Angular

Qinshift Academy

HTTP

Qinshift Academy

Understanding communicating with backend services using HTTP

Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the HttpClient service class in @angular/common/http.

HTTP client service features

The HTTP client service offers the following major features:

- The ability to request typed response values
- Streamlined error handling
- Request and response interception
- Robust testing utilities

Providing HttpClient through dependency injection

HttpClient is provided using the provideHttpClient helper function, which most apps include in the application providers in app.config.ts.

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(),
  ]
};
```

Providing HttpClient through dependency injection

If your app is using NgModule-based bootstrap instead, you can include provideHttpClient in the providers of your app's NgModule:

```
@NgModule({
  providers: [
    provideHttpClient(),
  ],
  // ... other application configuration
})
export class AppModule {}
```

Providing HttpClient through dependency injection

You can then inject the HttpClient service as a dependency of your components, services, or other classes:

```
@Injectable({providedIn: 'root'})
export class ConfigService {
  constructor(private http: HttpClient) {
    // This service can now make HTTP requests via `this.http`.
  }
}
```

Making HTTP requests

HttpClient has methods corresponding to the different HTTP verbs used to make requests, both to load data and to apply mutations on the server. Each method returns an RxJS Observable which, when subscribed, sends the request and then emits the results when the server responds.

Note: Observables created by HttpClient may be subscribed any number of times and will make a new backend request for each subscription.

Through an options object passed to the request method, various properties of the request and the returned response type can be adjusted.

Fetching JSON data

Fetching data from a backend often requires making a GET request using the HttpClient.get() method. This method takes two arguments: the string endpoint URL from which to fetch, and an optional options object to configure the request.

For example, to fetch configuration data from a hypothetical API using the HttpClient.get() method:

```
http.get<Config>('/api/config').subscribe(config => {
  // process the configuration.
});
```

Note the generic type argument which specifies that the data returned by the server will be of type Config. This argument is optional, and if you omit it then the returned data will have type any.

Mutating server state

Server APIs which perform mutations often require making POST requests with a request body specifying the new state or the change to be made.

The HttpClient.post() method behaves similarly to get(), and accepts an additional body argument before its options:

```
http.post<Config>('/api/config', newConfig).subscribe(config => {
  console.log('Updated config:', config);
});
```

Best practices

While HttpClient can be injected and used directly from components, generally we recommend you create reusable, injectable services which isolate and encapsulate data access logic. For example, this UserService encapsulates the logic to request data for a user by their id:

```
@Injectable({providedIn: 'root'})
export class UserService {
  constructor(private http: HttpClient) {}

  getUser(id: string): Observable<User> {
    return this.http.get<User>(`/api/user/${id}`);
  }
}
```

Best practices

Within a component, you can combine @if with the async pipe to render the UI for the data only after it's finished loading:

```
import { AsyncPipe } from '@angular/common';
@Component({
  standalone: true,
  imports: [AsyncPipe],
  template: `
    @if (user$ | async; as user) {
      <p>Name: {{ user.name }}</p>
      <p>Biography: {{ user.biography }}</p>
    }
  `,
})
export class UserProfileComponent {
  @Input() userId!: string;
  user$!: Observable<User>;

  constructor(private userService: UserService) {}

  ngOnInit(): void {
    this.user$ = userService.getUser(this.userId);
  }
}
```

Questions?

Trainer Name

Trainer

trainer@mail.com

Assistant Name

Assistant

assistant@mail.com

Qinshift Academy