

# JPA implementation patterns: Service Facades and Data Transfers Objects

by Vincent Partington  MVB · Aug. 07, 09 · Java Zone · Not set

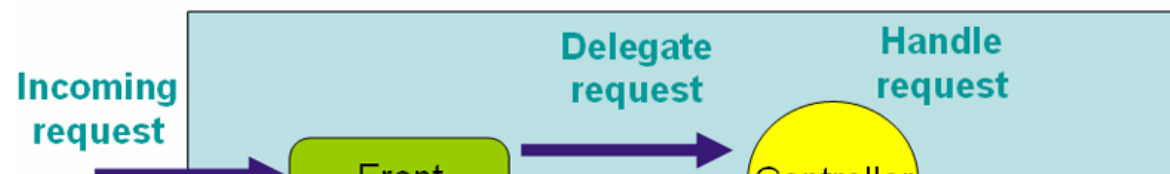
Learn how to build stream processing applications in Java-includes reference application. Brought to you in partnership with Hazelcast.

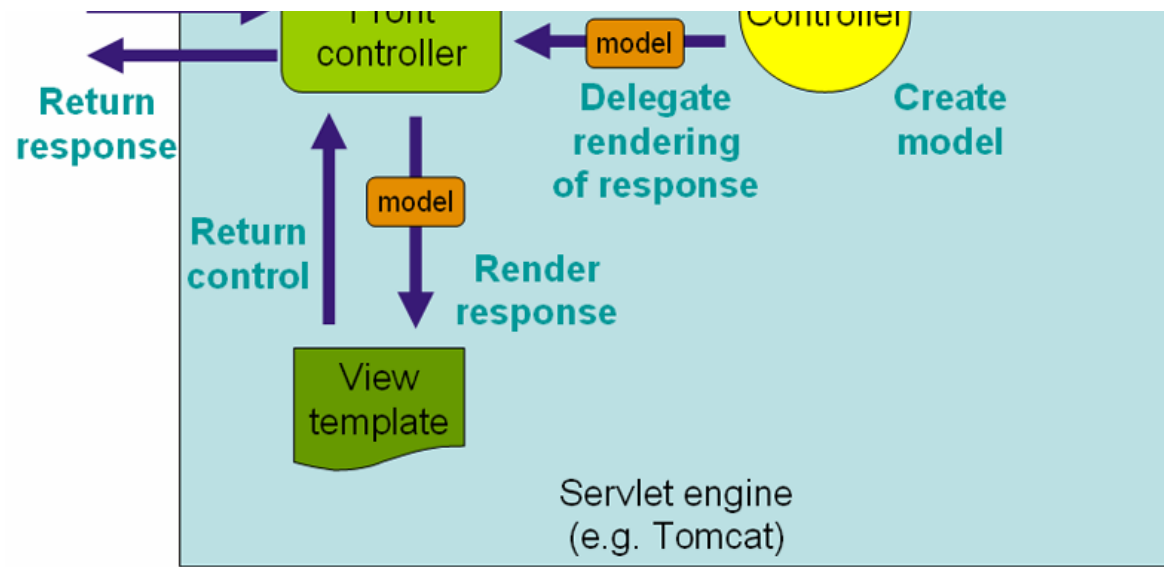
*In this blog I will explore why we would even need such patterns and put these patterns and the DAO pattern into the broader context of JPA application architecture.*

If there is one thing that I learned when implementing JPA for the first time is that some of the "old school" enterprise application architecture patterns still apply, even though some people have proclaimed them to be no longer necessary:

- The DAO has been declared dead because you might just as well invoke the EntityManager directly. It provides a nice enough interface and switching from JPA to a different persistence implementation is not something the DAO abstraction would make much easier.
- DTO's have been deemed superfluous because you can also use your domain objects directly in the presentation layer. This is made possible by a combination of the *open EntityManager in view* pattern, tag libraries to display your domain objects in JSP's and data-binding utilities to map HTTP request parameters back to domain objects.
- And finally Service Facades also seem to have gone out of fashion. Instead you can have the controller directly invoke the services it needs or, even simpler, directly contain the business logic.

The resulting application architecture will look something like this diagram of the Spring Web MVC application architecture:





## So why bother with DAO's, DTO's and Service Facades?

I've already argued why the DAO pattern is still relevant in JPA application architecture. A similar argument can be made for the DTO and Service Facade patterns. While the DTO-less and Service Facade-less architecture displayed above works quite well for straight forward web applications, it has two major drawbacks:

- If you wish to expose your application to other non-HTML clients (think of web services using SOAP, Flex frontends using AMF or Ajax applications using JSON), you will need a more clearly defined interface. That interface specifies which services get exposed to the client and what types are used as the input and output. Service Facades and DTO's respectively help you define this interface.
- Using domain objects directly in the presentation layer requires those domain objects to expose all their fields as public properties with getters and setters. If the fields are not exposed as such, the tag libraries won't be able to render them and the data binding code won't be able to set them. As Allen Holub has already argued before, getters and setters are evil. (BTW, some people interpret Holub's article as an excuse to make all fields public. The article is saying that public getters and setters are no better, but that does not mean your code should just have public fields. Instead the article is advocating use of the tell, don't ask approach to OO.)

In fact the reason I found out why Service Facades and DTO's are still useful, is that the application my team and I are developing has a Flex frontend and a command-line interface that communicate to the core using AMF and Hessian respectively. The only HTML code our application produces loads the SWF file for the Flex frontend! We started out without DAO's, DTO's and Service Facades as one would do, but added them all to our architecture to make this work. And we got a well defined interface between the service layer and the presentation layer as an added bonus.

## Pros and cons of DTO's

Of course it would be silly to say you should always use DTO's in your architecture. As always it depends. 🤖 To allow you to make up your own mind, I will list a number of pros and

Of course it would be silly to say you *should always use DTO's* in your architecture. As always it depends. 😊 To allow you to make up your own mind, I will list a number of pros and cons of using DTO's:

- **Con:** DTO's cause code duplication. This is especially the case when your DTO's have exactly the same fields as your domain objects, and even more so when they both have getters and setters for those fields. But having DTO's in your architecture allows you to get rid of the getters and setters in your domain objects.
- **Con:** DTO's require you to write boilerplate code to copy properties back and forth. Some people have suggested using a Java Bean mapper framework such as Dozer, Apache Commons BeanUtils, or the Spring Framework's BeanUtils class, but that requires you add getters and setters to your beans and we just decided we did not want to do that anymore!
- **Pro/Con:** DTO's make it impossible to use `EntityManager.merge` to copy their state to your persistent objects. Instead you'll have to apply the *DIY merge pattern* as described in my blog on saving (detached) entities. Of course forcing you to do something a particular way, and one that is not 100% satisfying, is not really an advantage. But at least the DTO and the DIY merge work well together.
- **Pro:** DTO's ensure you are not hit by unexpected lazy loading problems in your presentation layer. Or in the case of remote invocations they protect you from lazy loading problems when serializing for transport or, even stranger, on the client.
- **Pro:** The DTO pattern forces you to think about the interface of your application. You can make your DTO's richer than your plain domain objects, e.g. by adding security information to them. Or you can group information from multiple domain objects into one DTO to make the interface easier to use.

Before we move on to discuss Service Facades, it might be worth to have a look at Anirudh Vyas's blog on common abuses of the DTO pattern.

## Pros and cons of Service Facades

Just like is the case with DTO's, there are cases when Service Facades make a lot of sense and there are cases when they just add meaningless overhead. Let's have a look at some pros and cons:

- **Con:** Service Facades add an extra layer that does not do much apart from delegate to the actual service. This argument appeals especially to Java EE developers that have bad memories of the EJB 1.0 based multi-tier architectures we set up in the beginning of this century. 😊
- **Pro:** Service Facades can (should!) be made responsible for mapping from DTO's to domain objects and back. Service Facades are invoked with DTO's as arguments, map them to domain objects, invoke the actual service (or services), map the result back to DTO's and return those DTO's to the client. Actually, to make sure your Service Facades only adhere to Single Responsibility Principle, you should factor the mapping logic out to separate *DTO2DOMapper* and *DO2DTOMapper* classes.
- **Pro:** Service Facades can function as the transaction boundary of your application, i.e. a transaction is started for the duration of a request to the Service Facade. Instead of having to define the transaction attributes of all your services, you can assume that all service invocation arrive through the Service Facade. Actually, setting your Service Facade to be the transaction boundary is obligatory when you want to invoke more than one service during the handling of one request or when you want the Service Facade to map lazily loaded domain objects to DTO's. In that case you end with something akin to the *open EntityManager in view* pattern; a transaction is started that lasts from the moment the incoming DTO's are translated to domain objects to the moment the resulting DTO's are returned to the client. (BTW: if you want to enforce that all services invocations go

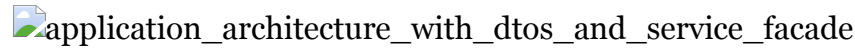
the incoming DTO's are translated to domain objects to the moment the resulting DTO's are returned to the client. (BTW, if you want to enforce that all services invocations go through the Service Facade you can set their transaction attribute to MANDATORY. If a transaction hasn't already been started when such a service is invoked, an exception is thrown)

- **Pro:** The Service Facade pattern forces you to think about the interface of your application. Instead of letting the client invoke all your services, you get to decide which ones to actually expose.

Hmm, I can't seem to think of many disadvantages to the Service Facade pattern. 😊 Apart from the "overhead" argument and that is a subjective thing. Feel free to add more cons of this pattern to the comment section!

## Impact on application architecture

If we apply the DAO, DTO and Service Facade patterns, we end up with a JPA application architecture that looks like this:

application\_architecture\_with\_dtos\_and\_service\_facade

When a request is made, the sequence of events goes something like this:

1. A service client sends a request to the service facade. All objects sent are DTO's.
2. A transaction is started.
3. The service facade invokes the DTO2DOMapper to map the incoming DTO's to domain objects. The DTO2DOMapper may invoke one or more DAO's to load domain objects from the database.
4. The service facade invokes one or more services to perform actual business logic.
5. The service facade passes the return values to the DO2DTOMapper and gets DTO's back. The DO2DTOMapper may invoke one or more services or DAO's to enrich the DTO's.
6. The transaction is committed. Or rolled back in case an exception has occurred.
7. The service facade passes the DTO's to client.
8. The service client receives the DTO's.

In fact, if you take this diagram and replace "DTO2DOMapper" with "Data binding", "DO2DTOMapper" with "View rendering", and "Service Facade" with "Front Controller", you get the original Web MVC architecture we started out with. The big difference is that the incoming "DTO's" are request parameters while the outgoing "DTO's" are HTML:

application\_architecture\_with\_dispatcher\_servlet

That makes this blog come full circle, so now is a good time to wrap up. 😊 As you can see there is no clear-cut answer to whether or not to use DTO's and Service Facades. It all depends on what you are trying to achieve with your application, for example:

- Will it be a straight HTML application or will you want to expose it over different protocols?
- How tightly coupled do you want your clients and your services to be?
- Do you want your domain objects to not have any getters and setters?

*I am very interested to hear what you guys think about the validity of these patterns in modern Java EE architecture. When would you apply then? Or when not and why not? See you all at the next blog in which I will try and tackle the subject of how to handle inheritance in JPA.*

*From <http://blog.xebia.com>*

---

Learn how to build distributed stream processing applications in Java that elastically scale to meet demand- includes reference application. Brought to you in partnership with Hazelcast.

---

## Like This Article? Read More From DZone



**Proposed Jakarta EE Design Principles**



**How to Configure NGINX High Availability Cluster Using Pacemaker on Ubuntu 16.04**



**Advanced Research Initiatives: Focus on the Future**



**Free DZone Refcard  
Getting Started With Kotlin**

Topics:

Opinions expressed by DZone contributors are their own.

# Java Partner Resources

Building Real-Time Data Pipelines with a 3rd Generation Stream Processing Engine

Hazelcast



Level up your code with a Pro IDE

JetBrains



Modern Java EE Design Patterns: Building Scalable Architecture for Sustainable Enterprise Development

Red Hat Developer Program



Build vs Buy a Data Quality Solution: Which is Best for You?

Melissa Data

