Don't clone Back-End models in Angular

Adriano di Lauro Follow
Mar 20, 2018 · 36 min read

In Front-End, Model-View-Controller paradigm has been definitely discarded in favour of Model-Component: this is true in general in all modern frameworks, but it's particularly evident in Angular world, if you look at the evolution from AngularJS to Angular2+.

The main reason of MVC's inadequacy in Front-End has been generally identified in the ambiguous role of controllers, but in this article I want to focus on a different aspect of such inadequacy: ***the different role of models in Back-End and in Front-End.***

Roadmap of the article

The goal of this article is to show how copying Back-End models into an Angular app can lead to architectural problems (and generally, how such problems can be caused by a Back-End approach in Front-End); I will highlight weak points, show patterns, propose solutions.

Since the article is about both Angular and AngularJS, I'll always refer to Angular as "Angular2+", in order to avoid confusion; I'll use the generic term "Angular", without specifying any version, only when I intend to talk about a generic concept that is common to the two frameworks.

Here is a rough roadmap of the article:

1. I'll go through **general architectural differences** between Back-End and Front-End
2. In the specific case of Angular, I'll show the distinction between **different levels of abstraction**, starting from properties we bind to the DOM up until the Back-End models: I'll show how the best way to avoid mistakes is **keeping these abstractions clearly separated**
3. I'll show a couple of simple examples about how bringing unnecessary Back-End variables into Front-End leads to **redundancy** and makes an Angular app error prone
4. I'll present the typical example of a nested JSON returned by the Back-End, and I'll show how a **naif AngularJS implementation** of such JSON hides **unexpected structural problems**
5. I'll propose a different implementation of the nested JSON based on **component pattern in Angular2+**, and show its advantages
6. I'll focus on the best way to pass data from API to components, and show how important is to **decide the architecture of Front-End regardless of the data structure we receive from the Back-End**
7. I'll propose a different way to populate Angular2+ components with data that comes from Back-End API, by using **services instantiated at component level**

8. I'll briefly talk about the best way to send "*/post*" calls to the API while still keeping Front-End separated by Back-End
9. I'll briefly show use cases of Front-End models that have attributes without any correspondence in the Back-End

Intrinsic architectural difference between Back-End and Front-End

Nowadays everybody knows that MVC is not good for Front-End. Many blame controllers for this (among the good articles about controllers out there, I can recommend this one): but the deeper reason why MVC failed must be probably searched in the fact that ***we were trying to apply a Back-End architecture to Front-End***.

But are Back-End and Front-End actually so different?

In my opinion yes, they are: ***the approach to solving a problem in Back-End is intrinsically different from the approach to solving a problem in Front-End.***

- Back-End development has to achieve a logical abstract representation of the ***entities*** involved, in order, for external users and applications, *to be able to perform actions on those entities regardless of the context*
- Front-End development has to achieve a logical abstract representation of the ***views*** involved, in order not only *to interact smoothly with the user*, but also to ensure that the implementation is **optimised** (since it has to run on a browser) and **scalable** (because client's requirements change faster in Front-End than in Back-End).

This difference didn't acquire importance up until approximately 2010, when single page applications started to spread: before single page applications, Front-End consisted simply of CSS, HTML and some sporadic JavaScript, and Front-End development was regarded as a task that didn't require a specific pattern or guideline.

What happens if somebody tries to develop a Back-End using the approach of Front-End? He'll try to infer the logical structure of entities just looking at how they are used in the specific views he has to develop; he'll create only the set of functionalities for what he needs to do, without thinking if those functionalities have deeper consequences in the abstract logic of the entities: the resulting Back-End will be **naif** and **unstable**, everybody has always agreed on this.

***Now, exactly in the same way, if we turn the problem around and try to develop Front-End using the approach of Back-End, the result will be* naif *and* unstable.**

One side of this problem is of course the unclear role of controllers in MVC, as already mentioned; but *in my opinion a deeper and more hidden architectural lack is when a developer with Back-End mentality tries to define a structure of JavaScript* **models** *which are the exact clone of their Back-End counterparts.*

How Back-End and Front-End models diverge in Angular

Both in Back-End and Front-End models are an *abstraction*, but this doesn't imply that they have to be identical: ***the reason is that they are an abstraction of different things.***

The first obvious example of this difference is when a Back-End model contains internal attributes that are not used in the Front-End (and viceversa, a Front-End model might contain attributes that are not relevant in the Back-End).

But in Angular world, where the DOM gets automatically updated as the application's status changes, things become less trivial: what do we mean by "*application's status*"? Where does such status get stored, and how does it relate to models?

In Angular2+, the application status is stored in **components**; but what are actually components, and how do they interact with the DOM? From the official documentation of data binding techniques in Angular2+:

> *When you write a data binding, you're dealing exclusively with properties and events of the target object. […] The target of a data binding is something in the DOM. Depending on the binding type, the target can be an (element | component | directive) property, an (element | component | directive) event, or (rarely) an attribute name.*

So, ***Angular2+'s components are nothing else than a (natural) extension of the DOM***: the DOM can be seen as a hierarchical tree composed of nested native elements and components. Let's see an example:

```
@Component({
  selector: 'text-in-paragraph',
  template: `
    <p>{{text}}</p>
    `
})
export class TextInParagraphComponent {
  @Input() text:string;
}
```

dont-clone-models-1.ts hosted with ♡ by GitHub view raw

When we want to use the component defined in the snippet above, we simply use the tag that identifies it, passing along the input variable "*text*" as if it was a normal HTML attribute:

```
<text-in-paragraph text="Mary had a little lamb">
</text-in-paragraph>
```

dont-clone-models-2.html hosted with ♡ by GitHub view raw

When we use the tag "*text-in-paragraph*", it looks exactly as if it was a regular DOM element: and it can indeed be considered a DOM element, because it behaves exactly in the same way (with a few extensions).

A feature that a custom component has in addition to the DOM, for instance, is *data binding*:

```
<text-with-paragraph [text]="someVariable">
</text-with-paragraph>
```

dont-clone-models-3.html hosted with ♡ by GitHub view raw

When we wrap the attribute "*text*" inside brackets, it means that the expression passed inside corresponds to a variable in the parent component: when the value of that variable changes, Angular2+ updates the child component and reloads its template.

The analogies between actual DOM and components go beyond this simple example (for instance, as in the DOM, components can use alias attributes that bind to a property with a different name), but the core concept is that components can be safely considered an extension of the DOM.

More generally, a tree composed of nested native DOM elements, components, directives, and all their public attributes, is an abstraction of the DOM provided by Angular2+.

So, coming back to the difference between Back-End and Front-End models: *the extension of the DOM in Angular can be considered as a* **third abstraction***, a sort of* **DOM model***, which in turn has to be different from both.*

In AngularJS the concept is the same but the implementation is different: entities responsible for holding variables that are bound to the DOM are called **controllers**. Even though AngularJS controllers are not exactly an extension of the DOM, the best practice is still to keep them separated from models (both Front-End and Back-End).

Keep clearly separated Back-End, Front-End and DOM models

We just distinguished among three different layers of abstraction:

1. The **Back-End model**, that holds the logical and general abstraction of an entity
2. The **Front-End model**, that represents an entity in the Front-End (*in Angular2+ it corresponds to TypeScript Model classes*): many of these models are similar to the Back-End ones, but they are rarely exactly the same, plus in Front-End we might need additional entities used to interact with the user
3. The **DOM model**, that abstracts and extends the DOM (*in Angular2+ it corresponds to Component classes*)

In order to achieve a healthy and scalable architecture, these three abstractions must be kept clearly separated.

When developing an Angular app carelessly and with a Back-End approach, you risk mixing the three abstractions at different degrees, with various side effects:

- If you *create Front-End models that are an exact clone of Back-End models*, you are likely to have problems of **redundancy**, and fat components / controllers that are **difficult to maintain**
- If you *bind the DOM directly to a Front-End model*, you end up producing a **complex HTML**, you risk **losing isolation** of components, and you can experience **unpredictable side effects**
- If you you *bind the DOM directly to a Front-End model that is an exact clone of a Back-End model*, you'll get a bit of both points above

The aim of this article is to highlight risks caused by confusing Back-End, Front-End and DOM models, and propose patterns that avoid these risks by keeping the separation clear. Let's start with some practical example.

Example: unnecessary variables brought along with a Back-End model

We have a user model, represented in the Back-End with two boolean properties called "*subscribed*" and "*confirmed*". These represent two internal states of the Back-End, so that the user is considered a "*pro*" user if and only if both booleans are true.

In the Front-End, "*subscribed*" and "*confirmed*" are not relevant, the only thing that we need to know is whether the user is "*pro*" or not.

However, the Back-End API returns a user JSON which includes both "*subscribed*" and "*confirmed*", so if we run an AngularJS application we are tempted to do something like this:

```
app.service('User', function () {
  var data = getDataFromAPI();
  this.subscribed = data.subscribed;
  this.confirmed = data.confirmed;
  this.isPro = function() {
    return (this.subscribed && this.confirmed);
  }
});
```

dont-clone-models-4.js hosted with ♡ by GitHub view raw

```
<navbar>
  <a class="user-active-{{user.isPro()}}">
    <img src="login-icon.png"/>
  </a>
</navbar>
```

dont-clone-models-5.html hosted with ♡ by GitHub view raw

This implementation is simple, but it contains the seed of potential problems that can affect the application as it becomes larger.

The DOM is bound to the expression "*isPro*" that is not a property of the DOM model but rather a function that calculates its value on the fly, based on two variables of the corresponding Front-End model: this is a potential performance risk because such function has to be called at every "*$digest*" loop.

The Front-End model, in turn, is a copy of the Back-End model, because the variables "*subscribed*" and "*confirmed*" are not directly exposed in the Front-End: every time we want to handle the "*user*" model we have to deal with this redundancy, and this adds unnecessary complexity to the application.

Especially, if the API expects us to send the whole object when creating or modifying a user, the Front-End shouldn't be responsible for remembering to attach both "*subscribed*" and "*confirmed*" to the payload: this should be done by the Back-End.

The lesson to learn from this example is: ***when you design a Front-End model you should reflect carefully before including just any data that comes from the Back-End.***

I have seen Back-End developers who felt comfortable only once they had completely replicated in Front-End the full structure of a Back-End model, sometimes even including a set of unit tests written in JavaScript. This complexity is out of place, because it already exists in the Back-End where it belongs.

A brief note on forms

Forms are a case in which it's particularly easy to mix Front-End model and DOM model, because they do so many things automatically that developers tend to consider them "magic" (forms in Angular bind inputs to your data and keep them updated in more sophisticated ways, as the framework evolves).

Let's hear it from the docs themselves, where there is an explicit warning about distinction between models:

> *The component must copy the hero values in the* data model *into the* form model.
>
> *There are two important implications: the developer must understand how the properties of the* data model *map to the properties of the* form model; and *user changes flow from the DOM elements to the* form model*, not to the* data model.
>
> *The form controls never update the* data model.

In the quote above, "*data model*" corresponds to what in this article I call Front-End model, and "*form model*" to what I call DOM model.

I have so many things to say in this article that there is no space for a detailed example of binding to forms in Angular, you'll get plenty of those in the docs. For the purpose of the article, I'll just remark that Angular's forms are another extension of the DOM.

A meaningful example: Back-End model with nested sub-models

This is the example that I'll follow until the end of the article, because it's controversial and has a lot of implications. I say "controversial" because, at first sight, binding a nested model directly to the DOM might seem a natural choice: but the resulting architecture can have unpredictable problems in terms of scalability (and of performance as well, in particularly bad cases).

The example I want to use is a post with comments and replies; this is the JSON structure that our API returns:

```
this.post = {
  "userId": "2",
  "userName": "John Black",
  "title": "Amazing post",
  "text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit ubi maior minor cessat",
  "comments": [
    {
      "userId": "5",
      "userName": "Mary White",
      "text": "I agree with your post",
      "replies": []
    },
    {
      "userId": "4",
      "userName": "Sebastian Green",
      "text": "Nice post",
      "replies": [
        {
          "userId": "5",
          "userName": "Mary White",
          "text": "I agree",
        }
      ]
    }
  ]
}
```

dont-clone-models-6.json hosted with ♡ by GitHub view raw

First naif implementation (in AngularJS)

The most immediate way to construct a Front-End from this JSON is simply binding it to the DOM, using two nested loops that correspond to comments and replies.

I'll present this implementation using AngularJS, because it doesn't enforce component pattern and it's easier to show how simple this wrong approach apparently seems. The fact that I chose AngularJS for this example doesn't imply that in AngularJS you have to use not performing patterns.

We want to display comments and replies related to a post, and give the possibility to add and delete comments and replies. This is the controller:

```
function postController(apiAdaptor, currentUser) {
  'ngInject';

  this.newComment = this.generateEmptyComment();
  this.newReply = this.generateEmptyReply();

  this.submitNewComment = function() {
    this.post.comments.push(this.newComment);
    this.newComment = this.generateEmptyComment();
    apiAdaptor.post('/posts', this.post);
  };

  this.submitNewReply = function(anIndex) {
    this.post.comments[anIndex].replies.push(this.newReply);
    this.newReply = this.generateEmptyReply();
    apiAdaptor.post('/posts', this.post);
  };

  this.generateEmptyComment = function() {
    return {
      userId: currentUser.getId(),
      userName: currentUser.getName(),
      text: '',
      replies: [],
    };
  };

  this.generateEmptyReply = function() {
    return {
      userId: currentUser.getId(),
      userName: currentUser.getName(),
      text: '',
    };
  };
  this.deleteComment = function(anIndex) {
    this.post.comments.splice(anIndex, 1);
    apiAdaptor.post('/posts', this.post);
  };

  this.deleteReply = function(commentIndex, anIndex) {
    this.post.comments[commentIndex].replies.splice(anIndex, 1);
    apiAdaptor.post('/posts', this.post);
  };
}
```

dont-clone-models-7.js hosted with ♡ by GitHub view raw

And this is the corresponding template, that contains simply two nested loops of "*ng-repeat*" directives.

```
<div class="post">
  <h1>
    {{$ctrl.post.title}} by {{$ctrl.post.userName}}
  </h1>

  <p>
    {{$ctrl.post.text}}
  </p>

  <ul>
    <li ng-repeat="comment in $ctrl.post.comments">
      {{comment.userName}} says: "{{comment.text}}"

      <a ng-click="$ctrl.deleteComment($index)">
        Delete Comment
      </a>

      <ul ng-if="comment.replies.length > 0">
        <li ng-repeat="reply in comment.replies">
          {{reply.userName}} replies: "{{reply.text}}"

          <a ng-click="$ctrl.deleteReply($parent.$index, $index)">
            Delete Reply
          </a>
        </li>
      </ul>

      <input ng-model="$ctrl.newReply.text"/>

      <a ng-click="$ctrl.submitNewReply($index)">
        Add reply
      </a>
    </li>
  </ul>

  <input ng-model="$ctrl.newComment.text"/>

  <a ng-click="$ctrl.submitNewComment()">
    Add comment
  </a>
</div>
```

This implementation has nothing special, if you are a bit familiar with AngularJS, you'll understand it without problems. I used the controller as namespace so it's clear that the example has nothing to see with the infamous scope soup (which is not the topic of this article).

Analysis of the naif implementation: hidden architectural defects of binding directly a nested JSON to the DOM

As in the first example we analysed, I tried to do my worst mixing up together Back-End, Front-End and DOM models.

Let's go straight to the point and look at the most evident symptom that something is wrong:

The purpose of this line is deleting a reply. But it looks a bit too long, doesn't it? First you have to get back to the "*post*" object, extract the comments, then get the one that corresponds to the current "*commentId*", extract the replies, get the correct one and remove it from the array.

But all we needed to do to was removing a simple node from the DOM. So why going up until the parent of the parent of our model?

You might object that clicking on a "delete" button on a reply is a meaningful event that goes beyond the act of removing a node in the DOM: because the user doesn't just want to have fun with the DOM, his goal is actually to remove the reply from the database.

You are wrong, there is no hidden meaning behind the user's action, at least not at the level of abstraction of the DOM model. ***The only responsibility of the DOM model is to remove that node: notifying the API of the user's desire to delete the reply is somebody else's task.***

This is exactly what I mean when I talk about keeping the abstractions separate: *the most upfront of the three models is nothing else than an extension of the DOM, and it should only focus on that.* We'll think later about how to notify the API.

To what unexpected consequences that doubly nested call leads us? **Answer is, lack of scalability**: the example we are analysing is pretty simple, *but in Front-End world things change fast, and that apparently harmless line of code can easily turn into a nightmare.* For instance, what if we wanted to delete comments and replies asynchronously, keeping them in the DOM until the API has responded? Or, what if we needed to add several nested replies until a certain maximum depth? How long and complex would that line become?

Let's now step back from the DOM and focus on the other side of this implementation: the API call that tells the Back-End to update the post.

At first sight, this might seem the simplest choice to communicate with the API: we have this big object that we keep updated as the user interacts with the app, let's just post the whole of it to the API.

But things are not as simple as they look: ***the strategy of posting the whole JSON works only because our implementation assumes that such complete object is always available.***

I'll have to repeat myself, **in Front-End client's requirements change fast**: a simple static post with nested comments and replies can turn in no time into something highly dynamic, where you need to save single comments separately, or modify just the post's text without touching comments and replies. Or who knows what else.

Here is already an early symptom that something is not right:

When we want to create a new comment, the DOM needs to bind an “*input*” HTML tag to a variable in the controller: ideally, the natural choice to integrate such a variable to our existing architecture would be a new empty “*comment*” appended to the list of comments in our JSON.

But this would imply that, if the user leaves a comment incomplete, and then performs some other action that submits the whole post to the API, his draft comment would be sent along with everything else, and saved.

So, in order to make it work, we had already to start complicating things a bit, and created a separate “*newComment*” object that we attach to the “*post*” only when the comment is submitted: again, this is not a solid architecture, and it can easily become too complicated as the application grows.

And by the way, if we look at the implementation of the empty comment generator, we find another potential issue:

The “*userId*” property is repeated in every single new comment or reply the user creates. This is redundant because the ID of the commenting user is always the same, we don’t need to carry it around and copy it everywhere: it should be only stored in one place for when we need to post something to the API.

A brief note on APIs: I am aware that sometimes they are not designed to be optimised for Front-End, and I am aware that often they are not modifiable at your desire. ***So it is possible that you are forced to post a whole nested object to the API, but this doesn’t mean that you have to bind such an object directly to the DOM.***

The best solution is wrapping your API inside an adaptor that takes care of keeping the objects updated, I’ll go into details in the end of the article.

A “Copernican Revolution”: the component pattern approach

If I had to synthesise in a few words why the previous implementation was not right, I would say: **we developed the Front-End around an object that doesn’t represent it.**

A “*post*” JSON with nested “*comments*” and “*replies*” is a very good logical abstraction of how posts, comments and replies are actually represented in our minds: and this is why such a data structure was chosen to represent them in the Back-End.

But in Front-End we need to focus on something different, we need to let the user write inside inputs, add texts, and do any sort of interactive things: hence, models we use in the Front-End must be a good abstraction of ***these interactions***. And this is even more true in Angular, where the DOM is bound to its own model and gets updated automatically.

The fact that, in Angular, the DOM cannot be modified manually, doesn’t imply that you should bind it to a Back-End model and expect it to update magically: the idea of extending the DOM, by binding it to JavaScript objects, was meant to provide a more powerful and solid way to manipulate it, NOT to attach to Back-End models some kind of scaffolded automatic HTML and bypass the Front-End.

So, let’s try and approach our problem in a completely different way: *we won’t start from Back-End’s models to build the Front-End around them, but we’ll rather analyse the view, study in which ways it needs to interact with the user, and abstract it* (this is why I call it a “Copernican revolution”, because we look at the same problem from the opposite point of view).

1. First, we’ll decompose the view in **components** that interact with the user: we’ll bind them to the DOM and they will form our **DOM model**.
2. We’ll then decide which **further abstraction** we need to coordinate different parts of our app: this will be the **Front-End model**; *notice that we are building Front-End models without yet knowing if the same models are in the backend, or how they are implemented there.*
3. Finally, once the whole Front-End’s architecture has been designed, **we’ll look at the API** and decide how to send data to the Back-End, and how to adapt to our architecture the data that is coming from the Back-End.

The first step is called ***component pattern***, and it’s the basic concept behind the architecture of Angular2+.

So, let’s look at the requirements and decide which components we need. Let’s start from the simplest thing, the array of replies relative to a comment: this can be a component, and all it needs to do is letting the user **add** and **delete** a reply.

Looking at HTML produced by the previous implementation, such a component would correspond to this bit:

This component itself is very simple and doesn’t need to know what is above it, but we still need to send calls to the API when the user adds or deletes a reply: **so we’ll make**

our component dispatch an event to whatever its parent is. The event will either say "added a reply with this text", or "deleted reply at this index".

The second component can be the list of comments, which corresponds to this HTML:

This component looks the same as the first one, with the exception that it additionally listens to the child component "replies", and scopes the original message with the index of the comment to which the child component is attached.

Finally, the last component should be the one representing the "*post*":

Implementation of the component pattern approach (using Angular2+)

For this implementation I am going to use Angular2+, because the framework makes it easier to show the benefits of component pattern; this doesn't mean that AngularJS is not good to implement component pattern, if you are interested I suggest you to read this very interesting and detailed article.

In the implementation that follows I will limit myself only to the DOM model (components in TypeScript and corresponding HTML templates); so, I will temporarily omit:

- the mechanism that retrieves data from the API and pushes it back as the user interacts with the DOM (interaction with Back-End models);
- the definition of Front-End models that are behind the DOM model (omitting them means that I will set the corresponding types to "*any*").

In the paragraphs after this, I'll extend the example, explaining in order (1) how to make the components communicate with the API, and (2) how to create types representing Front-End models we need.

So, let's get started, let's begin with the implementation of the "*post*" component. This is the TypeScript part:

And this is the (very simple) HTML template:

There is nothing special in this part of the implementation: the template binds to the dom the three main properties of a post, and calls the component “*comments*”.

The simplicity of this component is also its strenght. *The only variables present in the TypeScript class are the ones that we bind to the DOM: this guideline should be respected in general, because* ***we are creating a DOM model, something which main purpose is to extend the DOM.***

You’ll notice that we are not passing anything to the “*comments*” component: don’t worry about that now, I’ll come back to it later when I’ll describe the interaction with API. For the moment my only goal is to focus on user interactions and DOM model.

Let’s move forward to the second component, that represents comments:

Template:

This is the key component in the implementation, because it’s the one that actively posts data to the API.

We can simply list everything the component does by reading each of the functions bound to it. Let's do it first with the two functions that correspond to DOM interactions (the other two functions are used to react to events happening at level of "*replies*", the child component):

- When the user hits "*delete*" on a comment, the component removes the corresponding index from the comments array, plus it posts a notification to the API (not implemented here)
- When the user hits "*save*" on the new comment input, the component adds a new comment to the comments array, **adding to it the user's name on the fly**; it empties the input so that the user can comment again; and separately, it posts the new comment to the API, along with the user's ID (not implemented here)

The key point of these two actions is **separation of concerns**:

- There is an object that represents the DOM (the array of comments), and this object ***contains exactly what we need to render those comments, nothing else.***
- Separately, we have an external API (not implemented here, we'll do it later) ***to which you pass only the essential data the API needs to know.***

Let's analyse in detail the minimal data that we need for each layer of abstraction:

1. The DOM model is an array of comments, each consisting of a **text** and a **user name**
2. The call that notifies the API of deletion of a comment consists only of the **index** of the comment to delete
3. The call that notifies the API of creation of a comment consists only of **text** and **user ID**

Let's now look at the other two functions, the ones that deal with replies. We are using a very nice feature of Angular2+, that allows us to treat as clear output events sent up by a subcomponent:

The meaning of this line is that child component "*replies*" is expected to send up two possible events:

- An event called "*deleteEvent*", which argument is the **index** of the **reply to be deleted**
- An event called "*submitEvent*", which argument is the **text** of the **reply that has been created**

The template expression that is build on top of these events **adds the comment's index to the original parameters** that are sent from the child component, and passes everything to the corresponding method of "*comments*" component, **which only responsibility is to notify the API about the change**.

Why do we send events up from “*replies*” to “*comments*”, instead of firing them directly in “*replies*”? ***Because the child component doesn’t have enough information to notify the API: it’s flat, it doesn’t know to which comment it has been attached. We did this on purpose in order to simplify the interaction with the DOM.***

It’s time to finally look at the last component, the one representing replies:

And the (simple, again) template:

This component is **parent-agnostic**: it simply implements two interactions with the user, and sends up a signal when the user deletes or adds a reply; along with the signal, the only parameter sent is relative to the specific list of replies, ***it doesn't include any information about what these replies are attached to***.

As result, the implementation of the DOM model is **flat** and easy: we only store the **exact variables we need** to render the DOM, and when the user wants do change the DOM we are able to modify them straightaway.

I want to highlight two technicalities of Angular2+: first, the **output declarations**, that are the component counterpart of the events we bound in the template of "*comments*":

It’s so nice and clear to be forced to declare, for each component, a list of inputs and outputs: it makes you keep the architecture clean, especially when used together with dependency injection.

Second, look at the usage of the template variable “*#newReply*”:

I decided to avoid two way binding in this example: two way binding is something I always try to avoid because the data has to flow, at every key stroke, from the user to the DOM model and then back to the DOM, it's a great waste of energy. Sometimes two way binding is necessary, *but in this specific case, where I only need to read the input's value and empty it as the user submits, a template variable lets me do it safely*.

So, this is our new DOM model; summarising, in comparison to the first naif implementation, we got the following benefits:

- In each component our model contains **only the data it effectively needs to render** its portion of the DOM
- Such **data is immediately available**, we don't need to go up a list of nested JSONS: this because **we kept the DOM model separated** from whatever structure we

needed to send data to the API

- We are **not forced to keep artificial objects** to store empty comments or replies, every information is naturally retrieved by the DOM and sent to the API
- There is **no redundancy of user data that is not needed in the DOM**; actually the only place where we extract the user's ID is the component "*comments*", because it's the only one that sends data to the API
- All the three components benefit from **isolation** and **encapsulation**, which makes them **more likely to become reusable**, especially "*replies*": this is a good achievement since reusability is more difficult in Front-End.

Data flow from API to the DOM: general thoughts

In order to complete the implementation of our example, we need to take care of the API (interaction with Back-End models), which will turn our small implementation into complete and self-sufficient; after that, we'll talk about designing Front-End models, which are not strictly necessary to have a standalone working example, but are needed to make the components interact with the rest of the Front-End.

Regarding interaction with the API, this is itself a task that we should decompose in two subtasks:

1. How to make data flow from the API to the DOM
2. How to send information to the API as consequence of interaction of the DOM with the user

Data flow from API to DOM is the topic of this and the following few paragraphs; I want to talk about this in detail because I find it a controversial topic that causes a lot of **misunderstanding**: *in my opinion such data flow is often the reason that makes people erroneously develop Angular applications around Back-End models.*

We already saw how we handled such data flow in the first naif AngularJS implementation: the API returns us a nested JSON, and we simply use it to construct the

DOM. If we forget about all the other defects that implementation had, and focus only on the way we bound to the DOM data coming from the API, we can generalise as follows the two concepts behind it:

> We bound the DOM directly to properties of the "post" object, only because the API returned us a "post" object and it was apparently easy to just use it straightaway

And then, regarding the nested structure:

> We decided to bind replies to the parent comment, and comments to the parent post, only because the API returned us the data in shape of a nested JSON

In other words, we took decisions on Front-End's architecture, based on an arbitrary data structure we received from the Back-End.

Regardless of how well we designed the DOM model, and how sophisticated and well designed is the specific framework we are using, such architectural decisions will still heavily affect the quality of our app, so it's important to always know what we are doing.

Data flow from API to the DOM: naif implementation in Angular2+

I'll now describe a "hybrid" implementation: I'll add, on top of the good Angular2+ DOM model we developed two paragraphs ago, a way of receiving data from the API that is an Angular2+ analogue of the first naif implementation we did in AngularJS.

My goal is to show that a good DOM model is not enough to guarantee that our app is well designed and error-prone: regardless of the DOM model, if we don't use correctly

the data we receive from the API, our app will be weak and subject to unpredictable bugs.

In order not to make my code snippets become too large, In the following implementation I'll only focus on how to use data we fetch from the API: I'll inline HTML inside the component's decorator, and in some cases I won't repeat some functions and nodes that we already described two paragraphs ago.

This is the implementation of "*post*":

This is the implementation of “*comments*”

And finally “*replies*”:

The two key features introduced by this implementation are:

1. Instead of binding the DOM to properties of the component “*PostComponent*”, we bound them to properties of the object “*post*”; (we’ll see that this is not a problem by itself*, the problem rather lies in the fact that we are using the same object we received from the API*)

2. Data is not passed anymore to “*comments*” and “*replies*” by fetching it directly from the API, *but it’s rather passed from parent to child component, following the nested structure of the JSON that the API has returned* (in the implementation, “*post*” is the only component responsible to fetch data from the API, and the only one that implements “*ngOnInit*”)

Consequence of these two decisions is that, for the specific purpose of our example, **in spite of Angular2+’s syntax and in spite of the DOM model we designed, the resulting architecture is not encapsulated nor solid**.

Data flow from API to the DOM: unexpected, unpredictable bugs of the naif implementation

In order to know what we are doing and analyse the new implementation, *we need to have at least a general idea about how change detection algorithm works*.

I am not going to go much into details: if you want a deeper read, I suggest you this great article about change detection in general, and this other one about DOM interpolation.

Change detection uses a **recursive algorithm**:

- The algorithm starts at the **root component**: the first thing it does is recalculating and updating the "*@Input()*" bindings on the component
- It then extracts the list of children components, and updates "*@Input()*" bindings on each of them
- The algorithm keeps doing the same thing recursively, **until it finds a leaf component** (a component without children components): on leaf components it passes to the next step, which is **updating the DOM** corresponding to their templates
- It then comes back to the parent component and updates the DOM corresponding to its template
- It repeats the same operation recursively, until it comes back to the root component where everything started: **now the whole DOM is updated**

Even if this was a very simplified description of change detection algorithm, it has something more than most descriptions you'll find online, **because it focuses on the recursive nature of the algorithm**, which is often ignored.

In fact, **recursiveness is the core** of the change detection process, specifically the ability of deciding **how to update a component**, only based on its status and the status of the parent component.

When the algorithm is looking at a child component from a parent component, it usually faces three possible scenarios:

1. Child component **only depends on the declared inputs**, and those inputs are not going to change without their reference changing as well: ***this is the most favourable scenario, in which you can use “onPush” strategy and detach from change detection the whole child component’s subtree***

2. Child component only depends on the declared inputs, but those inputs are **objects which attribues can change** while still keeping the same object reference: ***in this case it’s not possible to skip change detection for the subtree, because Angular2+ might not realise that something in that subtree has changed*** (for performance reasons, Angular2+ only updates bindings when their reference changes)

3. Child component **doesn’t depend only on its inputs**, because it additionally fetches data from the API or other external services: ***in this case as well optimisation is not possible, and change detection has to run for the whole subtree***

Now we are ready to look at the specific case of our latest implementation; this is the template of “*PostComponent*”:

When change detection finds this template, can you yell in which of the three described scenarios we are?

It can’t be any of the first two, because the component evidently doesn’t depend on its declared inputs: in fact, the component doesn’t have any input, because its whole content is determined by a call to the API.

But the problem is that, surprisingly, we are not in the third scenario either, **because the status of the template doesn’t depend only on the API**.

We are actually in a fourth scenario, that didn’t seem to be possible when at first we looked at the change detection algorithm: ***the object “post”, that is used to populate the template, is not modifiable only by the API, but it can potentially be modified by any other service or component who fetched the same JSON from the API service.***

In the code snippet above, “*ApiService*” caches the returned JSON in an instance attribute, which is good practice to avoid unnecessary external calls. ***But, since we bound directly to the DOM the object we received from the API service, our component is now exposed to trivial errors***, like the one in “*AddMisterToUserNameService*”.

Components, that bind directly to the DOM objects received from the API, are not isolated, no matter how nice and well designed is your DOM model.

How can we prevent this problem and make our components isolated? Let’s step back for one moment, and ask ourselves another question: do we actually want our components to be isolated?

There are actually situations in which it’s good to propagate automatically everywhere in the DOM a change that occurred in a model; *but, if we are in such a situation, we should decide intentionally to propagate those changes: we shouldn’t do it as a side effect of binding to the DOM an object returned by the API adaptor, just because it’s easy.*

In our specific case, especially because we are looking at a standalone example without knowing the context of the rest of the application, we definitely don’t want to be able to update the object “*post*” globally: so let’s look at possible solutions to achieve encapsulation.

If we want our component to be isolated, we should **either copy in the DOM model every attribute we need**, or **make the API adaptor copy the whole object** before retrieving it; if we choose the second option, the best way to achieve it is by wrapping the API data into a **Front-End model**.

If I didn’t talk yet about Front-End models, it’s only because ***I really wanted to put the stress on the correct architecture design flow****, that should always start from the DOM model, and, subsequently, abstract Front-End models; in fact, you start needing Front-End* *models only when you connect the different parts of your application, and standalone examples like the one we are studying can perfectly live without any abstract model behind them. Anyway, I'll dedicate some words to Front-End models before the end of this article.*

Look at the constructor of the Front-End model class "*Post*":

- It **copies every relevant property** of the JSON into a local attribute, so every time we instantiate a new "*Post*" object we get a different copy of those properties
- It **filters out all unnecessary properties** that might come from the API (we are ignoring redundant data)
- I left the JSON type as "*any*" because that JSON is something that comes from the API and not under our control

Let's now look at the other potential risk of our naif implementation of data flow from API: *the fact that the array of replies is bound to the parent "CommentsComponent" and the array of comments is bound to the parent "PostComponent".*

Obviously we have the same non-isolation problem we remarked for the “*Post*” object, because those arrays both came directly from the API adaptor service; but in this case there is also an additional risk, because the arrays can be accidentally modified from parent components themselves:

In the code snippet above, we forgot that the “*post*” object has already a property called “*comments*” (because such component has not explicitly declared anywhere, since we injected it directly from the API), and we accidentally use it to store other things. The side effect of this error is that we'll accidentally wipe away all the HTML related to comments and replies (and possibly even break the app).

Probably this specific error is not so realistic in a real app, but it serves for my purpose: *I wanted to stress how our nice DOM model is unexpectedly weakened by the way we import data from the API.*

With the architecture we designed, we made "*PostComponent*" responsible for DOM rendering of all comments, replies, and any other child component as well: ***as in the previous example, we have to ask ourselves if we really wanted to achieve that, or if we did it accidentally because of the data structure that came from the API***.

While there are definitely a lot of cases in which we want a subtree of components to be completely modifiable from the root component (in an admin interface for instance, where we want the user to be able to sort items at any nested level), **we definitely don't want it here**: a user shouldn't be able to modify comments and replies by performing actions on the "*post*" component. All subcomponents we designed should rather be isolated.

I think that there is a deeper problem behind architectural errors like this:

> People often make confusion between the processes of FIRST RENDERING and UPDATING the DOM.

The fact that, in order to render the DOM the first time, we need to connect comments to posts and replies to comments, doesn't imply that those connections are actually necessary to make the DOM interact with the user.

The solution I want to propose for this problem deserves its own paragraph, so I'm wrapping it up here.

To conclude the paragraph, I'd like to summarise again in a few words the following important concept: we said since the beginning that, when designing the DOM model, we shouldn't be influenced by the structure of Back-End models; ***in the same way, when***

we design the data flow from Back-End to the DOM, we shouldn't be influenced by the data structure that comes from the API.

Data flow from API to the DOM: a pattern that enforces isolation

In order to maintain isolation, we saw that we need to do the following things:

1. Don't use in the DOM model (Angular2+ components) the same JSON that has been returned by the API
2. Trim what comes from the API so that it only includes data we need
3. Somehow use the API **only to render the DOM for the first time**, and after the first rendering keep components isolated from each other

So far we implemented (1) and (2) in the same service that works as API adaptor; but it would be actually good to leave the API adaptor alone, and rather put this logic inside a separate service, that in addition takes care of point (3).

This service acts as **mediator** between the API adaptor and our DOM model: when you call the method “*storePost*”, the service:

- **Calls the API adaptor** once
- Uses the response to **initialise Front-Model objects** of type “*post*”, “*comments*” and “*replies*” (we delegate to the classes of those objects the responsibility to remove unnecessary data and make sure there is no reference to the original JSON)
- **Stores these objects internally**, providing getter methods to retrieve them

Notice that the API adaptor is not responsible anymore for caching the response, because the caching is being done in “*PostApiService*”: it’s better to leave the API adaptor in charge of only one thing, i.e. communicate with the Back-End.

How to use this service in our components?

The idea I have in mind is that the root “*PostComponent*” should somehow initialise this service and then **make it available exclusively to all components in its subtree**: this way there is no risk of getting confused if some other “*PostComponent*” gets rendered in the same page.

And here comes handy a cool feature of Angular2+’s dependency injection: ***creating a separate instance of a provider by injecting it at level of component***.

Voila! This is the final optimised version of “*PostComponent*”.

- Provider “*PostApiService*” is injected at component level, so this component and all its subcomponents will get a separate copy of it, there is no risk of clashing with other instances of “*PostComponent*”
- The post’s ID is passed as input to the component, which then uses it to initialise “*PostApiService*”; this happens during “*ngOnInit*” lifecycle hook, so that the instance of “*PostApiService*” will be available in all “*ngOnInit*” hooks of the subcomponents (because they are all called after the father’s “*ngOnInit*”)
- We are safely binding to the DOM the instance of “*Post*”, because we know that the model’s interface contains the variables we need to expose in the DOM; and also

because we are sure that this instance is just for us, it’s not going to be spread around the app and it doesn’t risk to be accidentally modified

- We don’t pass any “*@Input()*” binding to component “*comments*”, because that component already knows how to fetch its data through the local service instance we provided

By the way remember that, when a component depends on an API call, you should resolve it in the router, otherwise the DOM will flicker when the API receives a response.

I’ll conclude the paragraph with the implementation of the remaining two components:

Notice how “*CommentsComponent*” delegates “*PostApiService*” to send data to the API: “*CommentsComponents*” doesn’t know the ID of the post to which comments are attached, but since it uses that specific instance of “*PostApiService*” we can sure that every call will be scoped below the root post.

Notice also that this time we are using types: wrapping data that comes from the API in a Front-End model is a safe way of preventing accidental modifications from other parts of the app.

Finally, notice how we are passing to “*RepliesComponent*” the index of the parent comment, and how the child component uses it to fetch the appropriate array of replies from “*PostApiService*”.

Generalisation of this pattern: separation of models and responsibilities

The keyword behind the implementation above is **separation**:

- **Separation of DOM models** from the logic of Back-End models, that helps us designing DOM functionality smoothly and without noise
- **Separation of first rendering** from subsequent interactions with the user, because we don’t bind replies to comments and comments to post: during first rendering, we rather use an external service as an alternative way of passing data to children components
- **Separation of responsibilities**, because not all components are responsible to interact with the API

The component we chose to **fetch data from the API** is the root “*PostComponent*”, because this is the only one that knows the post’s ID. The component doesn’t fetch data directly, but it does it by initialising “*PostApiService*”, who in turn calls the API and stores safely all the objects that will be needed by that component and by its subcomponents.

The component we chose to **send data to the API** is rather “*CommentsComponent*”, because this is the one that holds all the knowledge of what the user is doing (remember that the child “*RepliesComponent*” is parent agnostic).

Both these choices were intentional, because doing things in a different way would have added unnecessary complexity to the architecture:

If “*CommentsComponents*” had been responsible to initialise “*PostApiComponent*”, then we would have had to pass to it the post’s ID (redundant information); moreover, if in the future we had to add more children components to “*PostComponent*” (siblings of “*CommentsComponent*”), how would we fetch the data to populate them?

Then, if “*PostComponents*” had been responsible to send data to the API, we would have had to forward up four different events from “*CommentComponents*”:

Which would have really added unnecessary complexity to the system.

A last word about the role of “*PostApiService*”: *this service might be a solution to the eternal fight between Front-End developer and unnecessary API calls.*

The ideal way to fill up separate components with data is, from a Front-End architectural perspective, to have each component fetch its own data directly from the API; but, if we do this, we stuff the user's connection with too many external calls, and in conditions of low network the app becomes too slow.

But with a scoped API adaptor like the one we designed, we are able to

- Look at the API and identify the largest area of the DOM that can be fed by a single call
- Wrap that area in an empty component which only responsibility is to initialise the scoped API adaptor
- The API adaptor calls the API once and stores all necessary objects
- Finally, each component in the subtree can call the API adaptor exactly as if it was the real API, without unnecessary calls to the Back-End

How to send data to "unfriendly" APIs

We talked a lot about the flow of data from API to DOM model, let's spend a few words about the flow in the other direction.

The main problem, when submitting data to the Back-End, is when **the API hasn't been optimised for Front-End**. For instance, the optimal API we desire for the implementation of post plus comments plus replies would look like this:

Unfortunately, sometimes APIs are not well designed like the one above; and often APIs are not under our control and cannot be modified easily (because, for instance, the Back-End might be outsourced to an external company). A terrible API would be the following:

This API is really difficult to interact with, because it only accepts the full JSON of post, comments and replies. A Front-End developer might be tempted to make components construct the large JSON themselves, and keep it updated as the user interacts: but we already saw that this is not a good choice, and ***we don't want to modify our Front-End architecture just because the API hasn't been designed properly***.

The right solution is using "*PostApiAdaptor*": at the moment of fetching data from the API, the service did receive the JSON that represents the post, so it will be enough to store it and keep it updated every time components send actions.

This way, components interact with the service using the **API that is optimal for them** ("*deleteComment*", "*postComment*", etc.): we can now say that our components have become **API agnostic**, because we can replace "*PostApiService*" with any other API adaptor that makes available those methods.

Example of Front-End models holding data that doesn't correspond to anything in a Back-End model

Front-End models are an abstraction of entities in the Front-End: we mostly need them to keep such entities consistent throughout our app.

You might have noticed that, in the example of post + comments + replies that we followed for most of the article, we almost never mentioned Front-End models: we introduced them only when we had to find a solution to the risk of binding the DOM to an object that is shared throughout the application.

The reason is that, in my opinion, when designing the architecture, we should focus on DOM models as first thing: ***Front-End models come only AFTER the DOM model has already been designed***, because these models are a further abstraction, which purpose is to make separate components interact with each other and keep the system consistent. So, in a single standalone example, Front-End models are not essential.

As I keep repeating since the beginning of this article, Front-End models are often similar to Back-End models but they aren't likely to be exactly the same. I have already remarked how common it is for a Back-End model to contain data that is not needed in the Front-End, ***now I want to show an example of the opposite: a Front-End model that holds data with no correspondence in the Back-End***.

Let's imagine that, somewhere else in our application, there is a status bar that shows statistics about the current user, including a badge that changes color depending on how many comments / replies the user posted in the last 24 hours.

In order to keep this status bar consistent, we need some sort of shared service to hold the user's data:

This Front-End model “*User*” contains an attribute that cannot be saved in the Back-End, ***because it abstracts a purely visual property that is present only in the Front-End***.

Conclusion

The DOM model should contain only attributes that are used directly in the DOM, and functions that are directly called from the DOM; any longer or more complicated function should be refactored and extracted into a service.

> First, design user interactions and abstract them in a solid, good DOM model (before even looking at the data structure in the Back-End)

Front-End models should be an abstraction of entities as seen in the Front-End: they should contain, besides regular attributes that are in the Back-End, other properties that need to be shared over different components in the Front-End.

> Second, decide which entities you have to abstract in Front-End models, and how: you can already have a fully working app at this stage, you just need to create services that simulate interactions with the API

Back-End models and relative API structure shouldn’t influence your choices while you design the Front-End’s architecture.

> Finally, connect your Front-End with the Back-End: ideally, try to simplify and optimise the API for your Front-End, and if it’s not possible, to take care of this use the API adaptors you have created

Angular 4 Angularjs Mvc Angular JavaScript