(that help to scale)

Bartosz Pietrucha • 2 Jul 2019 • #Angular #Architecture

Building scalable software is a challenging task. When we think about scalability in front-end applications, we can think of increasing complexity, more and more business rules, a growing amount of data loaded into the application and large teams often distributed around the world. In order to deal with mentioned factors to maintain a high quality of delivery and prevent technical debt, robust and well-grounded architecture is necessary. Angular itself is a quite opinionated framework, forcing developers to do things *the proper way*, yet there are a lot of places where things can go wrong. In this article, I will present high-level recommendations of well-designed Angular application architecture based on best practices and battle-proven patterns. Our ultimate goal in this article is to learn how to design Angular application in order to maintain **sustainable development speed** and **ease of adding new features** in the long run. To achieve these goals, we will apply:

- proper abstractions between application layers,
- unidirectional data flow,
- reactive state management,
- modular design,
- smart and dumb components pattern.

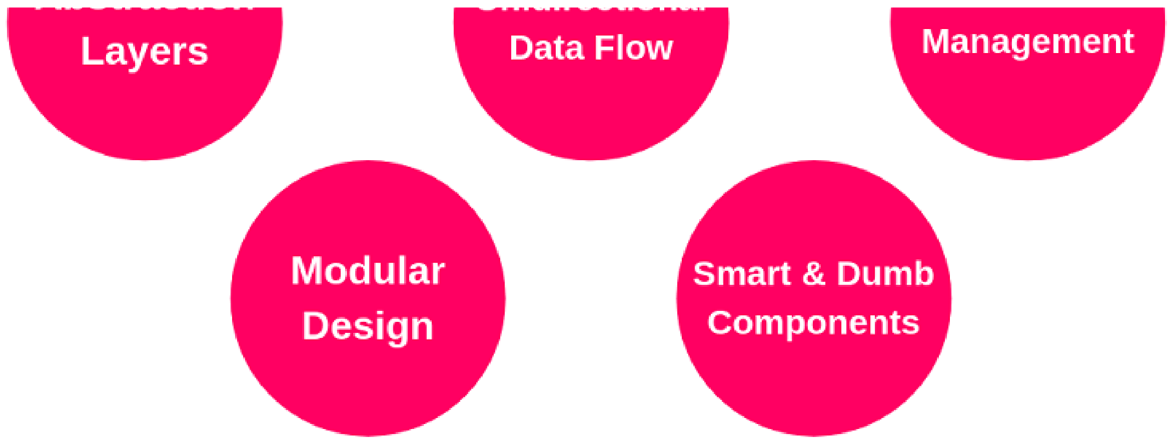
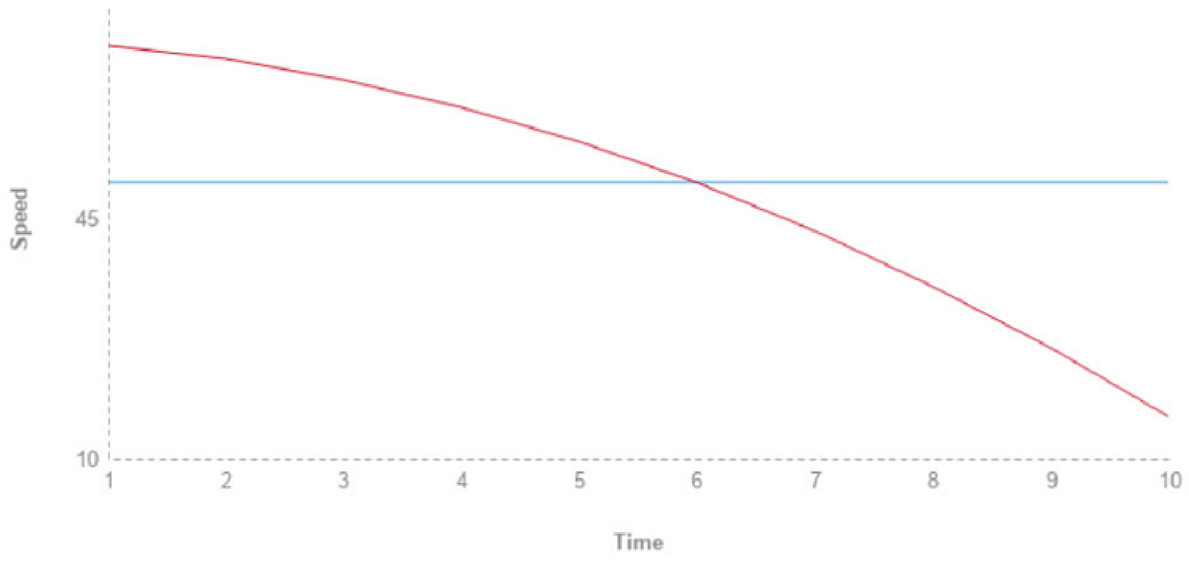


Table of contents

Problems of scalability in front-end

Let’s think about problems in terms of scalability we can face in the development of modern front-end applications. Today, front-end applications are not “just displaying” data and accepting user inputs. Single Page Applications (SPAs) are providing users with rich interactions and use backend mostly as a data persistence layer. This means, far more responsibility has been moved to the front-end part of software systems. This leads to a growing complexity of front-end logic, we need to deal with. Not only the number of requirements grows over time, but the amount of data we load into the application is increasing. On top of that, we need to maintain application performance, which can easily be hurt. Finally, our development teams are growing (or at least rotating - people come and go) and it is important for new-comers to get up to speed as fast as possible.

One of the solutions to the problems described above is solid system architecture. But, this comes with the cost, the cost of investing in that architecture from day one. It can be very tempting for us developers, to deliver new features very quickly, when the system is still very small. At this stage, everything is easy and understandable, so development goes really fast. But, unless we care about the architecture, after a few developers rotations, tricky features, refactorings, a couple of new modules, the speed of development slows down radically. Below diagram presents how it usually looked like in my development career. This is not any scientifical study, it’s just how I see it.



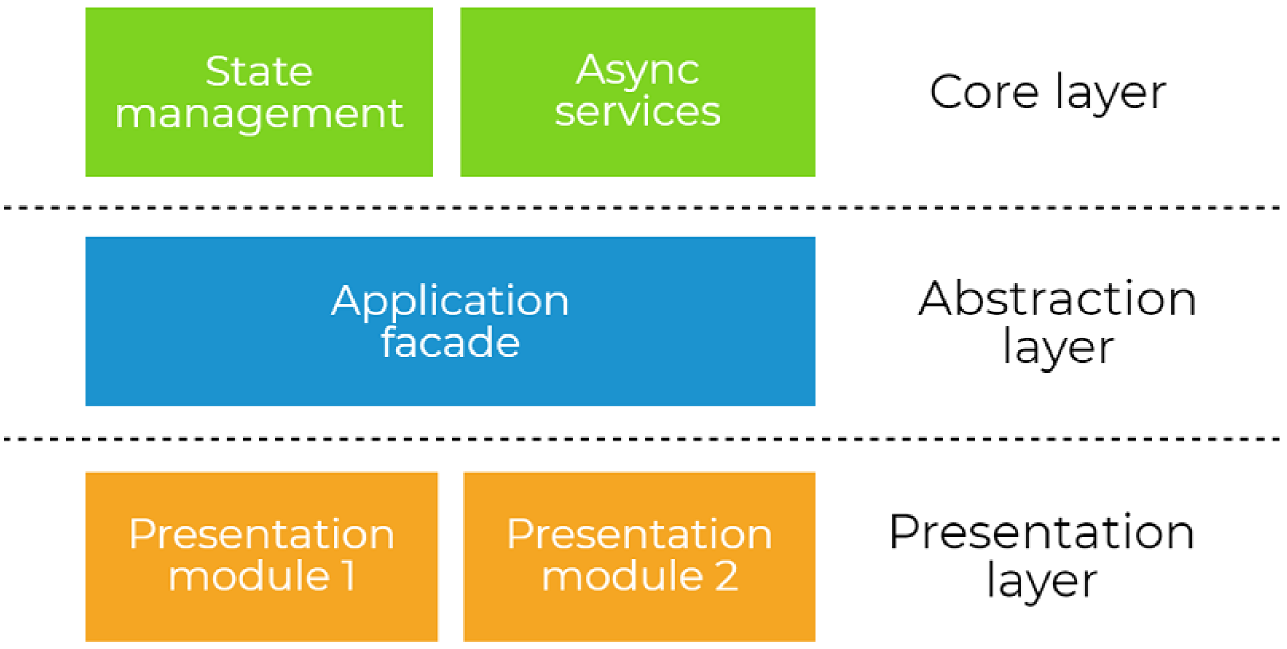
Software architecture

To discuss architecture best practices and patterns, we need to answer a question, what the software architecture is, in the first place. Martin Fowler defines architecture as "*highest-level breakdown of a system into its parts*". On top of that, I would say that software architecture describes how the software is composed of its parts and what are the *rules* and *constraints* of the communication between those parts. Usually, the architectural decisions that we make in our system development, are hard to change as the system grows over time. That's why it is very important to pay attention to those decisions from the very beginning of our project, especially if the software we build is supposed to be running in production for many years. Robert C. Martin once said: the true cost of software is its maintenance. Having well-grounded architecture helps to reduce the costs of the system's maintenance.

> **Software architecture** is the way the software is composed of its parts and the *rules* and *constraints* of the communication between those parts

High-level abstraction layers

The first way, we will be decomposing our system, is through the abstraction layers. Below diagram depicts the general concept of this decomposition. The idea is to place the **proper responsibility** into the **proper layer** of the system: **core**, **abstraction** or **presentation** layer. We will be looking at each layer independently and analyzing its responsibility. This division of the system also dictates communication rules. For example,



Presentation layer

Let’s start analyzing our system break-down from the presentation layer. This is the place where all our Angular components live. The only responsibilities of this layer are to **present** and to **delegate**. In other words, it presents the UI and delegates user’s actions to the core layer, through the abstraction layer. It knows **what** to display and **what** to do, but it does not know **how** the user’s interactions should be handled.

Below code snippet contains `CategoriesComponent` using `SettingsFacade` instance from abstraction layer to delegate user’s interaction (via `addCategory()` and `updateCategory()`) and present some state in its template (via `isUpdating$`).

```
@Component({
  selector: 'categories',
  templateUrl: './categories.component.html',
  styleUrls: ['./categories.component.scss']
})
export class CategoriesComponent implements OnInit {

  @Input() cashflowCategories$: CashflowCategory[];
  newCategory: CashflowCategory = new CashflowCategory();
  isUpdating$: Observable<boolean>;
```

```
    }

    ngOnInit() {
      this.settingsFacade.loadCashflowCategories();
    }

    addCategory(category: CashflowCategory) {
      this.settingsFacade.addCashflowCategory(category);
    }

    updateCategory(category: CashflowCategory) {
      this.settingsFacade.updateCashflowCategory(category);
    }

  }
```

Abstraction layer

The abstraction layer decouples the presentation layer from the core layer and also has it's very own defined responsibilities. This layer exposes the **streams of state** and **interface** for the components in the presentation layer, playing the role of the **facade**. This kind of facade *sandboxes* what components can see and do in the system. We can implement facades by simply using Angular class providers. The classes here may be named with **Facade** postfix, for example `SettingsFacade`. Below, you can find an example of such a facade.

```
@Injectable()
export class SettingsFacade {

  constructor(private cashflowCategoryApi: CashflowCategoryAp

  isUpdating$(): Observable<boolean> {
    return this.settingsState.isUpdating$();
  }

  getCashflowCategories$(): Observable<CashflowCategory[]> {
    // here we just pass the state without any projections

  loadCashflowCategories() {
    return this.cashflowCategoryApi.getCashflowCategories()
      .pipe(tap(categories => this.settingsState.setCashflow(
  }

  // optimistic update
  // 1. update UI state
  // 2. call API
  addCashflowCategory(category: CashflowCategory) {
    this.settingsState.addCashflowCategory(category);
    this.cashflowCategoryApi.createCashflowCategory(category)
      .subscribe(
        (addedCategoryWithId: CashflowCategory) => {
          // success callback - we have id generated by the s
          this.settingsState.updateCashflowCategoryId(categor
        },
        (error: any) => {
          // error callback - we need to rollback the state c
          this.settingsState.removeCashflowCategory(category)
          console.log(error);
        }
      );
  }

  // pessimistic update
  // 1. call API
  // 2. update UI state
  updateCashflowCategory(category: CashflowCategory) {
    this.settingsState.setUpdating(true);
    this.cashflowCategoryApi.updateCashflowCategory(category)
      .subscribe(
        () => this.settingsState.updateCashflowCategory(categ
        (error) => console.log(error),
        () => this.settingsState.setUpdating(false)
      );
  }
}
```

We already know the main responsibilities for this layer; to expose streams of state and interface for the components. Let's start with the interface. Public methods `loadCashflowCategories()` , `addCashflowCategory()` and `updateCashflowCategory()` abstract away the details of state management and the external API calls from the components. We are not using API providers (like `CashflowCategoryApi`) in components directly, as they live in the core layer. Also, how the state changes are not a concern of the components. The presentation layer should not care about **how** things are done and components should **just call** the methods from the abstraction layer when necessary (delegate). Looking at the public methods in our abstraction layer should give us a quick insight about **high-level use cases** in this part of the system.

But we should remember that the abstraction layer is not a place to implement business logic. Here we just want to *connect* the presentation layer to our business logic, abstracting *the way* it is connected.

State

When it comes to the state, the abstraction layer makes our components independent of the state management solution. Components are given Observables with data to display on the templates (usually with `async` pipe) and don't care how and where this data comes from. To manage our state we can pick any state management library that supports RxJS (like NgRx) or simple use BehaviorSubjects to model our state. In the example above we are using state object that internally uses BehaviorSubjects (state object is a part of our core layer). In the case of NgRx, we would dispatch actions for the store.

Having this kind abstraction gives us a lot of flexibility and allows to change the way we manage state not even touching the presentation layer. It's even possible to seamlessly migrate to a real-time backend like Firebase, making our application **real-time**. I personally like to start with BehaviorSubjects to manage the state. If later, at some point in the development of the system, there is a need to use something else, with this kind of architecture, it is very easy to refactor.

Synchronization strategy

Now, let's take a closer look at the other important aspect of the abstraction layer. Regardless of the state management solution we choose, we can implement UI updates (for example with HTTP request) and in case of success we update the state in the frontend application. On the other hand, in an optimistic approach, we do it in a different order. First, we assume that the backend update will succeed and update frontend state immediately. Then we send request to update server state. In case of success, we don't need to do anything, but in case of failure, we need to rollback the change in our frontend application and inform the user about this situation.

> **Optimistic update** changes the UI state first and attempts to update the backend state. This provides a user with a better experience, as he does not see any delays, because of network latency. If backend update fails, then UI change has to be rolled back.
> **Pessimistic update** changes the backend state first and only in case of success updates the UI state. Usually, it is necessary to show some kind of spinner or loading bar during the execution of backend request, because of network latency.

Caching

Sometimes, we may decide that the data we fetch from the backend will not be a part of our application state. This may be useful for **read-only** data that we don't want to manipulate at all and just pass (via abstraction layer) to the components. In this case, we can apply data caching in our facade. The easiest way to achieve it is to use `shareReplay()` RxJS operator that will *replay* the last value in the stream for each new subscriber. Take a look at the code snippet below with `RecordsFacade` using `RecordsApi` to fetch, cache and filter the data for the components.

```
@Injectable()
export class RecordsFacade {

  private records$: Observable<Record[]>;

  constructor(private recordApi: RecordApi) {
    this.records$ = this.recordApi
        .getRecords()
        .pipe(shareReplay(1)); // cache the data
  }
  }

  // project the cached data for the component
  getRecordsFromPeriod(period?: Period): Observable<Record[]
    return this.records$
      .pipe(map(records => records.filter(record => record.i
  }

  searchRecords(search: string): Observable<Record[]> {
    return this.recordApi.searchRecords(search);
  }
}
```

To sum up, what we can do in the abstraction layer is to:

- expose methods for the components in which we:
  - delegate logic execution to the core layer,
  - decide about data synchronization strategy (optimistic vs. pessimistic),
- expose streams of state for the components:
  - pick one or more streams of UI state (and combine them if necessary),
  - cache data from external API.

As we see, the abstraction layer plays an important role in our layered architecture. It has clearly defined responsibilities what helps to better understand and reason about the system. Depending on your particular case, you can create one facade per Angular module or one per each entity. For example, the `SettingsModule` may have single `SettingsFacade`, if it's not too bloated. But sometimes it may be better to create more-granular abstraction facades for each entity individually, like `UserFacade` for `User` entity.

Do you need project consulting?
Angular Academy can help!

Core layer

definition, actions and reducers. Since in our examples we are modeling state with BehaviorSubjects, we can encapsulate it in a convenient state class. Below, you can find `SettingsState` example from the core layer.

```
@Injectable()
export class SettingsState {

  private updating$ = new BehaviorSubject<boolean>(false);
  private cashflowCategories$ = new BehaviorSubject<Cashflow

  isUpdating$() {
    return this.updating$.asObservable();
  }

  setUpdating(isUpdating: boolean) {
    this.updating$.next(isUpdating);
  }

  getCashflowCategories$() {
    return this.cashflowCategories$.asObservable();
  }

  setCashflowCategories(categories: CashflowCategory[]) {
    this.cashflowCategories$.next(categories);
  }

  addCashflowCategory(category: CashflowCategory) {
    const currentValue = this.cashflowCategories$.getValue()
    this.cashflowCategories$.next([...currentValue, category
  }

  updateCashflowCategory(updatedCategory: CashflowCategory)
    const categories = this.cashflowCategories$.getValue();
    const indexOfUpdated = categories.findIndex(category =>
    categories[indexOfUpdated] = updatedCategory;
    this.cashflowCategories$.next([...categories]);
  }

```

```
      categories[updatedCategoryIndex] = addedCategoryWithId;
      this.cashflowCategories$.next([...categories]);
    }

    removeCashflowCategory(categoryRemove: CashflowCategory) {
      const currentValue = this.cashflowCategories$.getValue()
      this.cashflowCategories$.next(currentValue.filter(catego
    }
  }
```

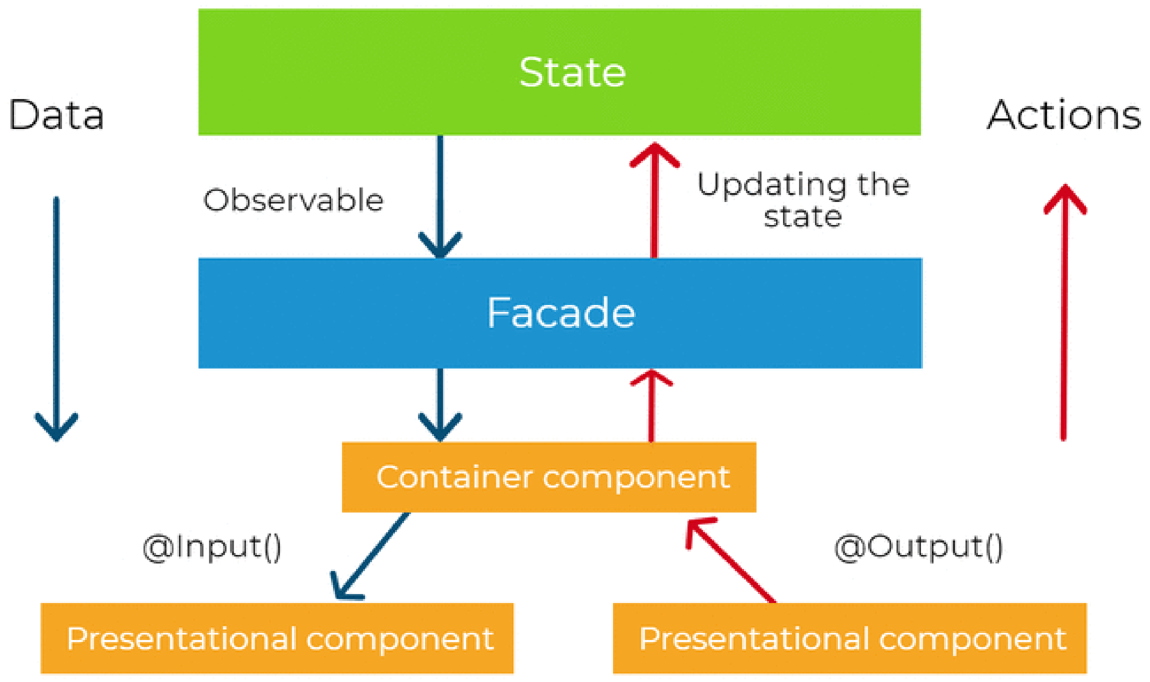
In the core layer, we also implement HTTP queries in the form of class providers. This kind of class could have `Api` or `Service` name postfix. API services have only one responsibility - it is just to communicate with API endpoints and nothing else. We should avoid any caching, logic or data manipulation here. A simple example of API service can be found below.

```
@Injectable()
export class CashflowCategoryApi {

  readonly API = '/api/cashflowCategories';

  constructor(private http: HttpClient) {}

  getCashflowCategories(): Observable<CashflowCategory[]> {
    return this.http.get<CashflowCategory[]>(this.API);
  }

  createCashflowCategory(category: CashflowCategory): Observa
    return this.http.post(this.API, category);
  }

  updateCashflowCategory(category: CashflowCategory): Observa
    return this.http.put(`${this.API}/${category.id}`, catego
  }

}
```

We have covered the topic of the abstraction layers in our frontend application. Every layer has it's well-defined boundaries and responsibilities. We also defined the strict rules of communication between layers. This all helps to better understand and reason about the system over time as it becomes more and more complex.

Unidirectional data flow and reactive state management

The next principle we want to introduce in our system is about the data flow and propagation of change. Angular itself uses unidirectional data flow on presentation level (via input bindings), but we will impose a similar restriction on the application level. Together with reactive state management (based on streams), it will give us the very important property of the system - **data consistency**. Below diagram presents the general idea of unidirectional data flow.



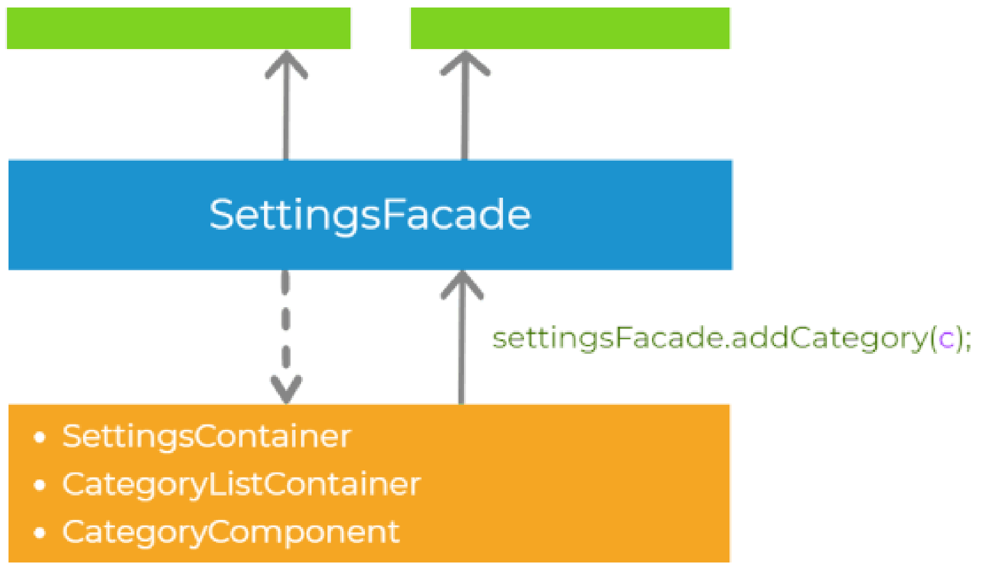
Whenever any model value change in our application, Angular change detection system takes care of the propagation of that change. It does it via input property bindings from **the top to bottom** of the whole component tree. It means that a child component can only depend on its parent and never vice versa. This is why we call it unidirectional data flow. This allows Angular to traverse the components tree **only once** (as there are no cycles in the tree structure) to achieve a stable state, which means that every value in the bindings is propagated.

level? We can place the application data (the state) in one place “above” the components and propagate the values down to the components via Observable streams (Redux and NgRx call this place a store). The state can be propagated to multiple components and displayed in multiple places, but never modified locally. The change may come only “from above” and the components below only “reflect” the current state of the system. This gives us the important system’s property mentioned before - **data consistency** - and the state object becomes **the single source of truth**. Practically speaking, we can *display* the same data in multiple places and not be afraid that the values would differ.

Our state object exposes the methods for the services in our core layer to manipulate the state. Whenever there is a need to change the state, it can happen only by calling a method on the state object (or dispatching an action in case of using NgRx). Then, the change is propagated “down”, via streams, the to presentation layer (or any other service). This way, our state management is *reactive*. Moreover, with this approach, we also increase the level of predictability in our system, because of strict rules of manipulating and sharing the application state. Below you can find a code snippet modeling the state with BehaviorSubjects.

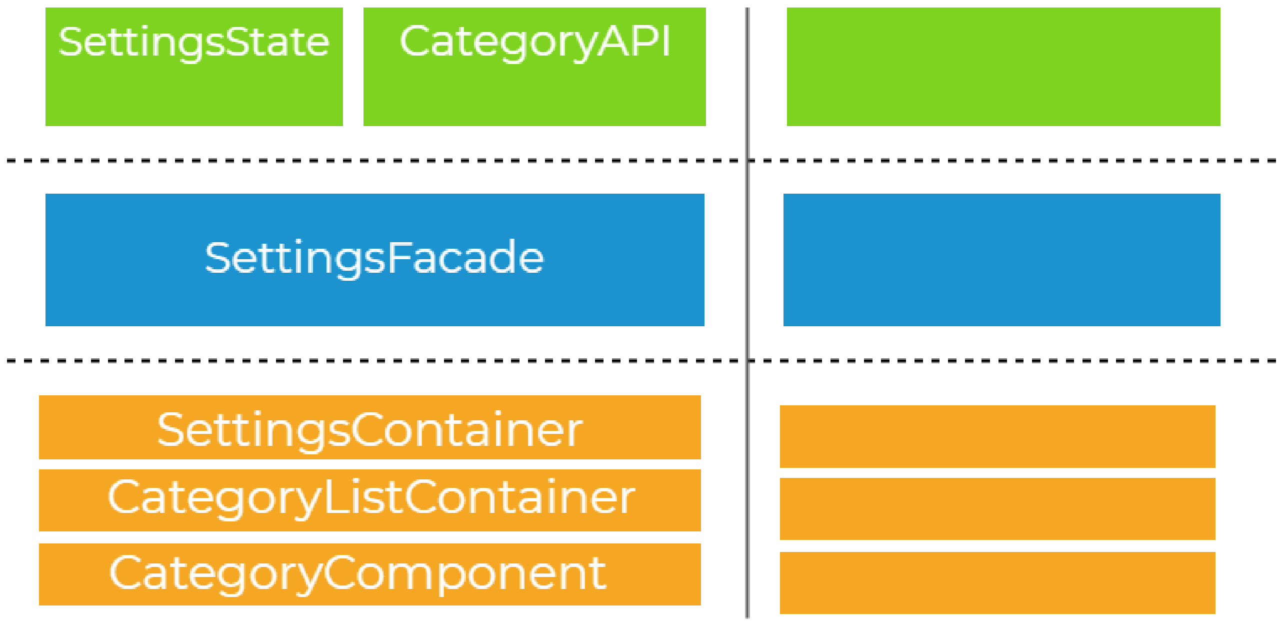
```
@Injectable()
export class SettingsState {

  private updating$ = new BehaviorSubject<boolean>(false);
  private cashflowCategories$ = new BehaviorSubject<Cashflow(

  isUpdating$() {
    return this.updating$.asObservable();
  }

  setUpdating(isUpdating: boolean) {
    this.updating$.next(isUpdating);
  }

  getCashflowCategories$() {
    return this.cashflowCategories$.asObservable();
  }

  setCashflowCategories(categories: CashflowCategory[]) {
    this.cashflowCategories$.next(categories);
    const currentValue = this.cashflowCategories$.getValue()
    this.cashflowCategories$.next([...currentValue, category
  }

  updateCashflowCategory(updatedCategory: CashflowCategory)
    const categories = this.cashflowCategories$.getValue();
    const indexOfUpdated = categories.findIndex(category =>
    categories[indexOfUpdated] = updatedCategory;
    this.cashflowCategories$.next([...categories]);
  }

  updateCashflowCategoryId(categoryToReplace: CashflowCatego
    const categories = this.cashflowCategories$.getValue();
    const updatedCategoryIndex = categories.findIndex(catego
    categories[updatedCategoryIndex] = addedCategoryWithId;
    this.cashflowCategories$.next([...categories]);
  }

  removeCashflowCategory(categoryRemove: CashflowCategory) {
    const currentValue = this.cashflowCategories$.getValue()
    this.cashflowCategories$.next(currentValue.filter(catego
  }
}
```

Let's recap the steps of handling the user interaction, having in mind all the principles we have already introduced. First, let's imagine that there is some event in the presentation layer (for example button click). The component delegates the execution to the abstraction layer, calling the method on the facade `settingsFacade.addCategory()` . Then, the facade calls the methods on the services in the core layer - `categoryApi.create()` and `settingsState.addCategory()` . The order of invocation of those two methods depends on synchronization strategy we choose (pessimistic or optimistic). Finally, the application state is propagated down to the presentation layer via the observable streams. This process is **well-defined**.

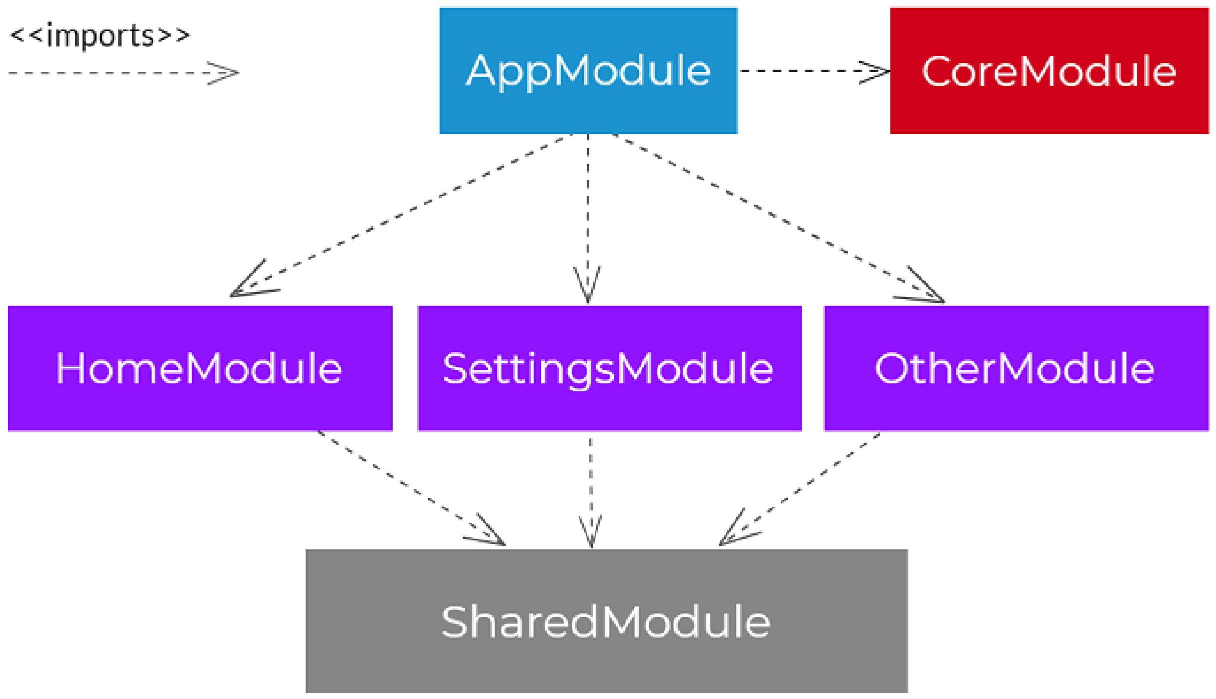


Modular design

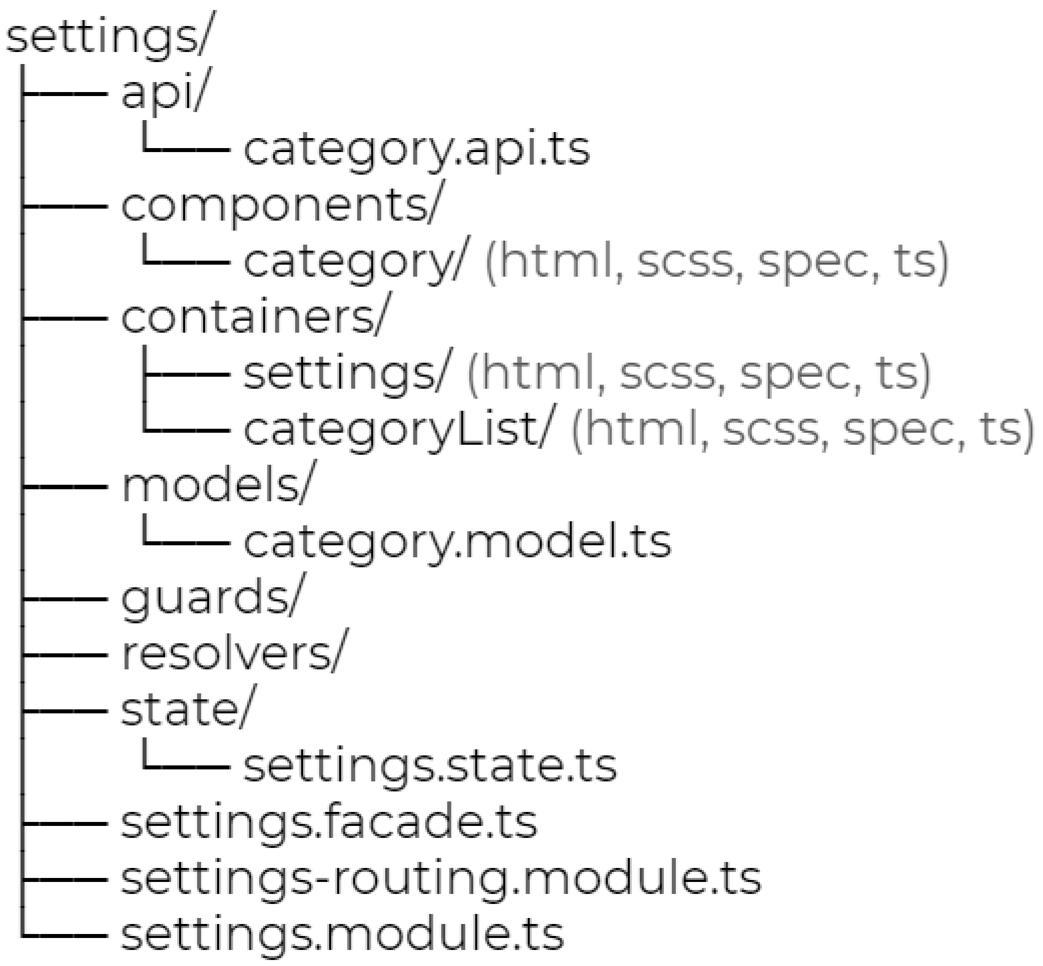
We have covered the horizontal division in our system and the communication patterns across it. Now we are going to introduce a vertical separation into feature modules. The idea is to slice the application into **feature modules** representing different business functionalities. This is yet another step to deconstruct the system into smaller pieces for better maintainability. Each of the features modules share the same horizontal separation of the core, abstraction, and presentation layer. It is important to note, that these modules could be lazily loaded (and preloaded) into the browser increasing the initial load time of the application. Below you can find a diagram illustrating features modules separation.



Our application has also two additional modules for more technical reasons. We have a `CoreModule` that defines our singleton services, single-instance components, configuration, and export any third-party modules needed in `AppModule`. This module is imported **only once** in `AppModule`. The second module is `SharedModule` that contains common components/pipes/directives and also export commonly used Angular modules (like `CommonModule`). `SharedModule` can be imported by any feature module. The diagram below presents the imports structure.



Module directory structure



```
settings/
├── api/
│   └── category.api.ts
├── components/
│   └── category/ (html, scss, spec, ts)
├── containers/
│   ├── settings/ (html, scss, spec, ts)
│   └── categoryList/ (html, scss, spec, ts)
├── models/
│   └── category.model.ts
├── guards/
├── resolvers/
├── state/
│   └── settings.state.ts
├── settings.facade.ts
├── settings-routing.module.ts
└── settings.module.ts
```

Smart and dumb components

The final architectural pattern we introduce in this article is about components themselves. We want to divide components into two categories, depending on their responsibilities. First, are the **smart components** (aka containers). These components usually:

- have facade/s and other services injected,
- communicate with the core layer,
- pass data to the dumb components,
- react to the events from dumb components,
- are top-level routable components (but not always!).

Previously presented `CategoriesComponent` is **smart**. It has `SettingsFacade` injected and uses it to communicate with the core layer of our application.

`me</button>` . That element does not have any particular logic implemented. We can think of the text ‘Click me’ as an input for this component. It also has some events that can be subscribed to, like click event. Below you can find a code snippet of a simple **presentational** component with one input and no output events.

```
@Component({
  selector: 'budget-progress',
  templateUrl: './budget-progress.component.html',
  styleUrls: ['./budget-progress.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class BudgetProgressComponent {

  @Input()
  budget: Budget;
  today: string;

}
```

Summary

We have covered a couple of ideas on how to design the architecture of an Angular application. These principles, if applied wisely, can help to maintain sustainable development speed over time, and allow new features to be delivered easily. Please don’t treat them as some strict rules, but rather recommendations that could be employed when they make sense.

We have taken a close look at the abstractions layers, unidirectional data flow and reactive state management, modular design, and smart/dumb components pattern. I hope that these concepts will be helpful in your projects and, as always, if you have any questions, I am more than happy to chat with you.

At this point, I would like to give a huge *kudos* to Brecht Billiet who wrote this blog post, that introduced me to the idea of Abstraction Layer and Facade. Thanks, Brecht! Big *thank you* also goes to Tomek Sułkowski that reviewed my perspective on a layered architecture.

Angular Academy BLOG WORKSHOPS VIDEOS

25 Comments Angular academy 1 Login

Recommend 15 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Arun Pareek • a month ago
Can you provide github link of this project?
4 • Reply • Share ›

elmota • 10 days ago
ouch my brain hurts :) question: what if, for example on update settings, the presentation component needs to do something, but it must wait for the update to finish, would the current set up handle such a case? how would it look like? (For example on update let's display a success message).
• Reply • Share ›

Bartosz Mod elmota • 10 days ago • edited
Hi @elmota! What do you mean "it must wait for the update to finish"? Do you mean showing a full-screen spinner, preventing user to click anywhere? Or some inside logic? In any case you can track it with some boolean flag, using pessimistic update, waiting for the request to finish. Or you can use `BehaviorSubject` to track it (as in here Angular JWT blog post).
• Reply • Share ›

elmota Bartosz • 9 days ago
I guess that would be what I'm saying, I looked back at your code and I think you are handling a scenario with setUpdating... like I said, my brain hurts :)
• Reply • Share ›

王猛 • 25 days ago
Great article!!!! If you can give me a github link,I will be very grateful.
• Reply • Share ›

Bartosz Mod 王猛 • 24 days ago • edited
Hi! I don't have any repo published for this article, but Angular Full Security

Sorry for the noob question, but can you describe how this is different from using Ngrx ? Or why one would use this vs ngrx ? Also, for the communication between the container and presentation component would that still happen through an input from container to presenter, or can you directly inject the facade into the presentation component as well and pull the value to be displayed from the facade/store as in ngrx ?

Reply • Share ›

Bartosz Mod john • 2 months ago • edited

Hi John, this approach does not prevent you from using NgRx, it only abstracts away the communication with it. You can still use actions and reducers in your core layer, but you do not directly dispatch actions in the presentation layer. Regarding injecting the facade into presentation components, it does not sound like a good idea. You need to carefully consider if the given component is smart or dumb.

Reply • Share ›

Michael Kariv • 3 months ago

Good article. The only thing I would love to do is this. Say I want several apps using the same feature, e.g. one in ionic for mobile and one in bootstrap for desktop. I would use nx for helping managing the separation. The question is then how would a feature module be divided between the app and the lib. I would guess the state, api and facade would be a lib, everything UI - components, containers and routing - in the app. However I would love Mr Pietrucha's thoughts. I am sure he has deeper things to say on the subject.

Reply • Share ›

Julio Cesar Ayasta • 4 months ago

Great article!!!! regards!

Reply • Share ›

Dmitry • 4 months ago

Really great article!

I'd only change the section about SharedModules. There is a well known issue with it. The general recommendation is to have a single SharedModule per app. This module would include all shared components of the app and it would be included in all Feature Modules. In theory this is great. However, in practice FeatureModule1 may use only a subset of these components whereas a FeatureModule2 would use a slightly different subset of the shared components. This means that when FeatureModule1 is lazy loaded it'd require all shared components - even these that it does not use.

This issue is solved by encapsulating each shared component into its own Module and importing these modules wherever they are needed. This idea is explained really well in this article: https://blog.angularindepth.... They are calling it "SCAM modules" :D

Reply • Share ›

individually?

Reply • Share ›

Dmitry akkonrad • 4 months ago

Currently the same component/directive cannot be a part of the two different modules. That's why you have to wrap each into a module. I believe this will be changing after ivy is at 100%

Reply • Share ›

Bartosz Mod Dmitry • 4 months ago

Thanks for your comment! Of course, you can always split a shared module in multiple shared modules, depending on your project structure.
But if you stick to a single shared module, you can always load it eagerly (or preload in background).
Anyway, don't you think that these SCAM modules could bring more work, than the benefit?

Reply • Share ›

Dmitry Bartosz • 4 months ago

There is no doubt that SCAM modules have a lot of overhead. I see them as a path forward to the optional Angular Modules.

Reply • Share ›

Yejiel Wahnich • 4 months ago

I think facade abstraction, needs a bit more than just 1 oover the top post :sweat_smile:

Reply • Share ›

Bartosz Mod Yejiel Wahnich • 4 months ago

What is missing in your opinion :) ?

Reply • Share ›

Michael Kariv Bartosz • 2 months ago • edited

If I may, I love the idea of the abstraction. It is well presented. If there is anything missing, I would say, it is a practical extended example. I would love to see the Real World angular implementation refactored to have the abstraction. Comparing the two would demonstrate vividly what an abstraction did.

But that is not to dis the article in any way. I bookmarked it and keep in the top drawer, alongside the best of them

Reply • Share ›

GLaw • 4 months ago

Do you maybe have a open source project with these principles implemented, so that

Bartosz Mod ➦ GLaw • 4 months ago
I am thinking about creating an online couse covering these topics step by step. What do you think?
4 ^ | ˅ • Reply • Share ›

Taylor A. Buckner • 4 months ago
One question I've often had concerning the shared module approach: "Does importing this module that presumably houses a lot of stuff into a feature module in order to consume one or two things negatively affect performance?"
^ | ˅ • Reply • Share ›

Bartosz Mod ➦ Taylor A. Buckner • 4 months ago • edited
Run-time performance should not be hit, but fetching the bundles from the server may be slower. What you can do is either to split a shared module into multiple shared modules or preload shared module in the background (easily done in Angular).
^ | ˅ • Reply • Share ›

Taylor A. Buckner ➦ Bartosz • 4 months ago
That's pretty cool. I'm more of a fan of the SCAM approach which seems pretty close to what angular material does.

FREE Angular Security Video Lesson

Watch now!

Join **+2100** subscribers
that receive latest Angular knowledge

Your name

Email Address

I want new articles

By clicking the button you agree to receive a newsletter. NO SPAM!

Angular Academy BLOG WORKSHOPS VIDEOS

Bartosz Pietrucha

Angular Academy founder, software consultant, trainer, international speaker

Follow @pietrucha

Subscribe for new articles and videos!

Your name

Email Address

I want new articles

By clicking the button you agree to receive email newsletter. NO SPAM!

© 2019 Angular Academy Terms and Conditions Privacy Policy