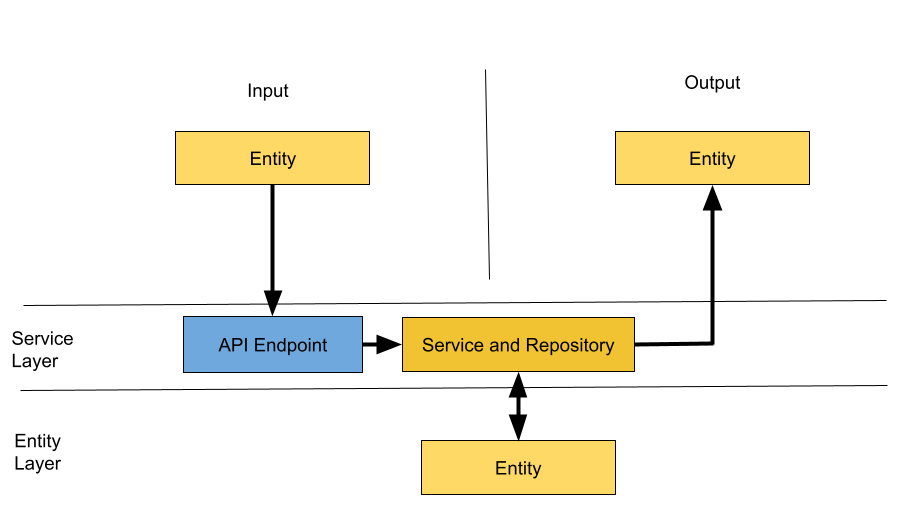
Separating Request, Entity and Response

<https://drive.google.com/drive/folders/1tsu2S_EnDds8oIRiiip4rwl2qhXEq-Se?usp=sharing>

Security is a major concern for web applications. One of the ways to achieve this is by using the least access principle. This means only exposing things that are necessary and hiding the rest.

In a typical web application, we perform CRUD operations on entities.

It’s quite common to have the same entity object be requested, saving to DB and response.

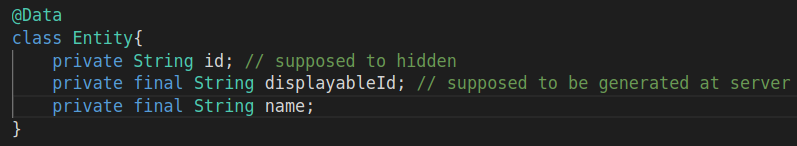


Here are the roles of classes mentioned above,

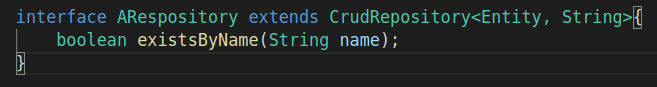
* API endpoints: deals with input objects. In this case, a single one entity.
* Service and Repository: deal with manipulating and saving entities to the database. And sending the same entity as response.
* Entity: Has 3 roles. Input Object, Output Object and being object stored in database.

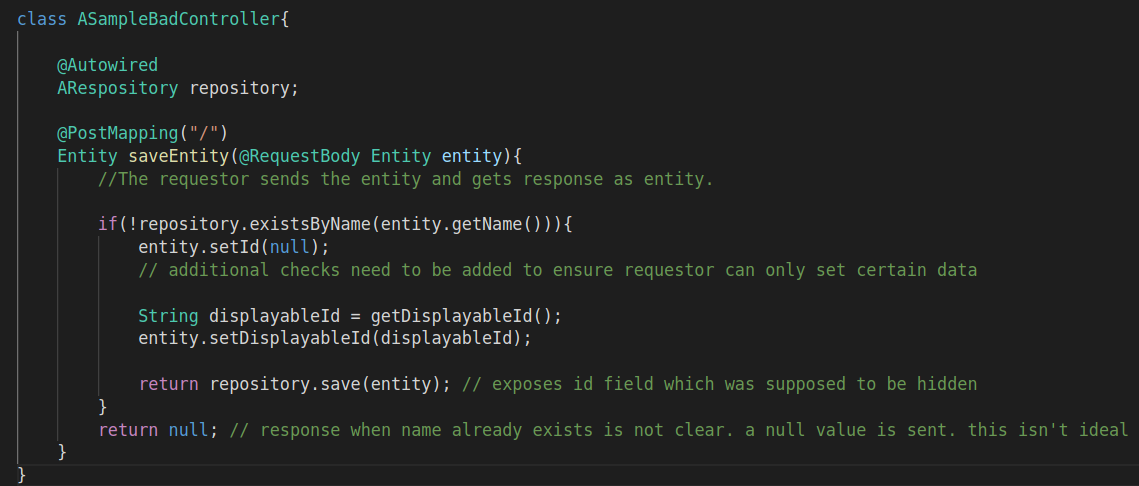
*Code Sample:*

***Entity***



**Repository**:



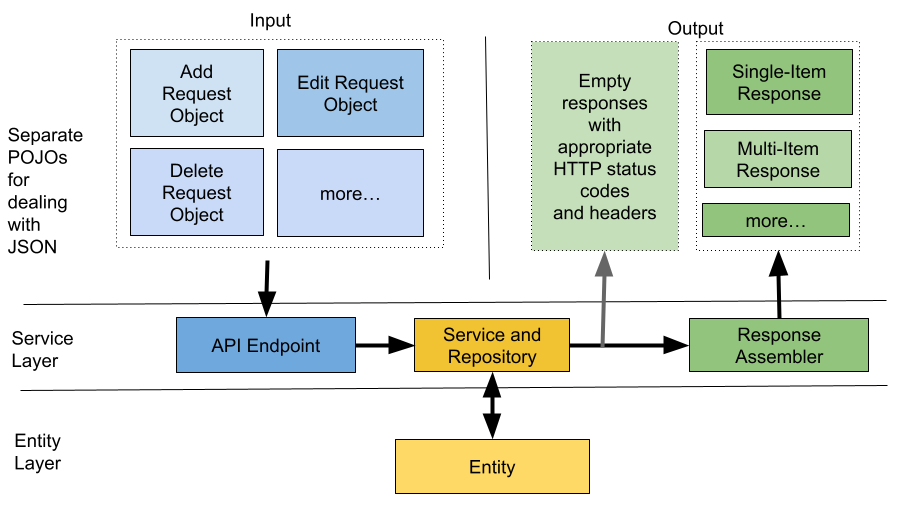
**Controller**:  


This way is problematic since it,

* Many fields used for one role (like input) won’t be used for other role (like storing to DB)
* Internal details are exposed in output
* Validations are harder for inputs

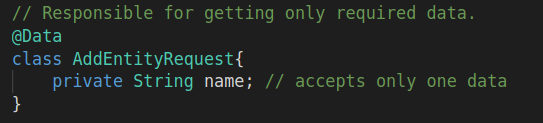
Many linters recommend having a separate DTO object for this task. This is a step up over the usual entity but isn’t sufficient as most of the time the entity and DTO are exact replicas of each other.

There is a better alternative to this, which is to separate inputs, output and entity classes entirely..

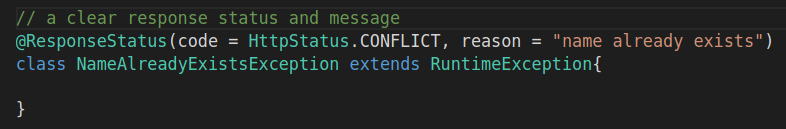


*Code Sample:*

***Request Object:***



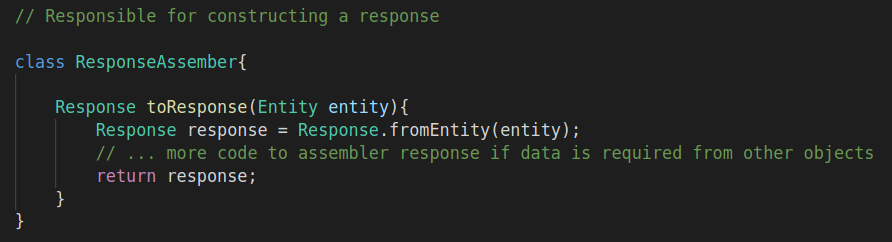
**Empty Body Response:**



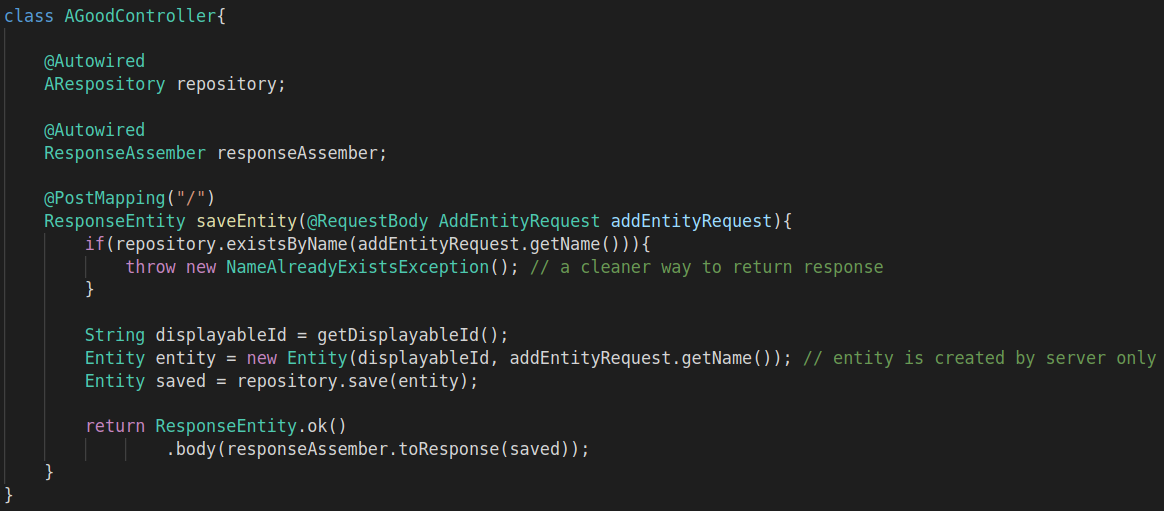
**Response With Body:**



**Response Assembler:**



**Controller:**

  
As you can see from the example above, there are many request objects and a single response object. Each of these requests and responses can have different fields.

**API endpoint**: deals with getting different formats.

**Service & Repository**: only deal with manipulating entity files based on the type of request. It then either sends a response directly if there is no body or offloads it’s work to a response assembler.

**Response Assembler**: job is to create a response. There may be multiple types of responses which response assembler may assemble

There can also be requests and responses that have no body at all.

For example:

* Request params maybe enough for GET and DELETE request
* Similarly, HTTP status code and response header maybe enough for responses
  + If an object is created, then 201 CREATED with response headers like location without a body may be enough.

HTTP suggests but does not enforce any kind of rules over requests and responses. But by following them as closely as possible, we can ensure communicating intent without transfer of bulky.

The benefits of this approach is:

1. Request, Response and Entity classes have a clear separation of concern. Each deal with their own specific usage.
2. Therefore, no extra fields are added to each. If new fields are to be added to one part, they may or may not concern other parts.
3. Securing the application becomes easier as less internal details are leaked.

PS: Tip: In most cases, the response object requires some fields from the entity. Commonly DTOs are created and data from entities is copied to response using some sort of mapper. There is a better alternative which is to use the Delegate pattern. The getters of response class can delegate to getters of entity class. In Java specifically, this can be made easier by using Lombok’s @Delegate annotation.