**Do database modelling and Create Various APIs**

**TEAM** **MEMBERS**:

ASIF FAROOK.K (211520205016) (**GITHUB** **ID**: <https://github.com/ASIF10504>)

ARSHAT AHAMED RAFIQ A.S (211520205015)(**GITHUB** **ID**: <https://github.com/AGTAHAMED>)

Reason 1: Tight coupling between API and database model.

A database model is specific to how you store your data in the underlying database. Web APIs are how clients or other consumers should experience your application. You don’t want change in your data model to impact your Web API clients. They should be hidden from the underlying implementation changes that happen in your data model. Clients need to work with stable Web APIs so that they don’t have to change every time you change your data model. With the data model exposed all clients are now coupled with the underlying data model. A good web API encapsulates the data model as it is considered an implementation detail. When you start having a separate data model and Web API model might feel redundant work but over time it shows its value and makes two layers evolve independently.

> Although not directly linked to this point, I prefer to create an Anti-corruption layer when I consume external APIs. Anti-corruption layer. prevents a downstream system/service domain model from polluting the domain model of a new service. Anti-corruption layer is a concept from Domain-driven design. I covered this point in detail in a [post](https://shekhargulati.com/2021/09/09/why-is-it-ok-to-write-pass-through-services-in-a-microservices-architecture/) I wrote about Pass-through services.

Reason 2: You don’t want to or you should not expose all the database fields.

If you expose your database model as Web API then you risk sharing sensitive fields like passwords, tokens in the API response. Even if you don’t do that there are many fields which you don’t need to share with the client. These are fields like updated timestamp, who updated, version fields, etc. Apart from the security issue related with sharing sensitive data there is also performance cost associated with sending extra data.

Another point is when you use an ORM framework like Hibernate(in Java) then with bidirectional OneToMany or ManyToMany relationships you will get exceptions when you serialize the object graph to JSON. This is because of cyclic nature. You will have to use framework specific hacks like in Jackson you will have to use JsonBackReference to avoid these. Now, your entities start getting polluted with your web API concerns.

The best way to not get into this mess is to have separate models for web APIs and the database.

Reason 3: The need for aggregated APIs.

Most of the time when designing web APIs you just don’t want details about a specific entity you might also want related entities as well. For example, when you make a GET call to get details about a specific Github repository you get the following response. Refer to Github REST API [documentation](https://docs.github.com/en/rest/reference/repos). I have removed fields and nested objects for brevity.

{

"id": 1296269,

"name": "Hello-World",

"full\_name": "octocat/Hello-World",

"owner": {

"login": "octocat",

"id": 1,

"url": "<https://api.github.com/users/octocat>",

},

"private": false,

"html\_url": "<https://github.com/octocat/Hello-World>",

"description": "This your first repo!",

"fork": false,

"url": "<https://api.github.com/repos/octocat/Hello-World>",

"forks\_count": 9,

"forks": 9,

"stargazers\_count": 80,

"watchers\_count": 80,

"watchers": 80,

"size": 108,

"default\_branch": "master",

"open\_issues\_count": 0,

"open\_issues": 0,

"is\_template": false,

"topics": [

"octocat",

"atom",

"electron",

"api"

],

"template\_repository": {

"id": 1296269,

"name": "Hello-World-Template",

"full\_name": "octocat/Hello-World-Template",

"owner": {

"login": "octocat",

"id": 1

},

"private": false,

"html\_url": "<https://github.com/octocat/Hello-World-Template>",

"description": "This your first repo!"

},

"license": {

"key": "mit",

"name": "MIT License",

"spdx\_id": "MIT",

"url": "<https://api.github.com/licenses/mit>",

"node\_id": "MDc6TGljZW5zZW1pdA=="

},

"organization": {

"login": "octocat",

"id": 1,

"avatar\_url": "<https://github.com/images/error/octocat_happy.gif>",

"gravatar\_id": "",

"url": "<https://api.github.com/users/octocat>",

"html\_url": "<https://github.com/octocat>",

},

"license": {},

"forks": 1,

"open\_issues": 1,

"watchers": 1

}

As you can see you are not only getting Github repository details, you are also getting owner, template repository, organization details as well.

These sort of aggregated APIs reduce network chattiness for the clients and make clients much more simpler. If clients don’t get this data in a single call then they will have to make multiple API calls to fetch this data.

The Web API contracts are also governed by UX needs so rather than modelling your API in an entity oriented manner you will have to consider what experience is required and model accordingly.

Reason 4: Derived fields.

There are times you want to expose a derived field in a web API instead of exposing field(s) in your database model. Let’s assume you are modelling a web API for an application that accepts assignments from users. An assignment has two datetime fields – assignment\_shared\_on and assignment\_received\_on. Rather than returning the dates to the client you want to return the days a candidate took to submit the assignment. This can be added to your web API model rather than adding it to your data model.

Reason 5: Choose a different and suitable data type.

This is linked to the reason 4 we just discussed. Having a different web API model gives you the flexibility to choose a different and suitable data type for your clients. So in the example discussed in point 4, we returned int data type rather than datetime type.

Conclusion

We can do a much better job at Web API design by following contract first model of API design. API Styles like GraphQL enforce contract first API design so they can help avoid this anti-pattern. I know with GraphQL there are tools that allow you to generate GraphQL APIs from the database schema. I prefer to avoid such tools. But, if you have to use such tools then I recommend writing a transformation layer in front of them. This will protect you from leaking your database model to the consumers.

*With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody.*