Navigation:

Ben Nadel

On User Experience (UX) Design, JavaScript, ColdFusion, Node.js, Life, and Love.

« Previous Photo

Next Photo »

Application Services vs. Infrastructure Services vs. Domain Services

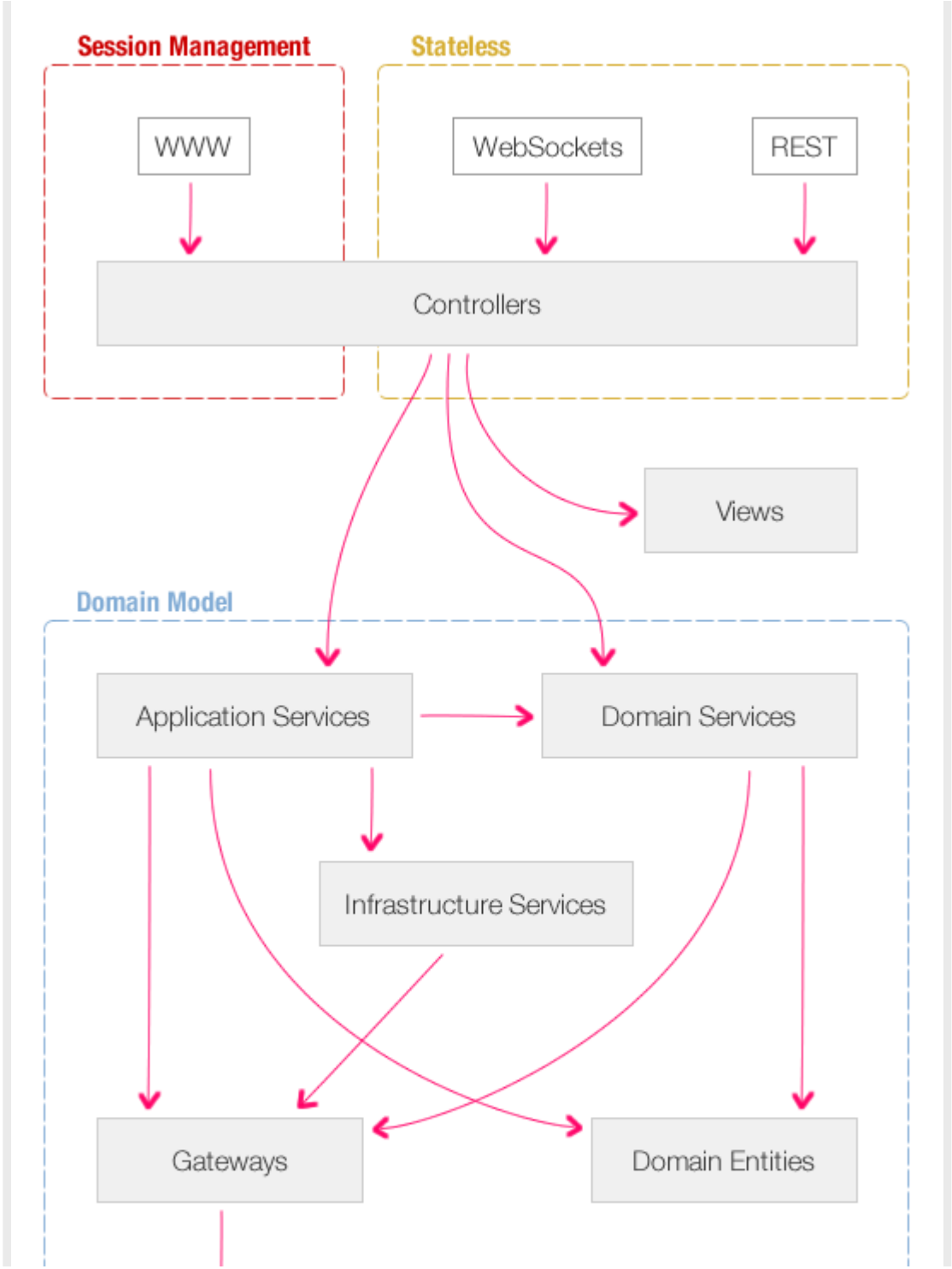
By Ben Nadel on June 6, 2012
Tags: ColdFusion

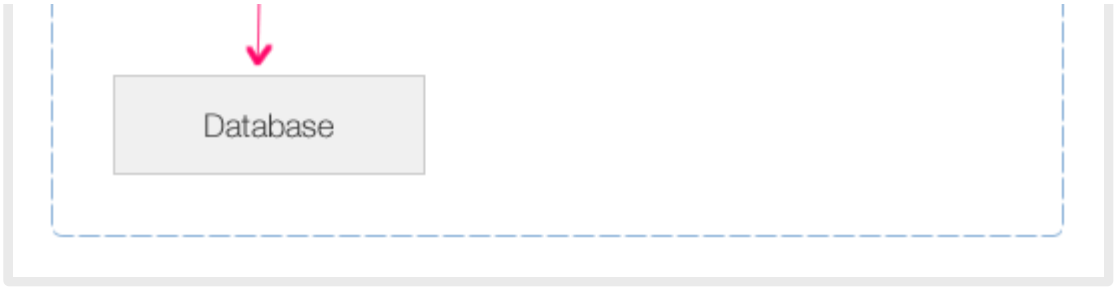
Yesterday, I outlined my current understanding of Application Architecture and the Model-View-Controller (MVC) approach to content delivery. Part of the application architecture included a "Service" layer. Traditionally, I have always thought of the Service layer as a single layer comprised of a single type of object (which, in my case, was actually Gateways - but that's a whole other problem). Recently, however, I have begun to think of the service layer as containing several different type of objects, each with its own kind of role and dependencies.

From what I've been reading lately, it seems that the service layer is composed of three distinct types of services:

- Application Services
- Infrastructure Services
- Domain Services

Integrating this breakdown into my previous application architecture diagram, we now get something that looks like this:


Session Management
Stateless
WWW
WebSockets
REST
Controllers
Views
Domain Model
Application Services
Domain Services
Infrastructure Services
Gateways
Domain Entities



I don't actually know if all of these Service objects technically belong inside something called the "Domain Model;" but, for now, I'll just keep them tucked in the same conceptual model. At this point, I don't think I could actually define what the domain model should encompass.

As you can see in the diagram, not all Service objects are created equal. Some can be accessed by the Controller, some cannot. Some can be accessed by other service objects, some cannot. At first, I found this separation annoying; but, the more I read about it, and the more I thought about it, the more it started to make sense. By breaking down services by "role," and not just by "problem domain," you can start to see how things might be wired together more effectively (and maintainably).

Application Services

Sometimes referred to as "Workflow Services" or "User Cases", these services orchestrate the steps required to fulfill the commands imposed by the client. While these services shouldn't have "business logic" in them, they can certainly carry out a number of steps required to fulfill an application need. Typically, the Application Service layer will make calls to the Infrastructure Services, Domain Services, and Domain Entities in order to get the job done.

I believe that the Application Service objects are "command" objects. That is, in relation to the command/query separation, you wouldn't talk to an Application Service simply to retrieve information. Instead, you'd go directly to the Domain Services for such information.

But, I am not 100 percent sure on that; if your "query" request involved additional actions such as logging or billing, then I'd hazard a guess that it would be part of the Application Service. Furthermore, if you didn't want to return domain entities to

the Controller, I supposed you'd have to go through the Application Service layer for data requests such that it could transform the data.

The Application Service can only be called by the Controller. Since the Application Service orchestrates application workflow, it would make no sense for them to be called by the Domain Service (or even by themselves) - a workflow has only one single starting point; if an Application Service could be called by other entities within the domain model, it would imply that a workflow has an indeterminate number of starting points.

Infrastructure Services

These are services that typically talk to external resources and are not part of the primary problem domain. The common examples that I see for this are emailing and logging. When trying to categorize an Infrastructure service, I think you can ask yourself the following questions:

- If I remove this service, will it affect the execution of my domain model?
- If I remove this service, will it affect the execution of my application?

If it will affect your domain model, then it's probably a "Domain Service." If, however, it will simply affect your application, then it is probably an Infrastructure Service. So for example, if I removed Email Notifications from an application, it probably wouldn't affect the core domain model of the application; it would, however, completely break the usability of the application - hence it is an Infrastructure service (and not a Domain Service).

In my diagram, the Infrastructure service can only be invoked by the Application Service. Given the above questions, this makes sense. If the Domain Services or the Domain Entities could invoke the Infrastructure Services (think Logging, think Email confirmations), then you'd probably have application logic leaking into your domain model. Plus, since logging and emailing is typically part of some greater "workflow," it makes sense that they'd be called from the "Workflow Services" (ie. Application Services).

NOTE: I have seen a number of people categorize Database access as an Infrastructure concern. In my diagram, I treat persistence encapsulation as a separate concept - Gateways. I am not sure how to reconcile this. If I moved Gateways into Infrastructure, then it

would mean the Domain Services could invoke the Infrastructure services. I'm not quite sure how I feel about that just yet - it might open up a whole can of worms. ***Data persistence is probably the biggest monkey wrench*** *in my understanding of all of this!*

Domain Services

These are services that coordinate domain-level actions that are application-agnostic. From what I have read, they provide a means to ensure the integrity of the domain model by encapsulating CRUD (Create, Read, Update, Delete) operations and data access. Plus, they provide a place to put behaviors that don't really fit into any single Domain Entity.

Right now, this is a lot of theory and thinking and almost no code. I am sure that as I start to try and put some of this thought into practice, my understanding will continue to evolve. Heck, you can see how much my thinking has already evolved since my application architecture post yesterday. Wild stuff!

B N

Enjoyed This? You Might Also Enjoy Reading:

- Patterns Of Enterprise Application Architecture By Martin Fowler
- Thinking About Tracking Requests And Errors Across Micro-Services
- Streamlined Object Modeling: Patterns, Rules, And Implementation
- What If All User Interface (UI) Data Came In Reports?

B N

Erik Meier
Jun 6, 2012 at 11:56 AM

6 Comments

Nice in depth analysis, I haven't spent the time to break it down at a high level like you have, we've just been building and separating concerns as we go.

For our structure, we maintain a /lib directory for the "Infrastructure services" layer, but maintain an "application service" that has a consistent API which wraps the calls to the libraries.

E.g. a "CacheService" abstracts calls to our memcached client, so if we *wanted* we could swap out implementation and only need to update a single class. Similarly a "SearchService" abstracts calls to Endeca, which could theoretically be swapped out for Solr or some other search engine.

The other advantage to this isolation is testability; we haven't yet, be we could create mock interfaces to these external resources to have true unit testing.

I'm curious how others handle these things...services like caching, search, webex, twitter, etc...aren't really part of my domain model, so I don't like having them there. At the same time, our /lib contains a mix of 3rd party libraries, e.g. ColdSpring and our own wrappers.

Phil Duba
Jun 6, 2012 at 12:46 PM

27 Comments

I'm glad you're still going down this path Ben. You have a lot of the same questions I do and after attending Scott's CF Dev Week presentation yesterday, I feel I'm more confused then ever. I still have to digest this post some more before I ask intelligent questions.

Ben Nadel

Jun 6, 2012 at 2:33 PM

14,280 Comments

@Erik,

Yeah, the definition of "domain model" is definitely something that I feel unsure about. As you say, things like Twitter don't really fit into the primary domain model of an app. I suppose those are all the "Infrastructure" services I was talking about; which makes me think they aren't really part of the "domain model."

In a Node.js application, I see people follow a similar separation: "lib" for 3rd party stuff and "node_modules" for application-specific stuff. Still trying to sort it all out in my head.

@Phil,

Glad I'm not the only one that gets a bit confused by all of this! I feel like I need to set aside a weekend just to hammer out some code.

Aaron Greenlee

Jun 6, 2012 at 3:55 PM

16 Comments

Love the diagrams.

Sean Corfield

Jun 6, 2012 at 8:21 PM

122 Comments

I fear you may be overanalyzing things and making life more complicated for yourself... :)

I'm not sure I buy the separation of "application" services from "domain" services. There's just domain entities (business objects) and orchestration services: services that exist purely to orchestrate operations across multiple business objects - i.e., for things that an individual business object can't do by itself.

Separating infrastructure from persistence is slightly more reasonable but in my experience infrastructure is the lowest level, below the data layer.

I'd expect calls from controllers to hit services or domain entities directly. Domain entities may delegate to services to perform certain tasks (where that provides a better API on the business object).

Services will call down into the data layer and the infrastructure layer. The data layer may call down into the infrastructure layer.

I think that's a cleaner separation:
* Business Domain (services and entiities)
* Data Layer
* Infrastructure Layer

You could even lump the data layer and infrastructure layer together (but in my mind the data layer contains some smarts about marshaling data between your business domain and the backing store, whatever that is, whereas the infrastructure layer is pretty dumb - it interfaces with the "backend" (environment management, hardware/network services, plumbing to 3rd party services).

Of course, in the CFML world the infrastructure part of the "data layer" is hidden from you - JDBC, datasource configuration, connection pooling. In the CFML / Clojure hybrid that I inhabit, my CFML services / business objects talk to a data layer (CFML) that talks to an infrastructure layer (Clojure) which talks to the database. That infrastructure layer includes clojure.java.jdbc (the Clojure wrapper for JDBC) and congomongo (the Clojure wrapper for MongoDB) as well as explicit connection pooling (via c3p0) and other environment-specific stuff.

I guess my main concern is you're going to end up with a plethora of CFCs that do a lot of delegation to other CFCs, just so you can maintain the separations you are trying to inflict upon yourself. My general mantra is to try to keep things as simple as possible and only as complex as necessary.

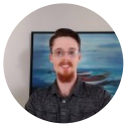
Sean Corfield

Jun 6, 2012 at 8:40 PM

122 Comments

Here's what I had in mind with my last comment:

http://corfield.org/articles/mvc-arch.pdf



Steven Neiland

Jun 6, 2012 at 10:04 PM

61 Comments

@Ben,
I second what Sean said. Your concepts do make sense in theory but you are over thinking this by trying to make your architecture pure when really its not necessary. Often the lines get blurred.

Take infrastructure vs gateways.

You have gateways as components that access the database, while infrastructure components are those that access external resources. But when you get down to it, a database is an external resource.

In your example what you actually sound like you mean is that you consider one way communication to external resources as infrastructure. You send an email or log an action and continue on regardless of if anything really happened. Almost a fire and forget type scenario.

Now take a twitter communication component that just sends tweets out. This is no different to an email component that sends out email. You tell it to send a tweet and continue on. This would make it infrastructure.

However what if later on you decide you want to pull back data from twitter, say your number of followers. You add a few methods to pull back data but make no changes to those that send data out.

This behaviour is no different to pulling data from a database which really means the component is now a gateway. The line is blurred without ever changing how existing parts of your application use the component.

What you need to do is simplify the concept down to its most basic elements. When I think of a gateway service, I define it as this.

A gateway is a component accessible by my application that allows the application to pass data into or pull data out of another system.

What infrastructure components really do is handle back end stuff that enables communication. Network protocols and connection handling etc. This is all handled by CF so you really don't have infrastructure components. Now if you were to create say a component that opened and maintained a ssh connection that would be.

Brian Kotek

Jun 7, 2012 at 1:31 AM

116 Comments

Ben, I read your post and went down to comment, then saw that Sean pretty much said what I was going to. Looks like you may be over-analyzing this. Services are primarily there to act as an interface to your domain model, persistence layer and other supporting classes (such as adapters for external services, utility classes, etc.).

Beyond that, I think some of your assumptions and rules may be off base. For example, the diagram and your comments indicate that these services can only be invoked through a controller. For a page-based HTML application

that's usually the case. But other consumers, such as AMF, REST/AJAX, or SOAP will normally be hitting the services directly. Or more precisely, they'll be hitting a protocol-specific set of Proxy objects that do nothing but handle converting the incoming request and the outgoing data into the proper format for that protocol.

There's really no reason to bog things down and jump through hoops to route the requests through a controller layer that is meant to manage an MVC flow for a page-based web application. Unless I'm misunderstanding what you mean by a "controller".

Further, there's most definitely cases where you need session management for REST, AJAX, and AMF requests. Almost any UI using your services will need this, even if it isn't a traditional request to render something as HTML.

This is just my opinion, of course, and there's certainly many viewpoints on the best way to tackle these problems. Still, I hope some of my two cents are a bit useful to you. ;-)

Erik Meier

Jun 7, 2012 at 7:43 AM

6 Comments

@Sean

My general mantra is to try to keep things as simple as possible and only as complex as necessary.

I think this is really the key. One of ColdFusion's strengths is its simplicity and adding multiple layers of abstraction really goes against that.

I find that it is sometimes difficult to walk the line and a lot of my design decisions may be overreactions to maintaining legacy applications where 5000 line CFCs and 500 line functions are the norm.

Ben Nadel
Jun 7, 2012 at 8:56 AM

14,280 Comments

@All,

Just as a first, sobering thought to put all of this in perspective - I typically **under-analyze** things. And, it's gotten me pretty far :D I mean, I have been building ColdFusion applications for a long time and they've been successful. But, they tend to reach a point where I pass the "payoff" line and the things that I didn't think deeply enough about at first come back to haunt me.

So, you could say that I should refactor when I get to that point. And, I guess what I'm getting at is that, when I DO get to that point, I don't even necessarily know how to refactor - I don't know which choices make more sense - I only know that I have something which is hard to maintain.

What I want now if for the pendulum to swing the other way - from under-analysis to over-analysis :)

@Steven,

I think what you're pointing out about my Infrastructure vs. Gateways is definitely a spotlight into a flaw in my thinking. That's awesome! It didn't feel right (I even confessed that persistence is confusing me); and I can see that perhaps that kind of separation is now necessary.

@Brian,

While I am clearly *trying* to over-analyze this stuff, perhaps part of our disconnect is simply naming? When you refer to "Proxy" objects for non-www calls, I think that's what I'm referring to as a "Controller."

I admit that my graphic makes is look as if there is only one unified controller for the whole app. What I meant to say is simply that there is a Controller "layer" that incoming requests go through. If all the API controller does is hit

services and turn them into JSON responses, I'm fine with that - to me, that is the "traffic cop" role that the Controller is supposed to be.

One thing that I really want to try to do is keep the Domain Model unaware of any delivery mechanism. So, while the front end might be getting HTML, JSON, XML, SOAP, or AMF (thought I know nothing about AMF), I don't want my domain model to care. Whatever sits between the client and the domain model is, I think, what should handle that transformation / rendering.

If you want to call that a Controller in some cases and a Proxy in others, I am fine with that.

And, as far as REST being session-based sometimes, most definitely. In my apps that are AJAX-driven, I definitely rely on the ability for AJAX calls to pass in session-cookies. My server-side API (for the sister www-app) definitely is session-based... though, I admit it's not really REST - just AJAX-driven POSTs with form values being passed.

... Ok, now I gotta think for a few minutes on what this all means for the above diagram :)

THANK YOU all for your most awesome feedback!!

Ben Nadel

Jun 7, 2012 at 9:20 AM

14,280 Comments

@Sean,

One thing that I wanted to clarify is that when it comes to Application Services and Domain Services, I don't see a lot of duplication happening. Meaning, the Application Service isn't simply handing requests off to a Domain Service to execute the same logic.

Things that exist in Application "portion" of the service layer would use the Domain "portion" of the service layer to carry out *part* of the workflow.

... just wanted to clarify that I don't see a lot of duplication (if that was a concern).

Ben Nadel

Jun 7, 2012 at 9:59 AM

14,280 Comments

@All,

Right now, this is all so theoretical; so, I wanted to come up with a small, concrete example of things I am thinking about, as far as the Application vs Domain services go.

First, I want to start referring to the "Application service" as the "Workflow service" as I think this helps me understand what it's doing - orchestrating use case execution.

Image an accounting system where we want to transfer money from one account to another. For this, the Controller would invoke the "Workflow service" with something like:

Workflow.transferMoney(fromID, toID, ammount)

This would take simple values because it's just coming from the Controller which is just passing off requests to the app.

The "workflow" within the transferMoney() method might look like this:

* from = accountGateway.get(fromID)
* to = accountGateway.get(toID)

* accountService.transfer(from, to, amount)
* emailGateway.sendTransferNotice(from)

... or something like that.

In this case, the "workflow" service is orchestrating several domain objects and a domain service.

The "accountService" object, referenced within the workflow, is a "Domain Service" that handles the transfer between two accounts.

The emailGateway is part of the orchestration because it is not a necessary part of the financial transfer (only part of the experience of the application).

Now, as a thought experiment, what happens if we want to record a "Ledger Transaction" for the transfer. Meaning, that somewhere, I want to record the Debit for the source account and the Credit for the target account. Who takes care of that?

My instinct would say, It should be part of the accountService.transfer() method - whenever you transfer money from one account to the other, there should always be a Ledger item recorded.

BUT, is that really any different from an email notification? I can also say an email notification should always go out any time money is transferred. But, I wouldn't put the email notification in the accountService - I'd leave that in the "workflow" service.

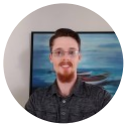
So perhaps the question should be, "If we didn't have a Ledger, would the account transfer still make sense as a concept?"

I think Yes. Transfers make sense without a Ledger the same way they make sense without an Email notification. Therefore, the creation of the Ledger should be part of the "workflow", not part of what it means to transfer money at the Domain level.

So, maybe the workflow might look like this:

* from = accountGateway.get(fromID)
* to = accountGateway.get(toID)
* accountService.transfer(from, to, amount)
* ledgerService.debit(from, amount)
* ledgerService.credit(to, amount)
* emailGateway.sendTransferNotice(from)

Anyway, I thought a more concrete example might help bring this together a bit more?



Steven Neiland

Jun 7, 2012 at 12:46 PM 61 Comments

@Ben,
That example makes it much clearer. I think you have the conceptual idea now.

I would probably put the ledger function in the domain section conceptually, but really once you are at this stage it only really starts to gel when you start writing code.

Dave Leeds

Jun 7, 2012 at 1:12 PM 4 Comments

@Ben, thanks for your concrete example! For me, that definitely clarifies the line you're drawing between the Workflow Service and the Domain Service. The approach you outlined would work fine.

Another solution for simple workflows might be to have hooks for interceptors in the AccountService for things like ledger logging and emailing. For example, you could pass in an email interceptor and a legder interceptor when constructing the AccountService. Both of those would conform to the same interface (e.g., OnFundsTransferred(from, to, amount)). They get the necessary arguments to send the email and write to the ledger, but the Account Service doesn't know much about them -- just that they need to be called when the funds are transferred.

Naturally, if we're dealing with a more complex workflow (say, with conditionals and loops), interceptors might not be as helpful.

Brian Kotek

Jun 7, 2012 at 1:29 PM

116 Comments

One thing that I really want to try to do is keep the Domain Model unaware of any delivery mechanism. So, while the front end might be getting HTML, JSON, XML, SOAP, or AMF (thought I know nothing about AMF), I don't want my domain model to care.

Absolutely. In fact nothing in any of your services or models should care about the request protocol. The only thing that should care is the Proxy (or as you call it, "controller") layer, since it is specifically responsible for handling any necessary conversion.

Ben Nadel

Jun 7, 2012 at 2:06 PM

14,280 Comments

@Steven,

Yeah, I started with code, then jumped into philosophy + architecture. Then back to code. Then back to philosophy :) I think it's time to jump back into code again.

Right now, I have a lot of non-descript anxiety about what objects should have access to what objects (ex. Should my domain service have access to a Gateway, etc.). I need to start letting feeling it out in code.

@Dave,

That's an interesting idea; kind of like a callback / event binding system for service events. I've really only dealt with "events" at client-side level (if you don't consider the Controller as responding to events).

Right now, since I'm coming from such a Procedural background, I think I might have to forgo the interceptors, if for no other reason, I can *see* the actions that are going to take place. Once I get more comfortable, I can definitely evolve my wiring.

@Brian,

Ah, ok cool - sounds like we were having a naming conversation (at least at the Controller level). Slowly, it's starting to come together ... maybe.

Sean Corfield

Jun 7, 2012 at 2:48 PM

122 Comments

@Ben,

So, you could say that I should refactor when I get to that point. And, I guess what I'm getting at is that, when I DO get to that point, I don't even necessarily know how to refactor - I don't know which choices make more sense - I only know that I have something which is hard to maintain.

Refactoring works best when you do it continuously. That way you avoid the "Big Refactoring" because you're always doing small refactorings to make things better.

Your example makes your line of thinking much clearer, thank you, but I think it also highlights that you're trying to achieve an artificial separation - and I think that's why you're struggling so much to "get it right". By trying to find the exact bucket for every small piece of functionality, you're constraining your design too much, rather than letting it grow organically.

The key thing here is that there's no One True Way(tm) to implement this - something that a lot of people trying to learn OOP and/or design patterns struggle with: they want to know "the best way" and there isn't one, there are just tradeoffs.

Ben Nadel

Jun 8, 2012 at 9:21 AM

14,280 Comments

@Sean,

I could definitely be much better about continuous refactoring. That is something I sorely miss for the usual excuses - no time, too much to get done, yada yada yada.

I will definitely admit that I get a bit lost in the abstract; and this has everything to do with an anxiety that I am going to get myself into a situation that I won't know how to get out of. I think Steven is right - at this point, I really just have to Code before it will start to make any more sense.

I start working on code when I got back from cf.Objective():

https://github.com/bennadel/MVC-Learning

... but I am not sure I am crazy with the content - a ToDo app. Probably, mostly, I am just feeling lazy about making all my Views :) I haven't it touched it in a while - time to get back to it, refactor, build, learn!!

Sean Corfield

Jun 8, 2012 at 4:14 PM

122 Comments

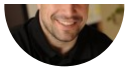
@Ben,

Well, continuous refactoring doesn't take any time if you do it... continuously. It's just part of your normal development process.

Refactoring only "takes time" if you don't do it continuously and then you face "The Big Refactor" which, of course, is a scary beast and will definitely eat up a bunch of time because now you've accrued a lot of technical debt that you need to pay back...

One of the problems with small example apps is that they're too small to benefit from the sort of separation you're talking about in these blog posts - they just end up looking complicated. It's why KISS and YAGNI are important - and why continuous refactoring ensures you don't over-engineer things: you build the simplest thing that works and continuously evolve it into what you need over time.

When I start a small app, I have everything in the controller(s) and the domain objects and as the app grows, I refactor things into various service layers. Since you have to edit code to change it, making small improvements every time you change code is reasonable: it's just part of the normal editing cycle. It ensures you deliver what's needed without spending time building something you don't need (yet) and you do "just enough" to ensure you have something maintainable.



Ben Nadel

Jul 2, 2012 at 10:26 AM

14,280 Comments

@Sean,

I think you make a very profound point. One thing that I am definitely bad about is continuous refactoring. And, it's probably one of the things that would have the biggest impact on the overall maintainability of my code.

Years ago, when I tried doing my OOPhoto project, I did start our with a fully procedural code, and then tried to refactor; but, my refactor was not successful. But, granted, that was like 5 years ago!

Perhaps what I'll do is implement my ToDo practice app in full procedural code and then refactor it. I understand that it's probably too simple to need so much refactoring; but, I think it will be good practice.

Sean Corfield

Jul 2, 2012 at 1:53 PM

122 Comments

@Ben,

That's still missing the point of continuous refactoring: you don't take a whole app and refactor it, instead you refactor code as you write it. With a TDD approach, you might write a test (red), write the simplest procedural code to make it pass (green), immediately rewrite that into a more OO style (or a more FP style) - while keeping all tests green - then rinse and repeat. In this way, you write the simplest possible code to solve the problem, using the idiom you know best, and as soon as each _piece_ works, you can refactor to a more structured solution _that still works_.

And, in fact, TDD at the object level will help you create an OO solution out of the gate.

Sean

Ben Nadel

Jul 5, 2012 at 9:59 PM

14,280 Comments

@Sean,

Good sir, you are right. I think sometimes I get so concerned with simply "moving forward," that I don't always think about what the right path actually is. Somehow, I think if I can just keep moving, it'll all make sense eventually.

I have to say, I completely love the idea of being able to refactor without fear.

On a conference call the other day, I think it was Jamie Krug, that brought up the quote, "What is 'Legacy' code -- code that doesn't have unit tests." This is on the top of my list of things to learn.

Ben Nadel

Jul 6, 2012 at 4:25 PM

14,280 Comments

@Sean,

I did it - I finally did it ! I ran a unit test!

www.bennadel.com/blog/2393-Writing-My-First-Unit-Tests-With-Jasmine-And-RequireJS.htm

Heck yeah!

Hamdi

Mar 28, 2013 at 3:33 PM

1 Comments

I Would like to Thank you for this post, I Was thinking about the Service Layer Like a Hole Concept : A SERVICE, but now, you made me think about the Whole Concept. First, there's a thought that i would like to share concerning the architectural Diagram You made, the diagram in my opinion is mingling the Domain Layer and the Service Layer, you can put the entites and eventualyy the related Values Objects into a layer called Domain Layer and Services in an other Layer that you'll call Service Layer (all the types Of Services I Think). An other Point is that Gateways (or DAO in the JAVA community) can only be used by Domain Services wich will do all the CRUDs and then these Domain Services will be called by a more coarse-grained interface and more workflow oriented layer wich is the Service Layer to do the job. Then The Service Layer will not have to deal directly with te Gateways.
The Third Point is that in much simple appications, the Domain Services are the same as the Application Services because the Use Cases will be like ('create that order','give me that order' ect..). But in much complex application where Business Rules are more and more complexe and evolved, this seperation can be made (may be that's why I didn't the difference between these types of layers when I had first seen them)
I really appreciate your job, it was helpful for me to think about so many things that I didn't realize and that I liked to share :)

Ben Nadel

Apr 4, 2013 at 9:57 AM

14,280 Comments

@Hamdi,

I can't believe how hard it is for me to wrap my head around all of this stuff. I think part of the problem is that I don't experiment enough with the code - especially in the last year, work has been crazy busy and I've not had enough R&D time.

But, from what I think you are saying, my current mental model matches yours. The workflow layer coordinates events, but passes off to the domain services to implement the event steps.

Here are some more recent posts where I try to spell out the roles of each layer more explicitly:

www.bennadel.com/blog/2437-Software-Application-Layers-And-Responsibilities.htm

... and this one, which adds more of an example to the above thought process:

www.bennadel.com/blog/2454-Exploring-Sample-Software-Application-Layers-And-Responsibilities.htm

Right now, it's all in my head, but the code I have in production doesn't quite match :(I really need to start playing more with some sort of sample project.

Alexandre Potvin Latreille

Jun 2, 2014 at 4:29 PM

4 Comments

Hi Ben,

I've also been quite interested to learn about application architecture lately and I think that reading a book like Implementing Domain-Driven Design by Vaughn Vernon would really help you to have a deeper understanding of all this complexity.

From what I can see the architecture you are trying to describe and perhaps are looking for is the Hexagonnal Architecture (also known as Ports and Adapters).

In the Hexagonnal Architecture (with DDD in mind), clients uses a Port (HTTP, WebSocket, NamedPipe) to reach out an Adapter (the controller in your diagram) which then adapts the request to be consumed by an Application Service.

The Application Service then invokes an operation on a Domain Service or more often (unless you want an anemic domain-model) on an Aggregate Root (root entity) retrieved through a Repository which is responsible for the persistence.

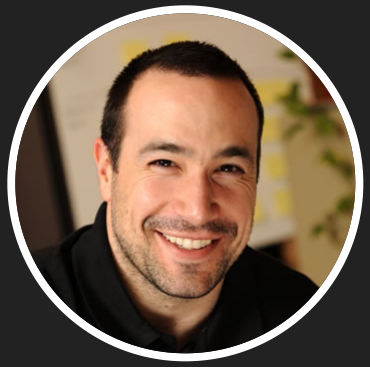
By following the Dependency Inversion Principle, we make sure that the the repository's interface is defined in the Domain layer, but that it's implementation live in the infrastructure layer.

One important thing to note is that the Application Services are NOT part of the Domain Model. They should NOT contain any business logic and are usually transactional.

Hope it helps!

Oh my chickens, this post is old!

Hit me up on Twitter if you want to discuss it further.



About Ben Nadel

I am the co-founder and a principal engineer at InVision App, Inc — the world's leading prototyping, collaboration & workflow platform. I also rock out in JavaScript and ColdFusion 24x7 and I dream about chained Promises resolving asynchronously.

GitHub | Twitter | LinkedIn | Facebook