Angular Services - Detailed Concepts

1. Minimal Angular Dev Environment Setup

To begin working with Angular Injectable services or any Angular application, a minimal development environment setup is essential. Here are the steps:

1. Node.js Installation (using NVM):
   - Install Node.js version 18 using nvm:
     ```
     $ nvm install 18
     ```
   - Set Node.js 18 as the default:
     ```
     $ nvm alias default 18
     ```

2. Angular CLI Installation:
   - Install Angular CLI globally:
     ```
     $ npm install -g @angular/cli
     ```
   - Verify the installation:
     ```
     $ ng version
     ```

3. Global npm Path Configuration:
   - Add global npm binary path to ~/.zshrc:
     ```
     export PATH=$PATH:$(npm config get prefix)/bin
     ```
   - Reload Zsh configuration:
     ```
     $ source ~/.zshrc
     ```

4. Project Scaffolding:
   - Create a new Angular project:
     ```
     $ ng new angular-service-demo
     ```

5. Running the Project:
   - Start development server:
     ```
     $ ng serve
     ```
   - Visit: http://localhost:4200

2. Default Angular Services

Angular provides many built-in services, categorized by functionality. These services are singleton instances and made available throughout the application using Angular's DI system.

Key Categories:

- Platform Browser: Title, Meta, DomSanitizer
- HTTP: HttpClient, HttpHeaders, HttpParams
- Routing: Router, ActivatedRoute
- Forms: FormBuilder, Validators
- Observables: Observable, Subject
- Others: Renderer2, ElementRef, DOCUMENT

Example:

Inject Title in constructor:

```
constructor(private titleService: Title) {}
```

Use:

```
this.titleService.setTitle('My Page');
```

3. Angular AsyncPipe

The AsyncPipe is used in templates to handle Observable and Promise values. It subscribes automatically and updates the view with emitted values. It also handles unsubscription to prevent memory leaks.

Syntax:

```
*ngIf="user$ | async as user"
<ul>
  <li *ngFor="let item of items$ | async">{{ item }}</li>
</ul>
```

Benefits:

- No manual subscription/unsubscription
- Cleaner template
- Eliminates risk of memory leaks

Performance Tip:

Avoid multiple async pipes for same Observable. Instead:

```
<ng-container *ngIf="user$ | async as user">
  {{ user.name }}
  {{ user.email }}
</ng-container>
```

4. Custom Services and @Injectable()

The @Injectable() decorator tells Angular to use DI on a class. Services are created using Angular CLI:

```
$ ng generate service services/logger
```

Example:

```
@Injectable({ providedIn: 'root' })
export class LoggerService {
  log(msg: string) { console.log(msg); }
}
```

Scopes of providedIn:

- 'root': Singleton across the app
- 'platform': Shared across Angular apps
- 'any': New instance per lazy module
- In component: Scoped to that component only

Best Practice:

Always use providedIn: 'root' for singleton services unless scoping is needed.

5. Fetching Data via HttpClient

To fetch data from a REST API:

1. Import HttpClientModule in AppModule:

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

```
@NgModule({ imports: [ HttpClientModule ] })
```

2. Create Service:

```
@Injectable({ providedIn: 'root' })
export class DataService {
  constructor(private http: HttpClient) {}
  getPosts(): Observable<Post[]> {
    return this.http.get<Post[]>('https://jsonplaceholder.typicode.com/posts');
  }
}
```

3. Use in Component:

```
posts$ = this.dataService.getPosts();
```

Template:

```
<ul>
  <li *ngFor="let post of posts$ | async">{{ post.title }}</li>
</ul>
```

6. Modifying Data with HttpClient (CRUD)

Services can use full CRUD functionality via HttpClient.

1. CREATE:

```
create(post: Post): Observable<Post> {
  return this.http.post<Post>(API_URL, post);
}
```

2. READ:

```
getAll(): Observable<Post[]> { ... }
getById(id: number): Observable<Post> { ... }
```

3. UPDATE (PUT):

```
update(post: Post): Observable<Post> {
  return this.http.put<Post>(`${API_URL}/${post.id}`, post);
}
```

4. PATCH (partial update):

```
patch(id: number, partial: Partial<Post>) { ... }
```

5. DELETE:

```
delete(id: number): Observable<any> {
  return this.http.delete(`${API_URL}/${id}`);
}
```

Best Practices:

- Always use Observable<T>
- Add error handling with catchError
- Keep services stateless
- Abstract business logic from components
- Use interfaces for type safety