Understanding Spring Web Application Architecture: The Classic Way

Petri Kainulainen October 19, 2014 99 comments

Clean Code, Design, Spring Framework

Every developer must understand two things:

1. Architecture design is necessary.
2. Fancy architecture diagrams don't describe the real architecture of an application.

The real architecture is found from the code that is written by developers, and if we don't design the architecture of our application, we will end up with an application that has more than one architecture.

Does this mean that developers should be ruled by architects?

No. Architecture design is far too important to be left to the architects, and that is why **every developer**, who wants to be more than just a type writer, **must be good at it**.

Let's start our journey by taking a look at the two principles that will help us to design a better and a simpler architecture for our Spring powered web application.

The Two Pillars of a Good Architecture

Architecture design can feel like an overwhelming task. The reason for this is that many developers are taught to believe that architecture design must be done by people who are guardians of a mystical wisdom. These people are called software architects.

However, the task itself isn't so complicated than it sounds:

> Software architecture is the high level structure of a software system, the discipline of creating such a high level structure, and the documentation of this structure.

Although it is true that experience helps us to create better architectures, the basic tools of an architecture design are actually quite simple. All we have to do is to follow these two principles:

1. The Separation of Concerns (SoC) Principle

The Separation of Concerns (SoC) principle is specified as follows:

> Separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern.

This means that we should

1. Identify the "concerns" that we need to take care of.
2. Decide where we want to handle them.

In other words, this principle will help us the identify the required layers and the responsibilities of each layer.

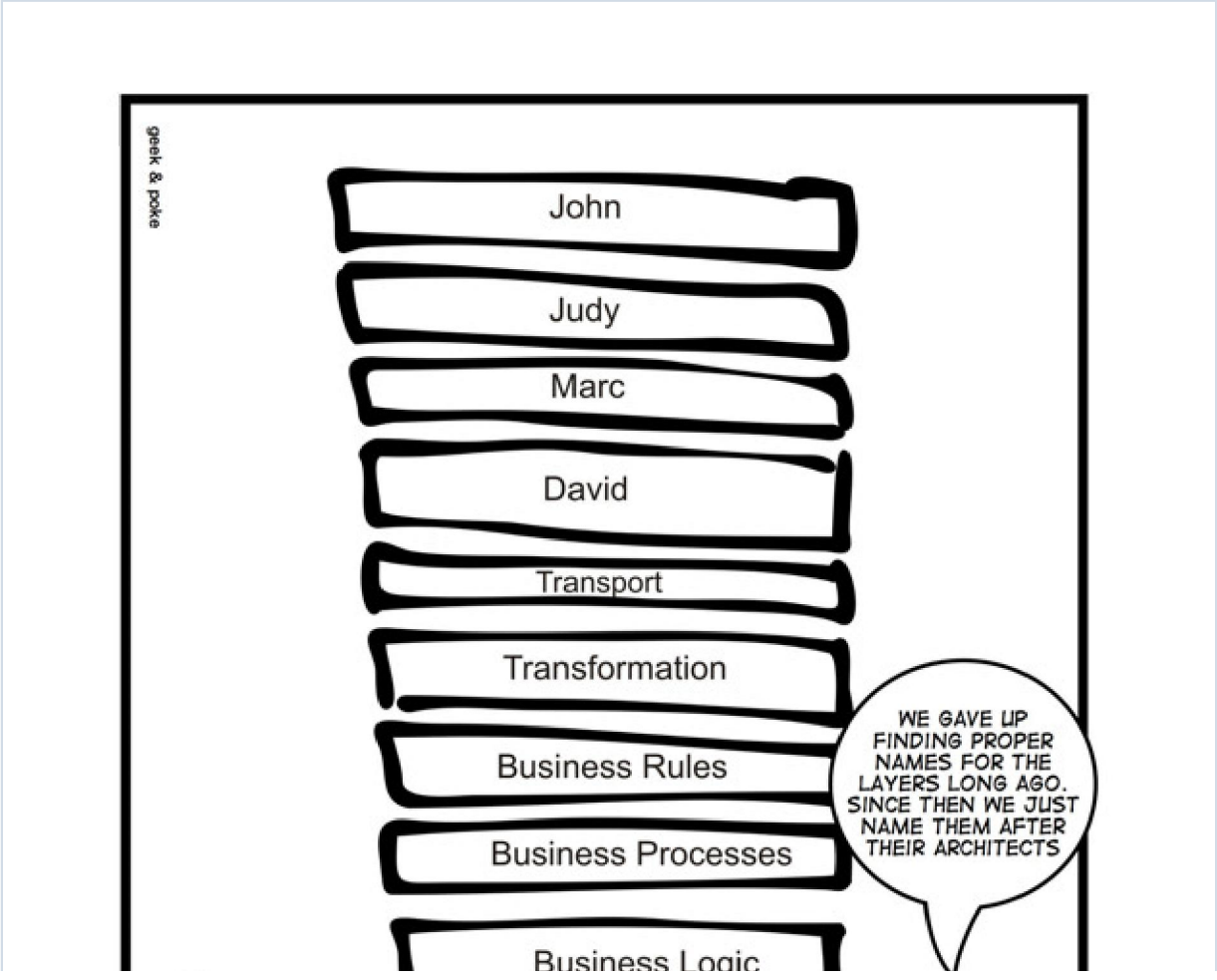
2. The Keep It Simple Stupid (KISS) principle

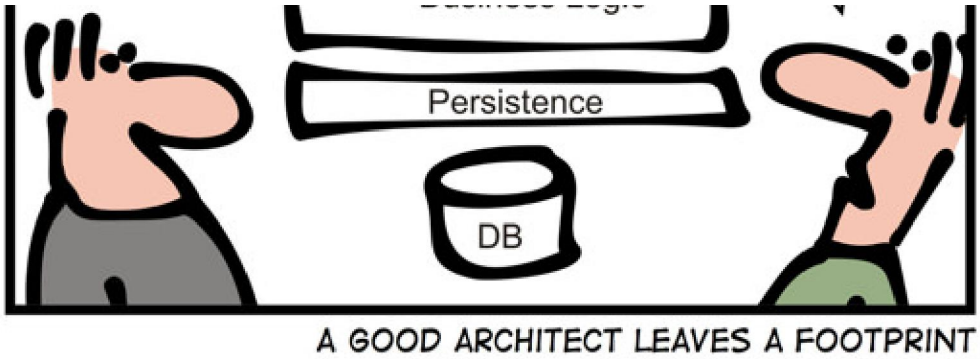
The Keep It Simple Stupid (KISS) principle states that:

> Most systems work best if they are kept simple rather than made complicated; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided.

This principle is the voice of reason. It reminds us that every layer has a price, and if we create a complex architecture that has too many layers, that price will be too high.

In other words, **we should not design an architecture like this**:





Source: Geek And Poke: Footprints – Licensed under CC 3.0

I think that John, Judy, Marc, and David **are guilty of mental masturbation**. They followed the separation of concerns principle, but they forgot to minimize the complexity of their architecture. Sadly, this is a common mistake, and its price is high:

1. Adding new features takes a lot longer than it should because we have to transfer information through every layer.
2. Maintaining the application is ~~pain in the ass~~ impossible because no one really understands the architecture, and the ad-hoc decisions, that are made every, will pile up until our code base looks like a big pile of shit that has ten layers.

This raises an obvious question:

What kind of an architecture could serve us well?

Three Layers Should Be Enough for Everybody

If think about the responsibilities of a web application, we notice that a web application has the following “concerns”:

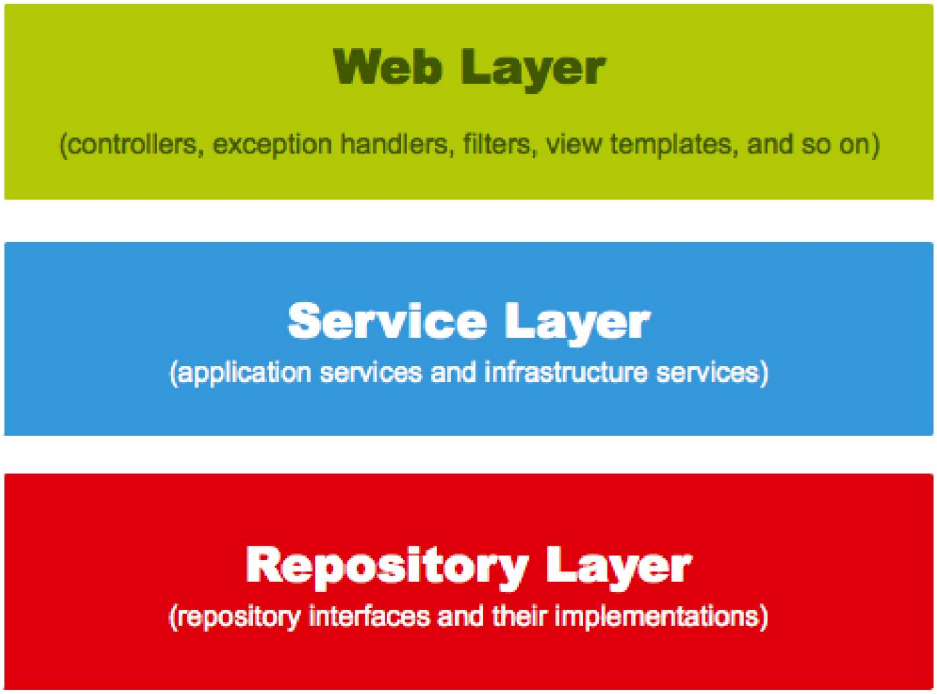
- It needs to process the user's input and return the correct response back to the user.
- It needs an exception handling mechanism that provides reasonable error messages to the user.
- It needs a transaction management strategy.
- It needs to handle both authentication and authorization.
- It needs to implement the business logic of the application.
- It needs to communicate with the used data storage and other external resources.

We can fulfil all these concerns by using "only" three layers. These layers are:

- **The web layer** is the uppermost layer of a web application. It is responsible of processing user's input and returning the correct response back to the user. The web layer must also handle the exceptions thrown by the other layers. Because the web layer is the entry point of our application, it must take care of authentication and act as a first line of defense against unauthorized users.
- **The service layer** resides below the web layer. It acts as a transaction boundary and contains both application and infrastructure services. The **application services** provides the public API of the service layer. They also act as a transaction boundary and are responsible of authorization. The **infrastructure services** contain the "plumbing code" that communicates with external resources such as file systems, databases, or email servers. Often these methods are used by more than a one application service.
- **The repository layer** is the lowest layer of a web application. It is responsible of communicating with the used data storage.

> The components that belong to a specific layer can use the components that belong to the same layer or to the layer below it.

The high level architecture of a classic Spring web application looks as follows:



The next thing that we have to do is to design the interface of each layer, and this is the phase where we run into terms like data transfer object (DTO) and domain model. These terms are described in the following:

- A **data transfer object** is an object that is just a simple data container, and these objects are used to carry data between different processes and between the layers of our application.
- A **domain model** consists of three different objects:
  - A **domain service** is a stateless class that provides operations which are related to a domain concept but aren't a "natural" part of an entity or a value object.
  - An **entity** is an object that is defined by its identity which stays unchanged through its entire lifecycle.

- A **value object** describes a property or a thing, and these objects don't have their own identity or lifecycle. The lifecycle of a value object is bound to the lifecycle of an entity.

Now that we know what these terms mean, we can move on and design the interface of each layer. Let's go through our layers one by one:

- The web layer should handle only data transfer objects.
- The service layer takes data transfer objects (and basic types) as method parameters. It can handle domain model objects but it can return only data transfer objects back to the web layer.
- The repository layer takes entities (and basic types) as method parameters and returns entities (and basic types).

This raises one very important question:

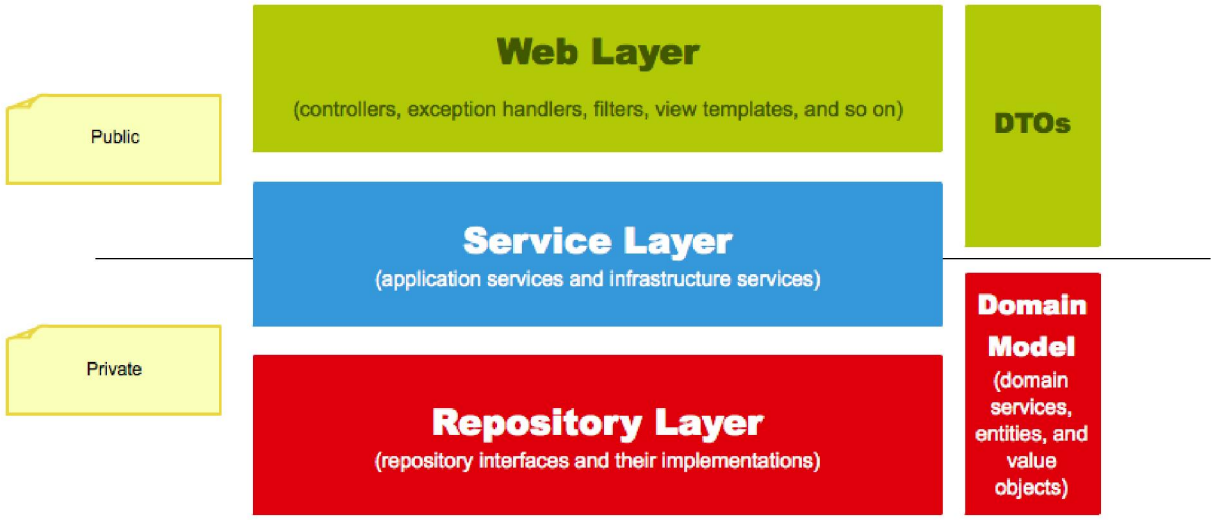
> Do we really need data transfer objects? Why cannot we just return entities and value objects back to the web layer?

There are two reasons why this is a bad idea:

1. The domain model specifies the internal model of our application. If we expose this model to the outside world, the clients would have to know how to use it. In other words, the clients of our application would have to take care of things that don't belong to them. If we use DTOs, we can hide this model from the clients of our application, and provide an easier and cleaner API.
2. If we expose our domain model to the outside world, we cannot change it without breaking the other stuff that depends from it. If we use DTOs, we can change our domain model as long as we don't make any changes to the DTOs.

The "final" architecture of a classic Spring web application looks as follows:

very good architecture



There Are Many Unanswered Questions Left

This blog post described the classic architecture of a Spring web application, but it doesn't provide any answers to the really interesting questions such as:

- Why the layer X is responsible of the concern Y?
- Should our application have more than three or less than three layers?
- How should we design the internal structure of each layer?
- Do we really need layers?

The reason for this is simple:

We must learn to walk before we can run.

The next blog posts of this tutorial will answer to these questions.