International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Integrating REST APIs in Single Page Applications using Angular and TypeScript

[1]Sneha Aravind, [2]Ranjit Kumar Gupta, [3]Sagar Shukla, [4]Anaswara Thekkan Rajan

Submitted: 25/04/2021 **Revised**: 02/06/2021 **Accepted**: 15/06/2021

Abstract: This research paper examines the integration of REST APIs in Single Page Applications (SPAs) using Angular and TypeScript. It explores the fundamental concepts of SPAs, the Angular framework, TypeScript, and RESTful architecture. The study delves into best practices for API integration, state management, performance optimization, and security considerations. By analyzing current methodologies and emerging trends, this paper aims to provide a comprehensive guide for developers and researchers working on modern web applications. The research covers various aspects of SPA development, including framework architecture, API integration techniques, state management strategies, and deployment methodologies, offering insights into the complexities and best practices of building robust, scalable web applications.

Keywords: *Single Page Applications, Angular, TypeScript, REST APIs, Web Development, Frontend Frameworks, State Management, Performance Optimization, API Integration, Security*

1. Introduction

1.1 Background

The landscape of web development has evolved significantly over the past decade, with Single Page Applications (SPAs) emerging as a popular architectural pattern for building responsive and dynamic web interfaces. Using the SPAs, page reloads are reduced aiming to deliver a more convincing application-like interface to Web applications. This shift has been made possible by the rising consumers' expectant for richer, more engaging content in web page s that are as engaging as alternative stand-alone desktop and, or mobile application.

As for today, Angular– comprehensive front-end development platform, created by Google is another most used technology of choice to build SPAs. The framework was first introduced in 2010 as AngularJS but was redesigned as the Angular 2 in 2016. The subsequent releases have maintained the development, enriching the subsequent Angular with new robust features; the latest is the Angular 12, at the time of writing in May 2021. Here Angular uses TypeScript, which is a statically typed script of ECMAScript that supersedes JavaScript and helps in improving the efficiency of the developers and quality of the code being developed.

Web based Application: REST (Representational State Transfer) APIs are the standard that are used for client-server interaction in present era. First proposed by Roy Fielding in 2000, for his doctoral thesis, REST architectural principles have garnered much acceptance because of their simplicity, performance, and compliance with HTTP. Incorporation of REST APIs in SPA using Angular is a very important facet when it comes to building applications that are data-driven and capable of making seamless interconnectivity with back-end services.

1.2 Objectives

The primary objectives of this research are to provide a comprehensive understanding of Single Page Applications and their benefits in the context of modern web development. This includes exploring the Angular framework and TypeScript, analyzing their roles in SPA development, and examining how they contribute to building robust and maintainable applications.

Also, the research seeks to discuss the proper implementation of REST APIs with regards to angular applications. This ranges from checking into different methods of performing HTTP requests, handling the result, and working with non-blocking tasks. The study also aims at looking at the best state management practices and optimization of performance that is vital in creating more robust SPAs.

Another work focus is to discuss the security issues and the testing approaches on the API integration. This comprises considerations of the arena of authentication, cross-origin resource shares (CORS), as well as methodologies of avoiding standard web weaknesses.

Finally, the research will focus on the current developments and trends of SPA deployment and future

[1]*Independent Researcher,USA.*
[2]*Independent Researcher, USA.*
[3]*Independent Researcher, USA.*
[4]*Independent Researcher,USA.*

trends on the SPA development with an aim of giving the readers glimpse of the future technologies and methodologies in development of web applications.

1.3 Scope of the Research

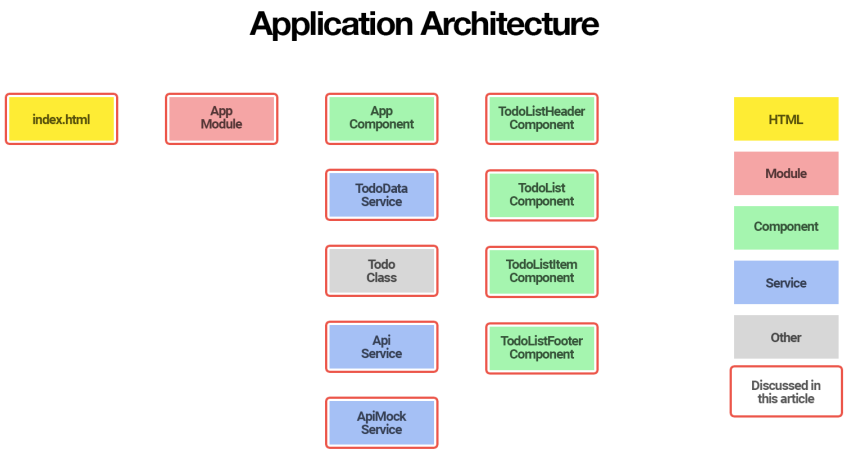
This research focuses on Angular version 12 (released in May 2021) and TypeScript 4.3 (released in May 2021), as these were the latest stable versions at the time of writing. The paper covers various aspects of SPA development, including framework architecture, API integration, state management, and deployment. While the primary focus is on REST APIs, alternative approaches like GraphQL are briefly discussed to provide a holistic view of the subject matter.

The study examines the entire lifecycle of SPA development, from initial setup using Angular CLI to final deployment and continuous integration strategies. It also explores advanced topics such as lazy loading, server-side rendering, and micro frontends, providing a comprehensive overview of modern SPA development practices.

2. Understanding Single Page Applications (SPAs)

2.1 Definition and Characteristics

Single Page Applications are those web applications where a single HTML page is loaded and it acts as a start page and all the pages and updates of the application are done through AJAX requests. In contrast to the conventional multipage applications, SPAs incorporate AJAX as well as HTML5 to construct sight applications that do not require frequent page reloads. This in turn makes for a more integrated experience for the user akin to richness and experience of application built entirely for the web or Desktop/Mobile rather than web/HTML Application.



The main features of SPAs are the constant updates of parts of the page instead of the whole, client-side routing, handling asynchronous data, as well as increased apparent efficiency. Generally, in SPAs, the HTML, CSS and JS that is required for the application to function are loaded in the first page load, or are loaded on the fly as the application is used. The initial load may be higher than that of a typical web page, but subsequent activates normally take less time since only data is transferred, not even layouts.

2.2 Benefits and Challenges

As much as SPAs, there is a long list of advantages which are in many ways not minor. It provides improved usability with faster response since content updates are done in real time as opposed to the whole page refresh. This causes the application to be more responsive and keep users glued to their screen. Partial content update, which is significant in implementing SPAs, acts as an advantage since the server is not burdened with sending HTML pages for every request but only the data.

From the developmental angle then, SPAs are an excellent approach to constructing complex applications with functional layers because they partition the issues into the frontend and the backend. This division makes the working process clear, enables better development of the work process and can be maintained easily. Also, to debug the SPAs might be easier using browser developer's tools, unlike in node. js, the complete application state can be found in the browser.

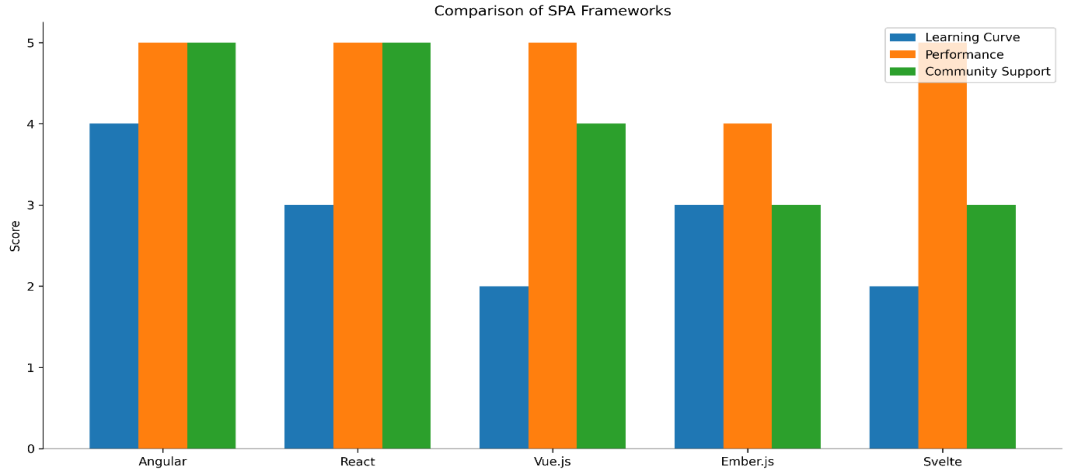
But also, SPAs have some peculiarities which can be regarded as disadvantages: The first time it loads there will be potentially larger JS bundles, which will affect subjective perception of performance on slow networks. As can be seen, SEO can be more challenging in SPAs since search engine crawlers struggle with indexing such content as those dynamically created. This problem has been somewhat resolved using such solutions as server-side rendering though it is still being taken into consideration.

SPA's approach to browser history comes as an added aspect to consider in which the application is responsible for routing and creating the feeling of application-like navigation without reloading the page. Lastly, the state management in SPAs could be even more complicated, more so for large application, which often requires the use of the state management libraries or the need to design the architecture of application development with this challenge in mind.

2.3 SPA Frameworks: An Overview

A few frameworks help in creating SPAs, and each of them has its conceptual approach and the number of characteristics. Angular is developed by Google and it is big framework powerful but with rather steep learning curve. It offers a comprehensive solution for developing applications of considerable size, it offers sophisticated Dependency Injection, has enhanced templates and is natively based on reactive programming.

React, by Facebook is an interface technology that has received a lot of attention because of its ease to implement and integrate into various entities. While React is not as developed, complete framework as Angular, it has more related libraries that can offer similar services.



Vue. js is another well-liked one, considered to be easy to learn and rather versatile. it ensures implementation does not happen in a haphazard way but can be adopted gradually; this means that is useful for a broad range of application, from small to large.

The other typical/other frameworks are; Ember. Previously, it used the js, which is known for its convention over the configuration approach, and Svelte which took a compile-time approach to the reactivity.

The choice of framework often depends on project requirements, team expertise, and personal preference. Table 1 provides a comparison of key features among these popular SPA frameworks.

Table 1: Comparison of Popular SPA Frameworks

Feature	Angular	React	Vue.js	Ember.js	Svelte
Learning Curve	Steep	Moderate	Gentle	Moderate	Gentle
Performance	Good	Excellent	Excellent	Good	Excellent
Community Support	Large	Very Large	Large	Moderate	Growing
TypeScript Support	Native	Via JSX	Via Class Components	Via Decorators	Via Preprocessor
State Management	NgRx	Redux	Vuex	Ember Data	Built-in
CLI Tool	Yes	Create React App	Vue CLI	Ember CLI	No official CLI

International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

3. Angular Framework

3.1 Core Concepts and Architecture

Angular is a platform with everything you need to create applications in plain HTML and only TypeScript. It provides core and optional features as a set of TypeScript libraries that you incorporate in your applications. Upon to build an angular application, there are basic fundamentals that support the architecture of such an application.

The most important component from the Angular world is the NgModules which helps in providing the compilation context for Groups. An NgModules gathers related code into functional groups; an Angular application is best described as a set of NgModules. An app always has at least a root module that allows bootstrapping and often has many, many more expressing features.

Components declare views, which are screens which contain on-screen elements of which Angular can select one and or change based on your program logic and data. Subcomponents in components use services which are not directly concerned with views. Services can be injected into components as dependency and makes your code much more modular, reusable and efficient.

3.2 Components and Modules

Modules are usually the main block of construction for Angular applications. Each component consists of:

- An HTML template that proclaims what shows up on the page
- An abstract class in TypeScript that expresses behavior
- An identifier that defines the usage of the component on a template.
- Optionally, some CSS styles that have to be used in the template

Here's an example of a simple Angular component:

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

@Component({
  selector: 'app-hello-world',
  template: '<h1>{{title}}</h1>',
  styles: ['h1 { font-weight: normal; }']
})
export class HelloWorldComponent {
  title = 'Hello, World!';
}
```

Modules in Angular help to organize an application and extend its capabilities. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule.

3.3 Dependency Injection

Dependency Injection (DI) is a design pattern and a method of implementing it, for introducing parts of an application into other parts of the application that need them. In Angular, DI is done via the constructor of a class and Angular has the ability to automatically instantiate and inject the dependencies.

In general, we can say that DI framework of Angular gives dependencies to a class at the time when the class is created. This makes the code more 'pluggable' and easier to test since dependencies can be easily replaced by forms of 'fakes' during testing.

3.4 Angular CLI

The Angular CLI is the standalone tool that is greatly helpful in the development of the Angular applications. The basic format is a set of commands that are used to initiate a project, which comes with predefined code, to execute tests, and to build the application.

Some key features of the Angular CLI include:

- Project scaffolding: Allows for the creation of a new Angular project in a standard format and setting up the needed architecture.
- Code generation: Create classes, services, pipes and other components and related things in Angular with right naming conventions and code templates.
- Development server: To encourage development, it is recommended to launch a local development server with live reload.
- Build optimization: The last step is to prepare the application for production and inclusion of optimisation methods.
- Testing utilities: Provide unit and end-to-end tests configured with pre-selected test runners.

4. TypeScript in Angular Development

4.1 TypeScript Features and Advantages

TypeScript is a typed scripting language developed and maintained by Microsoft; it is JavaScript based, albeit being a superset of it and compiling to plain JS. It introduces static typing, classes as well as modules to JavaScript; thus, helping developers write better code.

Key features of TypeScript include:

1. Static typing: In particular, TypeScript permits extending variable declarations, function parameters, and function return values with types. This makes a better understandings of tooling support such as autocomplete, refactorings, and detection of errors at compile time possible.
2. Object-oriented programming: One of the consequences of implementing Object-Oriented concepts in TypeScript is its support of classes, interfaces and modules.
3. Enhanced IDE support: Static typing feature of TypeScript allows IDEs to suggest better code completion, refactoring, and navigation than what dynamic typing allows.
4. ECMAScript compatibility: TypeScript was designed as JavaScript successor, it has all of the ECMAScript features up to the most recent ones that can be used to write applications for older browsers.
5. Gradual adoption: The TypeScript is designed that it can be gradually integrated into the JS project, which may help the team to gradually change to the typed JS.

The use of TypeScript in Angular has a number of benefits and the following are brief descriptions of these benefits. It offers improved tooling support, helps in matters concerning errors within the planning and designing phase and is advantageous in the matters concerning maintainability. Angular is itself written in TypeScript and the features of TypeScript are well integrated in the framework.

4.2 Type Safety and Object-Oriented Programming

Safety of the data type is one of the essential advantages that work on angular using TypeScript. It means that using type annotations, developers are able to fix all the potential type-related issues at compile-time. This results in very strong code and it has the effect of minimize the bug that may occur in the production.

```typescript
interface User {
  id: number;
  name: string;
  email: string;
}

function greetUser(user: User) {
  console.log(`Hello, ${user.name}!`);
}

// This will compile successfully
const validUser: User = { id: 1, name: "John Doe", email: "john@example.com" };
greetUser(validUser);

// This will cause a compile-time error
const invalidUser = { id: 2, name: "Jane Doe" };
greetUser(invalidUser); // Error: Property 'email' is missing in type '{ id: number; name
```

In this example, TypeScript catches the error of passing an object that doesn't match the User interface to the greetUser function, preventing a potential runtime error.

Object-Oriented Programming (OOP) in TypeScript allows developers to create reusable, modular code using classes and interfaces. This aligns well with Angular's component-based architecture. Here's an example of a simple class in TypeScript:

```
class Person {
  private name: string;

  constructor(name: string) {
    this.name = name;
  }

  public greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}

const person = new Person("Alice");
person.greet(); // Output: Hello, my name is Alice
```

4.3 TypeScript Decorators in Angular

Decorators are a TypeScript feature heavily used in Angular for metadata annotation. They allow you to modify or enhance classes and properties at design time. Angular uses decorators to define components, services, and other core concepts.

Here are some common decorators used in Angular:

1. @Component: Defines a class as an Angular component and provides metadata about the component.
2. @Injectable: Marks a class as available to be provided and injected as a dependency.
3. @Input and @Output: Used for component communication, marking properties as inputs or outputs.
4. @NgModule: Defines a module that contains components, directives, pipes, and providers.

5. REST APIs: Principles and Best Practices

5.1 RESTful Architecture

Representational State Transfer or REST is an architecture scheme for the development of the network application which has served as the backbone of contemporary Web services. Proposed by Roy Fielding in his PhD thesis in the year 2000, REST uses stateless, client-server, cacheable communication protocol that is predominantly HTTP. RESTful systems are used to enhance scalability, to decrease coupling, and to increase independence of components, these factors makes this approach highly suitable for development of distributed systems.

Client-server, stateless, cache, uniform interface, layered system, and code on request, these are the six principles that are the foundation of the REST architectures . The client-server principle can also facilitate that the focus lies on two different architectural aspects, such that the user interface and database can be developed on their own without necessarily forcing down tightly linking the two into one design. Statelessness entails that any message from the client to the server has to include all the information that the server must understand about the message and therefore the scalability and ease in designing the servers. Cacheability means that responses must provide information about their cacheability, which is itself cacheable, thus enhancing performance and request handling capacity.

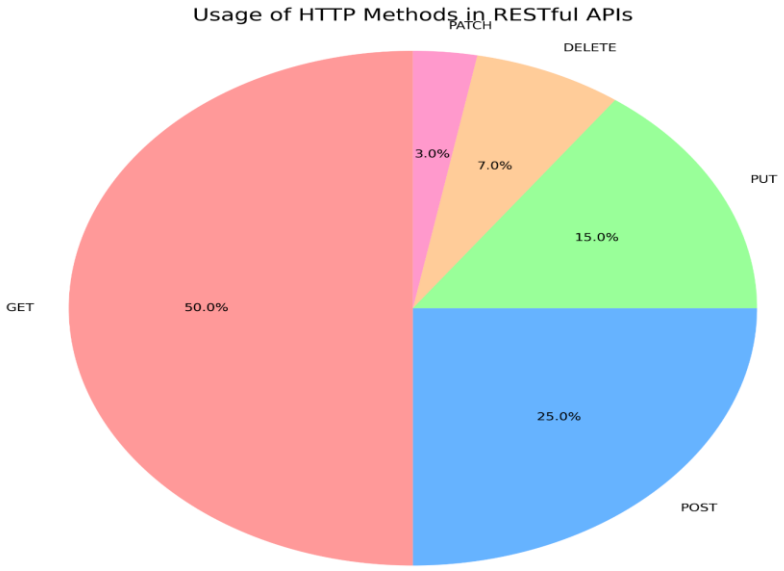
Among the principles of REST probably the most outstanding one is the uniform interface principle. This seems to lay down a standard manner of communicating with a given server whether through a PC, a mobile phone, etc, or when using an application. This is usually attained by means of standardizing HTTP methods and the notion of resources characterized by URIs. It remains to note that in contrast to client–server model, the layered system principle provides for the use of intermediary servers, which makes it possible to scale up the system to the necessary level and carry out load-sharing. Lastly, the code on demand option enables the servers temporarily to enhance the client's functions through transferring of code.

5.2 HTTP Methods and Status Codes

RESTful APIs always use HTTP methods that are used to interact with the specified resource. The four commonly used HTTP methods are; GET to request resources, POST for creating a resource, PUT for modifying a resource and DELETE to remove a resource and PATCH is used to partially modify a resource. These methods are in order with the CRUD functions – Create, Read, Update, Delete – normally found in data handling systems.

HTTP status codes are also very vital in RESTful communication as they indicate whether API request was successful or not. These codes are grouped into five classes: ,Second: Informational responses (100–199), Successful responses (200–299), Redirection messages (300–399), Client error responses (400–499), and Server error responses (500–599). Some of the well known status codes are 200, which is 'ok', 201 which means 'created',

400 which is 'bad request', 401 which means 'unauthorized', 404 which is 'not found' and the last one is 500 which means 'internal server error'.



These status codes are slight different and understanding and using these properly is very important when designing APIs. It speaks to clients about the results of their actions and allows to correctly respond to an error, thereby increasing the stability of the system.

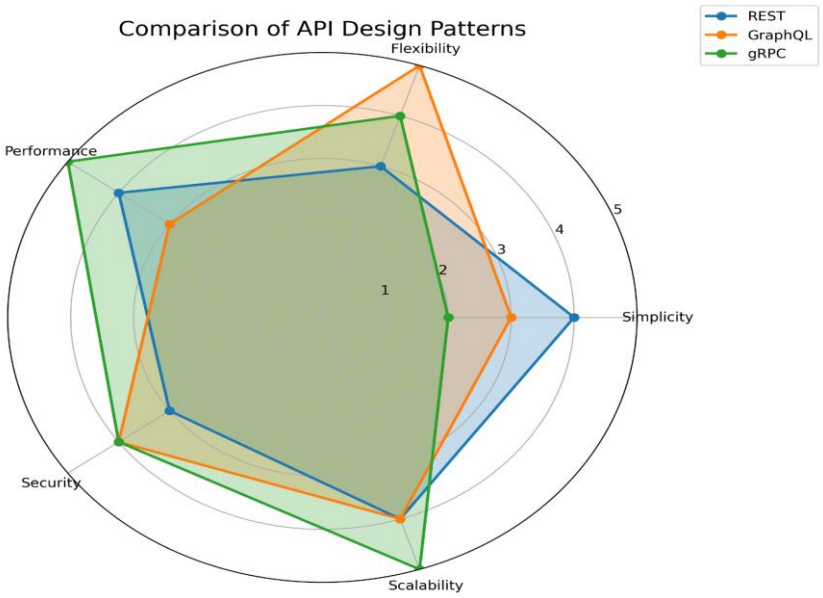
5.3 API Design Patterns

While implementing RESTful APIs some common patterns and conventions have been as follows in order to ensure that the APIs are consistent, manageable and consumable. There is one fundamental and effective rule, and it is the rule of the nouns used for naming the resources. It is preferable to refer to the resources using nouns rather Than verb since the REST is entity oriented. For instance, /users is better than /getUsers This approach is in harmony with the concept of resources as the only abstraction of information used in application in REST.

Another important pattern is the use of the plural nouns where the set is considered. If we are talking about a set of resources, plural nouns should be used, in order to avoid confusion and inconsistencies. For instance, /users is preferable to /user if we speak of a set of user resources. This convention useful in designing of APIs that are easy to understand and describe on their own.

Sub-resources present a good way of showing relationships of resources with resources. For instance, to work with the posts of a certain user, the API can use such a path as /users/{userId}/posts. This structure is quite evident in the API to denote the users and their post further enabling the users to navigate through the API easily while improving its semantic connectivity.

Versioning is another key element of design API. Given the changes in APIs, introducing breaking changes is disadvantageous in so far as client applications are concerned. To be able to address this, APIs involve versioning in their integration. Some of these are URL versioning for example /v1/users, header versioning, or use of content negotiation. These two methods have their advantages and disadvantages and the choice depends on the project that is to be handled.



Pagination is essential for handling large datasets efficiently. Instead of returning all results at once, which can be resource-intensive and slow, APIs should support pagination. This can be implemented using query parameters like limit and offset or page and per_page. For example, /users?page=2&per_page=20 might return the second page of results with 20 users per page.

Filtering, sorting, and searching capabilities enhance the flexibility of APIs. These features allow clients to request specific subsets of data, improving efficiency and reducing unnecessary data transfer. Filters can be implemented using query parameters (e.g., /users?status=active), while sorting might use parameters like /users?sort=name:asc.

5.4 Authentication and Security

As important as the implementation of APIs, security is of key concern where the API is dealing with restricted operations or data. The concept of users' access control involves the use of techniques such as authentication and authorization to control access to protected resources. Typically, the basic form of authentication among the RESTful APIs includes use of API keys, OAuth 2. 0, minimum usage of cookies and also JSON Web Tokens (JWT).

API keys are very easy to implement but they are not secure and can be sued for APIs that are not very sensitive or those that are public. OAuth 2. 0 is a more robust protocol that controls authorization in a standard method from Web, Mobile and Stand alone Desktop applications securely. Often employed for access delegation, it offers reasonable levels of protection while being rather easy to manage.

JSON Web Tokens are widely used because they are stateless and it is possible to envelop claims safely in them. JWTs are about both credit and content making it feasible for use in a variety of API security circumstances.

Besides authentication, APIs must incorporate the right form of authorization to allow a user to only request and use specific resources and actions that have been authorized to him or her. Role based access control (RBAC) is an example of a strategy and this identifies users by roles that are allowed certain privileges.

Transport layer security is used for protecting the data in transit. HTTPS should be used to avoid code eavesdropping and man in the middle attacks in all production APIs. Also, rate limiting can provide the APIs necessary safeguard against malicious usage and fairly distribute the requests among the clients.

Another measure of security is Cross-Origin Resource Sharing (CORS), the feature helpful when consuming APIs by web apps. CORS defines how a server may specify any origins (domains) aside from the server's own that a browser should allow, to load a resource.

When followed these principles and recommendation developers great RESTful APIs that are Secured, Multi-Tenant and Usable. These APIs are the foundational structure to most current-era Web and Mobile applications, facilitating client-server communication in a regular format.

6. Integrating REST APIs in Angular Applications

6.1 Angular HttpClient Module

Angular offers the HttpClient module as a fast and diverse framework for communicating with backend servers with HTTP protocols. This module is provided by the @angular/common/http package and provides a Readily Available client HTTP API for angular applications. It contains new elements such as typed request and response

models, interceptors, and a simple error-handling system the tools, it is convenient to work with.

To use HttpClient, you first need to import the HttpClientModule in your Angular application's root module or any feature module where you plan to make HTTP requests. Here's an example of how to import it in the AppModule:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';

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

Once imported, you can inject the HttpClient service into your components or services and use it to make HTTP requests. Here's a simple example of how to make a GET request:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-user-list',
  template: '<ul><li *ngFor="let user of users">{{user.name}}</li></ul>'
})
export class UserListComponent implements OnInit {
  users: any[];

  constructor(private http: HttpClient) {}

  ngOnInit() {
    this.http.get<any[]>('https://api.example.com/users')
      .subscribe(
        users => this.users = users,
        error => console.error('Error fetching users', error)
      );
  }
}
```

In this example, the component makes a GET request to fetch a list of users when it initializes. The HttpClient.get() method returns an Observable, which we subscribe to in order to handle the response or any errors.

6.2 Observables and RxJS

HttpClient methods of Angular return observables, and that takes a help of RxJS (Reactive Extensions for JavaScript). Observables are great at working with forms of data that come in a stream and are meant to be

processed at any time, which is why they're perfect for HTTP requests.

They are many operators provided by RxJS that can be used to filter transform, combine or manipulate these Observables. Some commonly used operators in the context of HTTP requests include:Some commonly used operators in the context of HTTP requests include:

- map: Modify the items released by an Observable.
- catchError: How to properly process error in the Observable stream?
- switchMap: Switch to an Observable, finish the prior inner Observable, and transmit values from the latter.
- retry: Specifying how often a failure should occur in an Observable sequence Retry an Observable sequence Click the image to enlarge Retry an Observable sequence Retry an Observable sequence should an error occur Click the image to enlarge

7. State Management in SPAs

7.1 Client-Side Data Storage

"Between all the patterns mentioned, data management is the most significant for Single Page Applications since the sensitivity of the application's reaction directly depends on it. " This process is facilitated by the client-side data storage which provides several means to store data in the local browser environment. These storage options can be as basic as the key-value pairs and can go up to the structure data storage options.

Perhaps the most popular of the storage formats is the Web Storage API, which includes two types of storage, one called 'localStorage' and the other 'sessionStorage'. localStorage on the other hand is the data that can be accessed even if the window is closed, other than that it is closed by default; on the other hand, sessionStorage stores the data only for the duration of the browser session. Caching is easy and these APIs designed for storing a small amount of data, for example, user preferences or an authentication token. For instance, you might store a user's authentication token in localStorage like this: instead, use localStorage. setItem('authToken', 'user123token'). However Web Storage is synchronous and it supports only string data which could be crucial point for using more complex data structures as well as for large data sets.

For a more complicated storage, IndexedDB is the answer to the call. IndexedDB is actually a low level API for client storage of large amounts of structured data; files and blobs included. It has indexes through which it is in a position to search for this data at high speed. Although the syntax and operation of IndexedDB are more complex than of the Web Storage, the API provides more freedom and works faster with the large amounts of data. Libraries like Dexie. Another two libraries, js or LocalForage, can help making work with IndexedDB more convenient and less problematic due to changing API at any time.

When one is handling data especially with a lot of information it is wise to factor in security measures. It is essential for developers to know that client-side storage is comparatively less secure to server-side storage because it is a vulnerability to XSS attack. Thus, data that is to be stored should be encrypted and server side validation such be performed on data that is critical.

7.2 NgRx for State Management

When it comes to the larger angular application where state management becomes a problem for the developers NgRx is the perfect solution. NgRx or Geki is a state management solution for Angular that was heavily influenced by Redux. It has the advantage of being very explicit and straightforward in identifying the process of managing the state of an application, which is very useful in generating an organized framework in tackling the state transitions of an application as well as the complexity of the state transitions.

The core concepts of NgRx include:

1. Store: A single data structure that cannot be changed and contains all the data of an application.
2. Actions: Simple forms stating what occurred in the application.
3. Reducers: Functions which do not have side effects and which receive the previous state and an action as input, and return the new state.
4. Effects: Side effect model for managing difficult callback based asynchronous operations.
5. Selectors: Functions to get some slices of the store state used to be pure.

Here's a basic example of how these concepts come together in an NgRx application:

```
// Action
import { createAction, props } from '@ngrx/store';

export const addTodo = createAction(
  '[Todo] Add Todo',
  props<{ text: string }>()
);

// Reducer
import { createReducer, on } from '@ngrx/store';
import { addTodo } from './todo.actions';

export interface TodoState {
  todos: string[];
}

export const initialState: TodoState = {
  todos: []
};

export const todoReducer = createReducer(
  initialState,
  on(addTodo, (state, { text }) => ({
    ...state,
    todos: [...state.todos, text]
  }))
);

// Effect
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { map, mergeMap } from 'rxjs/operators';
import { TodoService } from './todo.service';
```

```
import { addTodo } from './todo.actions';

@Injectable()
export class TodoEffects {
  addTodo$ = createEffect(() =>
    this.actions$.pipe(
      ofType(addTodo),
      mergeMap(action => this.todoService.addTodo(action.text)
        .pipe(
          map(todo => ({ type: '[Todo API] Todo Added Success', payload: todo }))
        )
      )
    )
  );

  constructor(
    private actions$: Actions,
    private todoService: TodoService
  ) {}
}
```

In this example, we define an action to add a todo, a reducer to handle this action and update the state, and an effect to perform the side effect of saving the todo to a backend service. This structure ensures that all state changes are predictable and traceable, which is particularly valuable in large, complex applications.

7.3 Caching Strategies

Use of caching is fundamental in the enhancement of the performance of Single Page Applications, and hence holds a critical position in the development process. Caching can help to decrease number of requests that are sent over the network, decrease the server load, and increase the amount of time which it takes to access frequently accessed data. There are various forms of caching used in Angular applications though they can be adopted consecutively or concurrently in enhancing the methods of accessing and managing data.

A typical on is in-memory caching, where the data is stored in the memory of the application so that it can be accessed soon. This may be done using RxJS's shareReplay operator which, enables multiple subscribers to share an observable execution and store the most recently emitted value. Here's an example:

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

@Injectable({
  providedIn: 'root'
})
export class UserService {
  private users$: Observable<User[]>;

  constructor(private http: HttpClient) {}

  getUsers(): Observable<User[]> {
    if (!this.users$) {
      this.users$ = this.http.get<User[]>('/api/users').pipe(
        shareReplay(1)
      );
    }
    return this.users$;
  }
}
```

In this case, the first time when getUsers() function is called, the HTTP request will be made and on any subsequent calls to this function, the cached data will be returned. This strategy comes in handy when dealing with information that does not often fluctuates within a users session.

8. Performance Optimization

8.1 Lazy Loading Modules

Among the most efficient approaches of how to optimize the performance of big angular applications, there is the approach called lazy loading. This approach entails the fact that the specific set of features for the application is separated and other features are loaded as the need arises. Lazy loading means the website will load only a part of the content at a time, thus making the website load lighter at start-up.

To enable lazy loading in Angular, you create routes that correspond to some feature modules. These modules are then loaded asynchronously as and when the user accesses the corresponding path/route. Here is how lazy loading should be set for the app-routing of the application: module. ts file:

In this example both the CustomersModule and OrdersModule are marked as lazy-loaded. These will only be downloaded and initialized when the user gets to any of the defined routes which are '/customers' or '/orders'. This can really help to minimize the time for which the burden is placed on the initial load of the application and is especially useful for projects with a large number of features divided into various modules.

Lazy loading works great to make the first impressions faster and, at the same time, may add a little bit of time when switching to the other parts of the application for the first time. These trade-offs should be carried out based on the requirements that the developer has for his application in terms of performance and the experience that he wants to provide the user with.

8.2 Change Detection Strategies

Angular change detection mechanism also has the rather important function of maintaining the view in sync with the application state. However, by default Angular is employing a zone-based change detection where Angular checks for changes in all components on every event that takes place. This workflow, as you know, makes the view always up to date, but it can cause great performance problems in large-scale complex applications that need to render trees of components.

For this purpose, Angular offers the so-called OnPush change detection strategy which helps to maximize the effectiveness of change detection. This strategy instructs Angular as to when it should look for changes in a component; the changes may be in the properties into the component or when an event is fired either by the component or one of its descendants. Here's how you can implement the OnPush strategy:Here's how you can implement the OnPush strategy:

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

@Component({
  selector: 'app-user-profile',
  template: `
    <h2>{{ user.name }}</h2>
    <p>{{ user.email }}</p>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserProfileComponent {
  @Input() user: User;
}
```

By making changeDetection set to ChangeDetectionStrategy. OnPush we are making Angular to check this component for changes only if the input properties have been modified. This can considerably decrease the iterate number of change detection, particularly within the big applications, which have numerous components.

What is crucial for understanding is that if you use OnPush, all inputs should be marked as being immutable. If you change an object which is being passed as an input, then the angular will not be able to identify the change. Rather, what you should do is to create a new object from the class and assign the values to it. This practice correlates with the ideology of immutability related to the state management which may result in more predictable applications and situations when it will be easier to track the error's source.

8.3 Server-Side Rendering (Angular Universal)

Server-Side Rendering is one where the initial rendering of the HTML content occurs on the server not on the client's browser. Angular Universal is the best way to perform SSR on Angular based applications. It can dramatically enhance the apparent download time of your application on low-end devices or low connection and is a requirement for SEO since it lets search engines index your dynamically-fed data.

In order to use SSR with Angular Universal, you need to create a server side application module and use and apply this to pre-render your application on the server side. Here's a basic example of how to set up the server module:

```
import { NgModule } from '@angular/core';
import { ServerModule } from '@angular/platform-server';
import { AppModule } from './app.module';
import { AppComponent } from './app.component';

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

Realization of SSR can be challenging, particularly for the applications that depend on the use of browser-specific interfaces. Among other things, you may need to confirm that this application can run in a server environment, which would almost certainly entail certain refinements as to how you handle browser-specific features.

There are however some disadvantages of using SSR which are the follows Despite such disadvantages, SSR can enhance initial load time and SEO significantly. It makes your application more complicated and it may lead to some extensive use of server belongings. Thus, the choice of the SSR implementation should be determined by your needs because, in applications where the SEO is critical, it is a decisive factor.

By adopting these performance optimization strategies; the above discussed lazy loading modules, optimizing change detection, and server-side rendering you are in a position to enhance the performance of your Angular applications. These optimizations also improve not only the positions in search engine rankings but also the usability of the site for the users regardless the device and the conditions of the network connection. Of course, with any optimisations trade-offs some of these changes may not be as beneficial in your particular application environments as expected.

9. Testing REST API Integration

9.1 Unit Testing with Jasmine and Karma

Testing is one of the critical conditions in ensuring that Angular applications is healthy in terms of scalability for maintenance and usage of REST APIs. Angular already has out of the box unit testing capabilities via the Jasmine testing framework and via using the Karma test runner. These tools enable the developers to write and execute the tests that check unit by unit of the components, services, and all other sub parts of the application independently.

9.2 Integration Testing

Although it is possible to test many units and services in isolation, there has to be an integration test to confirm that they are working coherently. They are typically performed in the context of REST API integration during the tests of interaction between components, services, and the API.

9.3 Mocking HTTP Requests

Mocking HTTP requests is particularly important when carrying out test termed as REST API integration. It gives the ability to simulate the application activity without using a real API that it can be unavailable or updated in time. HttpClientTestingModule from Angular API offers powerful tools for the mock HTTP calls.

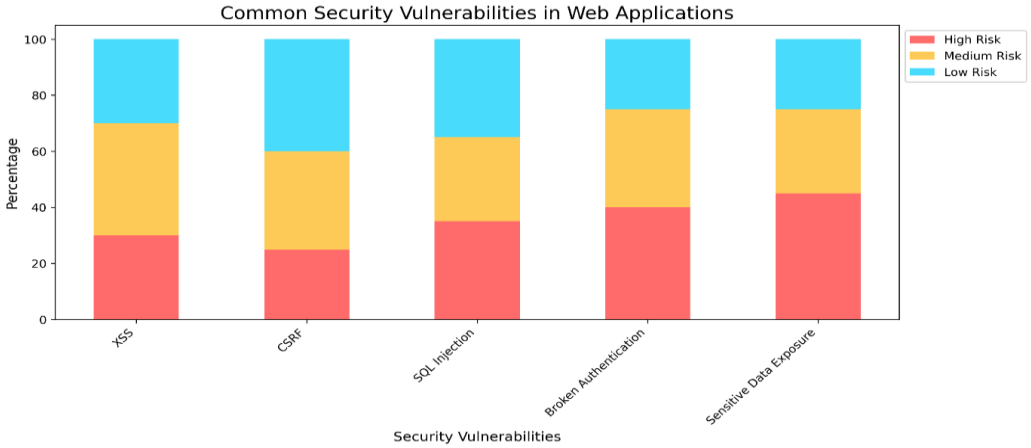
10. Security Considerations

10.1 Cross-Origin Resource Sharing (CORS)

Cross Origin Resource Sharing (C. O. R. S) is one of the most significant security measures that dictate the access rights of other domains (such as APIs) on a web server than that of the web application. When it comes to Single Page Applications where the main consumer of a REST API is developed, CORS has a very important role of banning the unauthorized access to the resources. There exist controls on allowing such cross-origin requests put in place by the browser through the Same-Origin Policy when a web application make a request in a different domain. CORS also allows the server to give out the list of allowed origins for the resource by the server.

CORS has to be set up properly in terms of security measures as well as in terms of its usage. On the server side, the correct headers related to origins have to be set in order to define which resources may be requested by them. Some of these headers are; Access-Control-Allow-Origin, Access-Control-Allow-Methods, and Access-Control-Allow-Headers. For such settings, it is necessary to set headers in a proper way with regard to security requirements in certain applications. If CORS policies are implemented in a liberal manner, then websites may be

opened up to malicious attacks, on the other hand, if CORS is implemented in a very conservative manner, then genuine access to resources may be blocked.



At the client side, the Angular developers have to pay attention at CORS consequences while performing the HTTP requests between the different domains. Angular HttpClient deals with the CORS preflight requests on its own, however, it is vital that API endpoints are configured to handle these requests. In some cases, the developer may have to include some other headers in the request and this may lead to the server sending preflight requests, thus need for CORS handling.

10.2 JSON Web Tokens (JWT)

JSON Web Tokens are now widely used to deal with the user identification and access control in many contemporary web apps, such as Angular-based Single Page Applications (SPAs) communicating with RESTful APIs. JWTs are small and rather independent tokens that can safely transfer information from one party to another in the form of JSON. They are specifically helpful in Single Page Applications because they allow for stateless authentication which is in tandem with RESTful architecture.

Whenever jwt is used in an angular project, the token is generally gotten after successful login, and it is usually stored on the client side, it can be stored in browser storage such as localStorage or sessionStorage. This token is used in the next API calls in the Authorization header for the user authentication purposes. What crucial is indicated is while using JWTs as medium of transferring information is secure, it has to be handled carefully on the client-side to avoid cases of security breaches.

There exist certain risks such as cross site scripting (xss) attack which may threaten the stored JWTs, developers should keep an eye on. To avoid these risks it is advisable to implement JWT s in HttpOnly cookies if possible since the cookies would not be accessible by an XSS script. Secondly, the time space when tokens can be used also plays an important role – usually there are token refresh mechanisms, which also can help to improve security.

10.3 Cross-Site Scripting (XSS) Prevention

Cross Site Scripting or commonly called XSS is still a popular attack vector and unfortunately, it is not exempted from Angular applications. Cross-site scripting also called XSS takes place when a malicious attacker injects scripts into the pages the other users visit. XSS vulnerability, when used in the context of consumption of REST APIs by SPAs, can result in exposure of fresh contents, session hijacking, or any other security attack.

XSS attack protection is inherently included in Angular, that is protected by template syntax, and binding expressions. It is important to note, by default, Angular marks all values as being untrusted and it's sanitizing and escaping of untrusted values. But, developers should be careful and need to avoid the XSS attack on their applications as much as possible.

One of them is not to utilize the methods such as bypassSecurityTrustHtml() when it is possible since such methods evade circumventing Angular's sanitization. When dynamic content has to be generated, then the input should be sanitized effectively. Also, while using APIs that return HTML or JavaScript code, developers should be careful so that such content should be sanitized before being rendered in an application.

The use of CSP headers is another way of guarding against XSS attacks as well as other types of attacks. Thus there is another source of content allowed to be loaded and executed by the browser, CSP (Content Scripting Policy), which enable developers to specify which sources may load execute content.

If one reflects on the security aspects aforementioned that are CORS, JWT implementation, and Basic XSS,

developers can boost the security of Angular apps interacting with REST APIs tremendously. Security also remains a constant and continuous endeavor, therefore maintaining a good practice of constant update on new security practices and possible URL vulnerability is very important should one wants to have strongly fortified web applications.

11. Deployment and Continuous Integration

11.1 Building and Bundling Angular Applications

This paper will focus on the process of building and bundling the Angular applications for deployment, which is an important step in the Angular applications. To this end, Angular CLI offers robust facilities for tasks such as creating optimized production builds, consequently requiring little configuration. When the ng build command is executed with the --prod flag, it will perform several optimizations such as the ahead-of-time or AOT, the tree shaking, and the minification.

Angular's AOT compilation works by converting Angular code from HTML and TypeScript to optimized JavaScript at the time of compiling and before the browser has a chance to download and then run the code. This process increases performance at runtime and decreases the application's size which has several benefits. Another important optimization strategy that is a part of tree shaking process is the process of excluding unutilized codes from the last bundle, which in this time makes the size of the application even smaller. It removes all forms of whitespacing, comments and other forms of characters that are not necessary in the working of the code the process makes the files smaller hence takes lesser time to load.

While using Angular for the implementation of application that interact with REST API, it is necessary to take into account the settings specific to the environment. Environment files in Angular provides the setting for development, staging and production seasons. This is even for API endpoints, and endpoint may differ depending on the environment it serves. The use of environment files makes it possible for the application to call the right API endpoints depending on the environment of the application.

11.2 Containerization with Docker

The concept of containerization has earned a lot of traction as a way of deploying web applications and goes for Angular applications relying on REST APIs. Docker as one of the most popular tools in the sphere of containerization helps developers place their applications and all the necessary components into containers. These containers can then be consistently deployed across different environments – from development, to test, to live.

Docker configuration for Angular applications is common to develop a multi-stage build process. The first stage use a Node; the second, a node and a delay; the third, an average four Nodes, two delays, and a sum; the fourth, a node, a sum, and an operator; the fifth, a node and an operator; and the sixth a node. js base image to make your Angular to build web application, the second build stage uses a light HTTP server like Nginx for serving static files. This leads to a smaller lasting image hence the build tools are not included in the production container.

In this article, containerization has various benefits when it comes to applications deployment particularly in angular. It provides uniformity of the applications across these environments, ease the application deployment process and the applications can be scaled out. In the case of a REST API, containerization can also be helpful when creating development environments precariously similar to production, though mock API servers may be necessary.

11.3 CI/CD Pipelines for Angular Projects

Continuous Integration and Continuous Deployment (CI/CD) need to be adopted in order to build and deploy high quality Angular application interfacing with REST APIs. CI/CD pipelines entail the process where code is built and tested to prove that the changes that are to be incorporated into production applications are okay.

A typical CI/CD pipeline for an Angular project might include the following stages:

1. Serve from the Version Control System
2. Installation of dependencies
3. Running unit tests
4. Running end-to-end tests
5. He or she constrains constructing the floor of the production version of the application.
6. Secure software and analyzing the code
7. Generating to a staging environment
8. Performing integration tests using the staging environment
9. Deploying to production

The mentioned pipelines can be run with the help of such CI/CD tools as Jenkins, GitLab CI/CD or GitHub Actions. In web development it is always a best practice to have API integration tests to prevent frontend and backend interferences when working on REST APIs.

Feature flags can also be helpful in a CI/CD environment in which deployment can be done on a different environment while customers on the current environment are still using previous versions. Anchors let developers turn on and off features on the client side without making new code changes, which is especially valuable in case of branching off new API connections or optional features that refer to backend adjustments.

The CI/CD pipelines should be reliable, in order to instill confidence in the development teams, contain and fix issues before being deployed and make deployment to the users more more frequent and frequent. It is always relevant for Angular applications that depend on REST APIs because it makes a guarantee that changes to the frontend do not harm the backend services during the development and deployment phases.

12. Future Trends and Considerations

12.1 GraphQL as an Alternative to REST

Because web applications are becoming increasingly common, development professionals are always in search of innovative methods to work with data retrieval and processing. GraphQL is an up-and-coming newer API that was developed by Facebook and can execute as an alternative to the RESTful API. GraphQL works in contrast to REST which often uses a number of endpoints for various data requirements in a singular call for data.

Therefore, switching to use GraphQL as a data access technique in Angular applications can enhance the loading of data, minimize the instance of overloading data in the application and enhance the flexibility in the design of APIs more than using libraries. Such libraries as Apollo Client help to use GraphQL in Angular more comfortably, offer functions such as caching, optimistic updates, working with real-time data using subscriptions. There is a great tendency that new projects, and software developers and organizations in general will turn their backs to REST APIs as a protocol and switch to GraphQL, at least for specific types of applications that require a lot of data, or where the bandwidth consumption is a problem.

But before that let me remind you that GraphQL is not the silver bullet that solves all problems. REST APIs are not entirely anachronistic yet; while APIs are increasingly being developed with GraphQL, it is still practical to use REST architecture for very basic applications or where a lot of work has already been invested in REST. For any organisation, the adoption of GraphQL should stem from the projects requirements, the skills within the project team, and the architecture of the system.

12.2 Micro Frontends

Micro frontends are becoming popular as a concept to grow the frontend in large organizations. Building upon the microservices architecture on the backend, micro frontends similar to microservices is the idea of decomposing a large web application into micro frontends that are more manageable, testable and can be deployed independently.

In terms of micro frontend architecture for Angular applications that consumes REST APIs means breaking down the application as different Angular applications each serving a sub-application or feature. These micro frontends could possibly be developed to use a different REST APIs or microservices and hence, the freedom to make changes is now not limited to Front End development alone but can equally apply to the back End developers.

The use of micro frontends in Angular can be done in various approaches including web components, iframes, and runtime through JavaScript. Some of the micro frontend frameworks discussed above include, Angular Elements where developers can build Angular component as a custom element which makes it easy in deploying them in large scale applications.

For all its virtues where scalability and team autonomy are concerned, micro frontends do present pertinent problems with regard to consistency of the user interface, state sharing, and build/deploy operations. With this trend, more technical and efficient solutions as well as benchmarking practices can be foreseen in handling these issues.

12.3 Web Components and Angular Elements

Web Components can be described as the collection of web platform APIs designed for the purpose of enabling application of custom elements in a reusable manner. These components are enclosed or can be reused with any web framework or even with applications that do not employ the use of frameworks. Angular Elements, appeared in v6 of Angular, allows to package Angular components as Web Components and vice versa.

This shift of more interoperable and reuse component may have a big impact on how Angular applications are constructed and how they interact with REST APIs. New modules could be built dedicated to interaction with API, to be used in various projects or even in non-Angular applications. This might translate into more efficient developmental circumstances and even improved reusability of codes.

In addition, as Web Components gain popularity we might find that Angular apps are better built as a composition of loosely coupled reusable components that can be easily composed in to complex apps. This can probably transform the way state management and API integration is done in Angular applications.

Angular developers would also need to acquaint themselves with such trends as they continue to progress in theikut market. It is for this reason that more developers are expected to turn to web components in the future, creating applications as a composite result of components that are as high-performing as possible and which are easily reusable and composable from similarly high-

quality components no matter their origin. The importance of REST APIs will not be debated but the methods of dealing with them and positioning the applications will probably be different. Keeping track of these trends and comprehending their effects will be important for developers who want to establish current, proficient, and adaptable Angular applications.

Certainly. Here is a conclusion to your research paper that is titled "Integrating REST APIs in Single Page Applications using Angular and TypeScript. :

13. Conclusion

The use of the REST APIs in Single Page Applications of Angular using TypeScript is one of the most important trends in web development. The aim of this research has been to identify the definitions, standards, and trends of the development in this particular field. In this post, we have discussed the great aspects about angular framework, advantages of using typescript for large application and basics of restful architecture. The paper has analyzed key areas of the framework those include state management, performance, securing and deployment.

More often than not, web technologies are rapidly trending as more innovational pieces surface, including the latest ones such as GraphQL, micro frontends, and Web Components. These innovations will certainly revolutionalise how web applications and the associated services are developed and connected. Still, it can be stated that the fundamentals of how to efficiently, securely, and simply maintain applications do not evaporate.

In the development of future web sites, which are web applications, modularity, performance and interoperability should be further promoted. Indeed, as we have analyzed using Angular and TypeScript forms a strong ground for developing such applications and especially when it comes to working with RESTful APIs. Application of recommended practices in design of APIs, state management, and security allows developers to design SPAs that offer capabilities of a modern application while also being designed to scale with changing needs.

It is evident that web development has steadily grown and will not follow a more complex paradigm like some industries; thus, the web development principles and practices analyzed in this paper will be paramount to future developers. The use of REST APIs in the Angular applications enhanced by TypeScript and Angular's capabilities will remain the major focus area for the upcoming years of web development.

References

[1] Angular. (2021). Angular - Introduction to the Angular Docs. https://angular.io/docs

[2] Buna, S. (2019). REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media.

[3] Dayley, B. (2020). Learning Angular: A Hands-On Guide to Angular 2 and Angular 4. Addison-Wesley Professional.

[4] Fain, Y., & Moiseev, A. (2020). Angular Development with TypeScript. Manning Publications.

[5] Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine.

[6] Freeman, A. (2018). Pro Angular 6. Apress.

[7] GraphQL Foundation. (2021). GraphQL: A query language for your API. https://graphql.org/

[8] IETF. (2014). RFC 7519: JSON Web Token (JWT). https://tools.ietf.org/html/rfc7519

[9] Jain, N., Mangal, P., & Mehta, D. (2020). AngularJS: Novice to Ninja. SitePoint.

[10] Microsoft. (2021). TypeScript Documentation. https://www.typescriptlang.org/docs/

[11] Mozilla Developer Network. (2021). Cross-Origin Resource Sharing (CORS). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[12] NgRx. (2021). NgRx: Reactive State for Angular. https://ngrx.io/

[13] OWASP. (2021). OWASP Top Ten. https://owasp.org/www-project-top-ten/

[14] Panda, S. (2018). Angular 6 for Enterprise-Ready Web Applications. Packt Publishing.

[15] Podila, P. (2018). REST API Design Best Practices Handbook. API-University Press.

[16] Rozentals, N. (2020). Mastering Angular: Explore powerful techniques to build Enterprise-grade applications. Packt Publishing.

[17] Seemann, M. (2019). Dependency Injection Principles, Practices, and Patterns. Manning Publications.

[18] Smith, S. (2020). Angular Security: Implementing Best Practices. Packt Publishing.

[19] Wasson, M. (2020). ASP.NET Core and Angular: Full-stack web development with .NET 5 and Angular 11. Packt Publishing.

[20] W3C. (2021). Web Components. https://www.w3.org/standards/techs/components#w3c_all