(url: String): Video
  deleteVideo(url: String): String
}
```



```
  [✓] feet
  [ ] meters
[ ] id
[ ] launch_payload_mass
[ ] launch_payload_vol
[ ] name
[✓] orbit_duration_yr
[ ] pressurized_capsule
[ ] return_payload_mass
[ ] return_payload_vol
[ ] sidewall_angle_deg
[✓] thrusters
  [ ] amount
  [✓] fuel_1
  [✓] fuel_2
  [ ] pods
  [ ] thrust
```

```
{
  dragon(id: "dragon1") {
    description
    diameter {
      feet
    }
    height_w_trunk {
      feet
    }
    orbit_duration_yr
    thrusters {
      fuel_1
      fuel_2
    }
  }
}
```

```
The Dragon spacecraft was
originally designed for human
travel, but so far has only
been used to deliver cargo to
the International Space Station
(ISS).",
      "diameter": {
        "feet": 12
      },
      "height_w_trunk": {
        "feet": 23.6
      },
      "orbit_duration_yr": 2,
      "thrusters": [
        {
          "fuel_1": "nitrogen
tetroxide",
          "fuel_2":
"monomethylhydrazine"
        }
      ]
```

2. Discuss pros and cons of REST vs GraphQL.

REST		GraphQL	
Pros	Cons	Pros	Cons
simplicity	overfetching - getting more data then we need	fetched data can be limited through the body of the request	steep learning curve
scalability - easy caching mechanisms, which improve scalability	under fetching - making us send multiple requests to fetch all the data we need	under fetching- it is possible to merge multiple request to one request	caching challenges
	multiple endpoints	nesting queries	error handling
	necessity to add some filtering logic for getting precisely what we need from queries	single endpoint	
		type safety	
		simple structure (like json)	

3. Show how to access a GraphQL Web API from React.

Apollo Client is a comprehensive state management library for JavaScript that enables you to manage both local and remote data with GraphQL.

https://codesandbox.io/p/devbox/example-graphql-x7gzvn?file=%2Fsrc%2FApp.tsx

How to do it in react?

1. npm install @apollo/client graphql - **install**
2. **In app.js/app.ts**
    a. import { ApolloClient, InMemoryCache, ApolloProvider, gql } from '@apollo/client';
    b. const client = new ApolloClient({
     uri: 'https://flyby-router-demo.herokuapp.com/',
    cache: new InMemoryCache(),
    });
    c.
    ReactDOM.createRoot(document.getElementById("root") as HTMLElement).render(
      <React.StrictMode>
            <ApolloProvider client={client}>
            <App />

```
                    </ApolloProvider>
                </React.StrictMode>,
            );
```

3. **Then just use hook ‘useQuery’ and ‘gql’**

```
import { useQuery, gql } from "@apollo/client";
const GET_LOCATIONS = gql`
  query GetLocations {
        locations {
        id
        name
        description
        photo
        }
  }
`;

function App() {
  const { loading, error, data } = useQuery(GET_LOCATIONS);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error : {error.message}</p>;

  return data.locations.map(({ id, name, description, photo }) => (
        <div key={id}>
        <h3>{name}</h3>
        …
)
```

Updating cached query results:
Two strategies for this:
• Polling
– Provides near-real-time synchronization with your server by executing your query periodically at a specified interval.
• Refetching
– Refetching enables you to refresh query results in response to a particular user action, as opposed to using a fixed interval.

Polling

- To enable polling for a query, pass a pollInterval configuration option to the useQuery hook with an interval in milliseconds.

```
const { loading, error, data } =
useQuery(GET_DOG_PHOTO, {
    variables: { breed },
    pollInterval: 500,
  });
```

- You can also start and stop polling dynamically with the startPolling and stopPolling functions that are returned by the useQuery hook.

Refetching

- You can optionally provide a new variables object to the refetch function.
- If you don't, the query uses the same variables that it used in its previous execution.

```
export default function DogPhoto({ breed }) {
  const { loading, error, data, refetch } = useQuery(GET_DOG_PHOTO, {
    variables: { breed },
  });
  if (loading) return null;
  if (error) return `Error! ${error}`;
  return (
    <>
      <img src={data.dog.displayImage} alt="" style={{ height: 400, width: 400 }} />
      <button style={{ display: "block" }} onClick={() => refetch()}>Refetch!</button>
    </>
  );
```

4. Show how to access a GraphQL Web API from Next, and explain why it is different from doing the same in React.

Accessing a GraphQL Web API from Next.js is similar to doing so in React, as Next.js is built on top of React. However, Next.js provides server-side rendering (SSR) capabilities, which can affect how you fetch and handle data compared to traditional React applications.

Next with just fetch POST (you can do it both on client and server side):

```
export default async function Page() {
  const data = await fetch(
    "https://main--time-pav6zq.apollographos.net/graphql",
    {
      method: "POST",
      body: JSON.stringify({
        query: '{ now(id: "1") }',
      }),
      headers: {
        "Content-Type": "application/json",
      },
    }
  ).then((res) => res.json());

  return <main>{data.now}</main>;
}
```

Next with SSR and Apollo:

```
import { ApolloClient, HttpLink, InMemoryCache } from "@apollo/client";
import { registerApolloClient } from
"@apollo/experimental-nextjs-app-support/rsc";

export const { getClient } = registerApolloClient(() => {
  return new ApolloClient({
    cache: new InMemoryCache(),
    link: new HttpLink({
      uri: "https://main--spacex-l4uc6p.apollographos.net/graphql",
    }),
  });
});
```

```
import { getClient } from "@/lib/client";
import { gql } from "@apollo/client";
const query = gql`
  query Launches {
    launches {
      id
      details
      mission_name
      rocket {
        rocket_name
        rocket_type
      }
    }
  }
`;

export default async function Home() {
  if (typeof window !== "undefined") {
    console.log("Client side HomePage");
  } else {
    console.log("Server side HomePage");
  }

  const { data } = await getClient().query({ query });

  return (
    <main>
      {data.launches.map((element) => (
        <tr>
          <td>{element.id}</td>
          <td>{element.mission_name}</td>
        </tr>
      ))}
    </main>
  );
}
```

Question 10

1. Explain what a native web component is.

Web components are a set of browser APIs that allow developers to create custom, reusable HTML elements. These elements encapsulate their own styles and behaviors, making them safe to use anywhere in your code without causing conflicts or unintended side effects. They can be used with or without a JavaScript framework, providing flexibility for developers to choose the right tool for their project.

The creation of a web component involves defining a class that extends from `HTMLElement`, specifying the functionality of the component. Then, the class is registered using the `customElements.define()` method, associating the custom element name with the class.

Web components are **interoperable**, meaning they can be used across different frameworks. For instance, a web component created in Angular can be used in a Vue application, or vice versa. This interoperability breaks down the artificial boundaries between developers and their chosen frameworks.

Despite their advantages, web components have some **limitations**. One significant challenge is the lack of straightforward support for server-side rendering (SSR). While there are workarounds and tools that address this issue, it remains an area where improvements can be made.

2. Which functionality is required by the browser?

To support native web components, a browser must implement several key technologies:

Custom Elements: Allows authors to define new HTML/DOM elements.
Shadow DOM: Encapsulates the internal DOM and styling of the components.
HTML Templates (<template> and <slot>): Used to declare fragments of markup that go unused at page load, but can be instantiated later at runtime.

3. Show how to build a native web component.

Creating a native web component involves several steps:

1. Define the Class: First, you need to define a class that extends from HTMLElement. This class will represent your custom element.

```
class HelloWorld extends HTMLElement {
 // connect component
 connectedCallback() {
   this.textContent = 'Hello World!';
 }
}
```

In this code, the connectedCallback method is called when the element is added to the DOM. Here, it sets the text content of the element to “Hello World!”

2. Register the Custom Element: After defining the class, you need to register it as a custom element using the customElements.define() method. The first argument to this method is the name of the custom element (which must contain a hyphen), and the second argument is the class itself.

```
customElements.define('hello-world', HelloWorld);
```

This code registers the HelloWorld class as a custom element with the name hello-world 2.

3. Use the Custom Element: Now you can use your custom element in your HTML just like any other HTML element.

```
<hello-world></hello-world>
```

When this element is rendered in the browser, it will display "Hello World!" 2.

4. How can Lit help with building web components?

Lit is a simple library for building fast, lightweight web components. It simplifies the process by providing:
Reactive properties and attributes: Automatically updates the component when properties change.
Efficient rendering: Minimal overhead and fast updates using lit-html for efficient rendering.
Declarative templates: Easy to read and write HTML templates in JavaScript.

5. Discuss pros and cons of native web components.

Pros:

Encapsulation: Keeps the markup structure, style, and behavior hidden and separate from other code.
Reusability: Components can be reused across different web applications.
Standards-Based: They are part of the web standard, so no additional libraries are needed.
Interoperability: Can be used with any JavaScript framework or no framework at all.

Cons:

Browser Support: While improving, not all browsers fully support all features of web components.
Learning Curve: Understanding Shadow DOM and custom elements can be challenging.
Integration: Can be tricky to integrate with some older frameworks or tools.
Tooling: Certain development tools might not yet fully support web components.