Design Patterns in Angular (part I)

Armen Vardanyan

Original cover photo by Caspar Camille Rubin on Unsplash.

Every experienced developer has at least some design patterns that they have heard about. But the common stereotype is that in *front end development* no one has ever used them. Today, let's dive into the design patterns that are either already being used in Angular development, or, even better, can be utilized to overcome common challenges.

Singleton

Singleton is a design pattern in which a certain class can have only one instance. This is useful when you need to have a single instance of a class, but you don't want to create a new instance every time you need it, and also useful if we want to share resources or data.

If you are using Angular's Dependency Injection, you are already using the singleton pattern, especially if you provide your services with `providedIn: root`. If we provide the service in a certain `NgModule` than it will be a "singleton" only in the scope of that certain `NgModule`.

Factory

A Factory is a design pattern that can create objects with the same interface (or extending from the same class) but with different implementations depending on the context. You might be familiar with the `useFactory` option when providing a service in Angular's DI. This is essentially utilizing that very design pattern. In my article "Angular Dependency Injection Tips" I provide an example of how to use the `useFactory` option to provide different implementations of a logger service. Here is the factory function if you don't want to read the entire article:

```
export function loggerFactory(
  environment: Environment,
  http: HttpClient,
): LoggerService {
  switch (environment.name) {
    case 'develop': {
      return new DevelopLoggerService();
    }
    case 'qa': {
      return new QALoggerService(http, environment);
    }
    case 'prod': {
      return new ProdLoggerService(http, environment);
    }
  }
}
```

We use the `environment` variable to determine which implementation of the

`LoggerService` we want to use. Then we provide it using this factory function:

```
@NgModule({
   providers: [
     {
       provide: LoggerService,
       useFactory: loggerFactory,
       deps: [HttpClient, Environment],
       // we tell Angular to provide this dependencies
       // to the factory arguments
    },
     {provide: Environment, useValue: environment}
   ],
   // other metadata
})
export class AppModule { }
```

You can read a more detailed explanation of how this works in the article.

Using design patterns for specific issues

Now, let us move on other design patterns and discuss how they can be used to address certain challenges. We will take a look at the following:

- Adapter Pattern
- Facade Pattern
- Strategy

Adapter

Adapter is a pattern that allows us to wrap other classes (usually from third parties) in a container class that has a predictable interface and can be easily consumed by our code.

Let's say we are using a third party library that deals with a specific API. It can be something
like Google Cloud, Maps, AWS services or whatever else. We want to be able to unplug that certain class and plug another one when working with the same resource.

An example of this can be when we have a service that provides us data as XML (a SOAP API, for instance), but all our coe consumes JSON, and there is a possibility that in the future, the XML API will be ditched in favor of a JSON one. Let's create an Angular service that can be used to consume the XML API:

```
@Injectable()
export class APIService {

  constructor(
    private readonly xmlAPIService: XmlApiService,
  ) { }

  getData<Result>(): Result {
    return this.xmlAPIService.getXMLData<Result>();
  }

  sendData<DataDTO>(data: DataDTO): void {
    this.xmlAPIService.sendXMLData(data);
  }
}
```

Now, there are several important aspects in the code that we need to pay attention to:

1. The service we wrote does not mention XML, or JSON, or any implementation detail of the API that we are working with
2. The method names are also only reflective of the fact that we deal with some data. What sort of API we are dealing with is unimportant
3. The data types used are also unimportant and not tightly coupled to the implementation - methods are generic
4. We wrap the third-party XML API with this service, so it can be easily replaced in the future

As mentioned in the last point, we only use our service to consume the API, and not the third party library class.
This means, that in the case the XML API gets replaced with a JSON API, we only need to change the service and not the code that uses it. Here is the code changes necessary to switch from XML to JSON:

```
@Injectable()
export class APIService {

  constructor(
    private readonly jsonAPIService: JsonApiService,
  ) { }

  getData<Result>(): Result {
    return this.jsonAPIService.getJSONData<Result>();
  }

  sendData<DataDTO>(data: DataDTO): void {
    this.jsonAPIService.sendJSONData(data);
  }
}
```

As you see, the interface of the service remains **exactly** the same, meaning other services and components that inject
this service will not have to change.

Facade

Facade is a design pattern that allows us to conceal a complex subsystem from the rest of the application. This is useful when we have a large class of group of interacting classes that we want to make easy to use for other services/components.

Facades became increasingly popular with the use of NgRx in Angular apps, when the components now need to deal with dispatching actions, selecting state, and subscribing to specific actions. Here is an example of an Angular component that uses NgRx Store without a facade:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  users$ = this.store.select(selectUsers);
  selectedUser$ = this.store.select(selectSelectedUser);
  query$ = this.store.select(selectQuery);

  constructor(
    private readonly store: Store,
    private readonly actions$: Actions,
    private readonly dialog: DialogService,
  ) { }

  ngOnInit() {
    this.store.dispatch(loadData());

    this.actions$.pipe(
      ofType(deleteUser),
      tap(() => this.dialog.open(
        'Are you sure you want to delete this user?',
      )),
    ).subscribe(() => this.store.dispatch(loadData()));
  }

  tryDeleteUser(user: User) {
    this.store.dispatch(deleteUser({ user }));
  }

  selectUser(user: User) {
    this.store.dispatch(selectUser({ user }));
  }

}
```

Now, this component is dealing with lots of stuff, and is calling `store.dispatch` and `store.select` multiple times, making the code mildly more complex. We would want to have a specific system dedicated to working with just the "Users" part of our `Store`, for example. Let's implement a Facade for this:

```
@Injectable()
export class UsersFacade {

  users$ = this.store.select(selectUsers);
  selectedUser$ = this.store.select(selectSelectedUser);
  query$ = this.store.select(selectQuery);
  tryDeleteUser$ = this.actions$.pipe(
    ofType(deleteUser),
  );

  constructor(
    private readonly store: Store,
    private readonly actions$: Actions,
  ) { }

  tryDeleteUser(user: User) {
    this.store.dispatch(deleteUser({ user }));
  }

  selectUser(user: User) {
    this.store.dispatch(selectUser({ user }));
  }

}
```

Now, let's refactor our component to use this facade:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {

  users$ = this.usersFacade.users$;
  selectedUser$ = this.usersFacade.selectedUser$;
  query$ = this.usersFacade.query$;

  constructor(
    private readonly usersFacade: UsersFacade,
    private readonly dialog: DialogService,
  ) { }

  ngOnInit() {
    this.usersFacade.tryDeleteUser$.subscribe(
      () => this.dialog.open(
        'Are you sure you want to delete this user?',
      ),
    ); // do not forget to unsubscribe
  }

  tryDeleteUser(user: User) {
    this.usersFacade.tryDeleteUser(user);
  }

  selectUser(user: User) {
    this.usersFacade.selectUser(user);
  }

}
```

Strategy

Strategy is a design pattern which allows us to design a system with customizability in mind.
For instance, we can create a library that operates with specific logic, but let's the end user (another developer)
to decide which API to use for that logic.

In some sense, it can be considered an inverse of the Adapter pattern:
in Adapter the end user wraps a third party service in a customizable class, while here with the Strategy
pattern, we are designing the "third party" while allowing the end user to choose which strategy to use.

Imagine we want to create a library that wraps around the `HttpClient`, and we want to allow the end user to choose
which API's to call, how to authenticate, etc. We can create an Angular module and a wrapper class, which would then
provide the functionality, while also allowing an import of a `Strategy` class which will help us decide how to use this wrapper service, what to do when the user is not authenticated, and so on.

First, we need to create a `Strategy` interface which the end user will have to implement:

```
export interface HttpStrategy {
  authenticate(): void;
  isAuthenticated(): boolean;
  getToken(): string;
  onUnAuthorized(): void;
}
```

Then, we need to implement our wrapper:

```
@Injectable({
  providedIn: 'root',
})
export class HttpClientWrapper {

  constructor(
    private readonly http: HttpClient,
    @Inject(STRATEGY) private readonly strategy: HttpStrategy,
  ) { }

  get<Result>(url: string): Observable<Result> {
    return this.http.get<Result>(this.http, url);
  }

  // other methods...
}
```

Now, we have to implement interceptors that will handle authentication errors and send headers to the client:

```
@Injectable({
  providedIn: 'root',
})
export class AuthenticationInterceptor implements HttpInterceptor {

  constructor(
    @Inject(STRATEGY) private readonly strategy: HttpStrategy,
  ) { }

  intercept(
    request: HttpRequest<any>,
    next: HttpHandler,
  ): Observable<HttpEvent<any>> {
    if (this.strategy.isAuthenticated()) {
      request = request.clone({
        setHeaders: {
          Authorization: `Bearer ${this.strategy.getToken()}`,
        },
      });
    }
    return next.handle(request);
  }
}
```

As you can see, we are injecting the `Strategy` class into the `AuthenticationInterceptor` class, so that the end user can decide how to authenticate. They may use `cookies`, `localStorage` or very well another storage for token getting.

Now we also need to implement the interceptor for when we get authorization errors:

```
@Injectable({
  providedIn: 'root',
})
export class UnAuthorizedErrorInterceptor implements HttpInterceptor {

  constructor(
    @Inject(STRATEGY) private readonly strategy: HttpStrategy,
  ) { }

  intercept(
    request: HttpRequest<any>,
    next: HttpHandler,
  ): Observable<HttpEvent<any>> {
    return next.handle(request).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error.status === 401) {
          this.strategy.onUnAuthorized();
        }
        return throwError(error);
      }
      ),
    );
  }
}
```

Here we again inject the `Strategy` class into the `UnAuthorizedErrorInterceptor` class, so that the end user can decide how to handle the error. They may use the Angular `router.navigate` or some `dialog.open` to either redirect the user to the login page or show some popup, or any other scenario. The last bit to do from the "third party" perspective is to create the `NgModule` to encapsulate all of the above:

```
const STRATEGY = new InjectionToken('STRATEGY');

@NgModule({
  imports: [
    HttpClientModule,
  ],
})
export class HttpWrapperModule {

  forRoot(strategy: any): ModuleWithProviders {
    return {
      ngModule: AppModule,
      providers: [
        {
          provide: HTTP_INTERCEPTORS,
          useClass: AuthenticationInterceptor,
          multi: true,
        },
        {
          provide: HTTP_INTERCEPTORS,
          useClass: UnAuthorizedErrorInterceptor,
          multi: true,
        },
        { provide: STRATEGY, useClass: strategy },
        // we use the `InjectionToken`
        // to provide the `Strategy` class dynamically
      ],
    };
  }
}
```

Now the user of this class has to just implement the `HttpStrategy` interface and provide that service when importing the module:

```
@Injectable({
  providedIn: 'root',
})
export class MyStrategy implements HttpStrategy {
  authenticate(): void {
    // do something
  }
  isAuthenticated(): boolean {
    return validateJWT(this.getToken());
  }
  getToken(): string {
    return localStorage.getItem('token');
  }
  onUnAuthorized(): void {
    this.router.navigate(['/login']);
  }

  constructor(
    private readonly router: Router,
  ) { }
}
```

And in the module:

```
import { MyStrategy } from './my-strategy';

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

Now we can also use this wrapper module in another application with a different strategy.

In Conclusion

Design pattern can be an integral part of Angular applications when used properly, so, in the next article, we are going explore some other patterns and their use cases