Don't expose your JPA entities in your REST API

Should you expose your entities in your REST API, or should you prefer a DTO projection?

That's one of the most commonly asked questions when I'm talking to developers or when I'm coaching teams who are working on a new application.

There are two main reasons for these questions and all the discussions that arise from them:

1. Entities are POJOs. It often seems like they can get easily serialized and deserialized to JSON documents. If it really works that easily, the implementation of your REST endpoints would become pretty simple.
2. Exposing your entities creates a strong coupling between your API and your persistence model. Any difference between the 2 models introduces extra complexity, and you need to find a way to bridge the gap between them. Unfortunately, there are always differences between your API and your persistence model. The most obvious ones are the handling of associations between your entities.

There is an obvious conflict. It seems like exposing entities makes implementing your use cases easier, but it also introduces new problems. So, what has a bigger impact on your implementation? And are there any other problems that might not be that obvious?

I have seen both approaches in several projects, and over the years, I've formed a pretty strong opinion on this. Even though it's tempting to expose your entities, you should avoid it. Exposing your entities at your API makes it impossible to fulfill a few best practices when designing your API; it reduces the readability of your entity classes, slows down your application, and makes it hard to implement a true REST architecture.

You can avoid all of these issues by designing DTO classes, which you then serialize and deserialize on your API. That requires you to implement a mapping between the DTOs and your internal data structures. But that's worth it if you consider all the downsides of exposing entities in your API.

Reason 1: Hide implementation details

As a general best practice, your API shouldn't expose any implementation details of your application. The structure that you use to persist your data is such a detail.

If you do it anyways, every change of your REST API will affect your entity model and vice versa. That means your API and your persistence layer can no longer evolve independently of each other.

Reason 2: Don't bloat your entities with additional annotations

Most entity mappings already require several annotations. Adding additional ones for your JSON mapping makes the entity classes even harder to understand. Better keep it simple and separate the entity class from the class you use to serialize and deserialize your JSON documents.

Reason 3: Different handling of associations

With JPA and Hibernate, you typically use managed associations that are represented by an entity attribute. Depending on the configured fetch type and your query, this association either references other entity objects or dynamically created proxy objects or a Hibernate-specific List or Set implementation.

In your REST API, you handle these associations differently. The correct way would be to provide a link for each association. But most teams decide to either not model the associations at all or to only include id references.

Links and id references provide a similar challenge: You need to handle them during serialization and deserialization. That requires additional queries that slow down your application.

Excluding associations creates problems when you merge a deserialized entity object. The association attribute is set to null and Hibernate removes all associations from the database.

Reason 4: Design your APIs

If you're not implementing a very simple CRUD operation, your clients will most likely benefit from carefully designed responses which use different representations of the same graph of entities. Creating these different representations based on use case specific DTO classes is pretty simple. But doing the same based on a graph of entity objects is much harder and most likely requires some manual mappings.

Reason 5: Support multiple versions of your API

Supporting multiple versions of an API is much easier if you're exposing DTOs. That separates the persistence layer from your API, and you can introduce a migration layer to your application. This layer separates all the operations required to map the calls from your old API to the new one. That allows you to provide a simple and efficient implementation of your current API. And whenever you deactivate the old API, you can remove the migration layer.