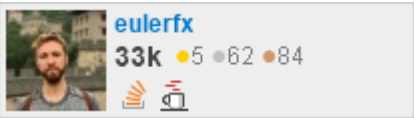
Lev Gorodinski

Computing, Math, Philosophy



Apr 14th, 2012
C#, DDD, architecture

Services in Domain-Driven Design (DDD)

UPDATE: *Vaughn Vernon provided some very valuable insight into the differences between application services and domain services as well as emphasizing the Hexagonal architectural style.*

The term service is overloaded and its meaning takes on different shades depending on the context. As a result, there is a cloud of confusion surrounding the notion of services as one tries to distinguish between application services, domain services, infrastructural services, SOA services, etc. In all these cases the term “service” is valid, however the roles are different and can span all layers of an application.

A service is indeed a somewhat generic title for an application building block because it implies very little. First and foremost, a service implies a client the requests of which the service is designed to satisfy. Another characteristic of a service operation is that of input and output - arguments and provided as input to an operation and a result is returned. Beyond this implication are usually assumptions of statelessness and the idea of pure fabrication according to GRASP.

Eric Evans introduces the notion of a service as a building block within Domain-Driven Design in the blue book:

> When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as standalone interface declared as a SERVICE. Define the interface in terms of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless.
>
> **Eric Evans** *Domain-Driven Design*

These types of services can be identified more specifically as domain services and are part of the domain layer. Domain services are often overlooked as key building blocks, overshadowed by focus on entities and value objects. On the other end of the spectrum is over-utilization of domain services

leading to an anemic domain model in what essentially becomes a separation of data, stored in entities, and behaviors, provided by the service. This can become an anti-pattern because the information expert aspect of OOP is lost.

Domain services are different from infrastructural services because they embed and operate upon domain concepts and are part of the ubiquitous language. Infrastructural services are instead focused encapsulating the "plumbing" requirements of an application; usually IO concerns such as file system access, database access, email, etc. For example, a common application requirement is the sending of an email notification informing interested parties about some event. The concept of the event exists in the domain layer and the domain layer determines when the event should be raised. An email infrastructure service can handle a domain event by generating and transmitting an appropriate email message. Another infrastructural service can handle the same event and send a notification via SMS or other channel. The domain layer doesn't care about the specifics or how an event notification is delivered, it only cares about raising the event. A repository implementation is also an example of an infrastructural service. The interface is declared in the domain layer and is an important aspect of the domain. However, the specifics of the communication with durable storage mechanisms are handled in the infrastructure layer.

An application service has an important and distinguishing role - it provides a hosting environment for the execution of domain logic. As such, it is a convenient point to inject various gateways such as a repository or wrappers for external services. A common problem in applying DDD is when an entity requires access to data in a repository or other gateway in order to carry out a business operation. One solution is to inject repository dependencies directly into the entity, however this is often frowned upon. One reason for this is because it requires the plain-old-(C#, Java, etc…) objects implementing entities to be part of an application dependency graph. Another reason is that is makes reasoning about the behavior of entities more difficult since the Single-Responsibility Principle is violated. A better solution is to have an application service retrieve the information required by an entity, effectively setting up the execution environment, and provide it to the entity.

In addition to being a host, the purpose of an application service is to expose the functionality of the domain to other application layers as an API. This attributes an encapsulating role to the service - the service is an instance of the facade pattern. Exposing objects directly can be cumbersome and lead to leaky abstractions especially if interactions are distributed in nature. In this way, an application service also fulfills a translation role - that of translating between external commands and the underlying domain object model. The importance of this translation must not be neglected. For example, a human requested command can be something like "transfer $5 from account A to account B". There are a number of steps required for a computer to fulfill that command and we would never expect a human to issue a more specific command such as "load an account entity with id A from account repository, load an account entity with id B from account repository, call the debit method on the account A entity…". This is a job best suited for an application service.

The following is an example application service from a purchase order domain. The example also contains a *PurchaseOrder* aggregate, an *Invoice* value object and a repository. (Please note that the code has been simplified for explanation purposes).

```
1       // A repository.
2       public interface IPurchaseOrderRepository
3       {
4           PurchaseOrder Get(string id);
5           // The commit method would likely be moved to a Unit of Work managed by infrastructure.
6           void Commit();
7       }
```

```
8
9       // A marker interface for a domain event.
10      public interface IDomainEvent { }
11
12      // A local domain event publisher.
13      public static class DomainEvents
14      {
15          public static void Raise<TEvent>(TEvent domainEvent) where TEvent : IDomainEvent
16          {
17              // see: http://www.udidahan.com/2009/06/14/domain-events-salvation/
18              // and: http://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/
19          }
20      }
21
22      // The root entity of the PO aggregate - aggregate root.
23      public class PurchaseOrder
24      {
25          public string Id { get; private set; }
26          public string VendorId { get; private set; }
27          public string PONumber { get; private set; }
28          public string Description { get; private set; }
29          public decimal Total { get; private set; }
30          public DateTime SubmissionDate { get; private set; }
31          public ICollection<Invoice> Invoices { get; private set; }
32
33          public decimal InvoiceTotal
34          {
35              get { return this.Invoices.Select(x => x.Amount).Sum(); }
36          }
37
38          public bool IsFullyInvoiced
39          {
40              get { return this.Total <= this.InvoiceTotal; }
```

```
41          }
42
43          bool ContainsInvoice(string vendorInvoiceNumber)
44          {
45              return this.Invoices.Any(x => x.VendorInvoiceNumber.Equals(vendorInvoiceNumber, StringComparison.OrdinalIgnoreCase));
46          }
47
48          public Invoice Invoice(IInvoiceNumberGenerator generator, string vendorInvoiceNumber, DateTime date, decimal amount)
49          {
50              // These guards maintain business integrity of the PO.
51              if (this.IsFullyInvoiced)
52                  throw new Exception("The PO is fully invoiced.");
53              if (ContainsInvoice(vendorInvoiceNumber))
54                  throw new Exception("Duplicate invoice!");
55
56              var invoiceNumber = generator.GenerateInvoiceNumber(this.VendorId, vendorInvoiceNumber, date);
57
58              var invoice = new Invoice(invoiceNumber, vendorInvoiceNumber, date, amount);
59              this.Invoices.Add(invoice);
60              DomainEvents.Raise(new PurchaseOrderInvoicedEvent(this.Id, invoice.InvoiceNumber));
61              return invoice;
62          }
63      }
64
65      // A domain event.
66      public class PurchaseOrderInvoicedEvent : IDomainEvent
67      {
68          public PurchaseOrderInvoicedEvent(string purchaseOrderId, string invoiceNumber)
69          {
70              this.PurchaseOrderId = purchaseOrderId;
71              this.InvoiceNumber = invoiceNumber;
72          }
73
```

```
74          public string PurchaseOrderId { get; private set; }
75          public string InvoiceNumber { get; private set; }
76      }
77
78      // A value object. In production scenarios this would likely be an entity or even an aggregate.
79      public class Invoice
80      {
81          public Invoice(string vendorInvoiceNumber, string invoiceNumber, DateTime date, decimal amount)
82          {
83              this.VendorInvoiceNumber = vendorInvoiceNumber;
84              this.InvoiceNumber = invoiceNumber;
85              this.InvoiceDate = date;
86              this.Amount = amount;
87          }
88
89          // The invoice number provided by the vendor.
90          public string VendorInvoiceNumber { get; private set; }
91          // The internal invoice number is used for internal lookups and is ensured to be unique and readable.
92          public string InvoiceNumber { get; private set; }
93          public DateTime InvoiceDate { get; private set; }
94          public decimal Amount { get; private set; }
95      }
96
97      // A domain service used for generating unique and user-friendly invoice numbers.
98      public interface IInvoiceNumberGenerator
99      {
100         string GenerateInvoiceNumber(string vendorId, string vendorInvoiceNumber, DateTime invoiceDate);
101     }
102
103     // The application service. Can either delegate to a domain model, as in this example, or a transaction script.
104     public class PurchaseOrderService
105     {
106         public PurchaseOrderService(IPurchaseOrderRepository repository, IInvoiceNumberGenerator invoiceNumberGenerator)
```

```
107         {
108             this.repository = repository;
109             this.invoiceNumberGenerator = invoiceNumberGenerator;
110         }
111 
112         readonly IPurchaseOrderRepository repository;
113         readonly IInvoiceNumberGenerator invoiceNumberGenerator;
114 
115         public void Invoice(string purchaseOrderId, string vendorInvoiceNumber, DateTime date, decimal amount)
116         {
117             // Transaction management, along with committing the unit of work can be moved to ambient infrastructure.
118             using (var ts = new TransactionScope())
119             {
120                 var purchaseOrder = this.repository.Get(purchaseOrderId);
121                 if (purchaseOrder == null)
122                     throw new Exception("PO not found!");
123                 purchaseOrder.Invoice(this.invoiceNumberGenerator, vendorInvoiceNumber, date, amount);
124                 this.repository.Commit();
125                 ts.Complete();
126             }
127         }
128     }
```

PurchaseOrder.cs hosted with ♡ by GitHub view raw

There are many refinements that need to be made to this code for it to be of production-ready caliber however it serves well to illustrate the purpose of an application service. The interface *IInvoiceNumberGenerator* is indeed a domain service because it encapsulates domain logic, namely the generation of invoice numbers. This process is something that can be discussed with domain experts and users of the system. After all, the purpose of the generator is to make use of invoice numbers of palatable. By contrast, the *PurchaseOrderService* application service performs technical tasks which domain experts aren't interested in.

The differences between a domain service and an application services are subtle but critical:

- Domain services are very granular where as application services are a facade purposed with providing an API.
- Domain services contain domain logic that can't naturally be placed in an entity or value object whereas application services orchestrate the execution of domain logic and don't themselves implement any domain logic.

- Domain service methods can have other domain elements as operands and return values whereas application services operate upon trivial operands such as identity values and primitive data structures.
- Application services declare dependencies on infrastructural services required to execute domain logic.
- Command handlers are a flavor of application services which focus on handling a single command typically in a CQRS architecture.

In a complete application, a domain model does not stand alone. Applications with GUIs contain a presentation layer which facilitates interaction between the domain and a user. The same application may wish to expose its functionality as a set of REST resources or WCF services or SOA services. In Alistair Cockburn's Hexagonal Architecture, the presentation layer, the REST resource and the WCF service are adapters which adapt the core application to specific ports. Application services form the API which encapsulate the application core and in the case of Domain-Driven Design they ultimately orchestrate and delegate to the underlying entities, value objects and domain services. The application service isn't strictly necessary since each adapter implementation can orchestrate the required domain elements, however encapsulating the domain layer provides a fitting demarcation allowing each component of the entire application to be viewed in isolation. Conversely, the DDD-based domain layer isn't strictly necessary and the application service could delegate to a transaction script. As seen from this perspective, DDD is an implementation detail.

Comments

Tony • 7 years ago

It would've been perfect if you had added a Domain Service in your example, which is very good. Would you reconsider re-edit and add it? Thanks. Tony

• Reply • Share ›

Lev Gorodinski Mod → Tony • 7 years ago

Good idea! I will add an example and update this thread when it is ready.

• Reply • Share ›

Michal → Lev Gorodinski • 7 years ago

Hi Lev,

Do we have any chance for that example?

Kind regards,
Michal

1 • Reply • Share ›

Lev Gorodinski Mod → Michal • 7 years ago • edited

I've made some updates. Let me know if they're helpful.

• Reply • Share ›

Guest • 7 years ago • edited

Having read lots of DDD books, this post is a superb overview. I think you have hit the nail on the head - application services are things that I found not explained very well until now. Described as you have put it, they allow us to mange the critical points of when things are retrieved and saved, a scaffold on which we implement the domain logic, typically one application per related application use-cases/stories.

Another point to mention is if an entity has a repository injected in to it, then we have to worry about saving... that is certainly not the responsibility of the entity, it is the job of the application service which the presentation layer

interfaces with typically as a unit of work.

1 ^ | v • Reply • Share ›

Marcel Wijnands • 7 years ago

Is it true that the PurchaseOrderInvoicedEvent is already raised before the persistance to the database? Because when the persistance fails / throws an exception that could be a problem depending on what handlers of that event are doing. They could for example send an e-mail while the invoice was not persisted to the database.

How would you solve this and still raise the DomainEvent from the domain layer?

^ | v • Reply • Share ›

Lev Gorodinski Mod ➚ Marcel Wijnands • 7 years ago

This is a challenge of implementing domain events in OOP. The usual solution is making sure to make the operation transactional so that it can be rolled back. For emails, it is better to send it async, after the event has been published - http://lostechies.com/jimmy...

^ | v • Reply • Share ›

Sushant ➚ Lev Gorodinski • a year ago

How about invoking the entity method that emits the domain event from the application service after saving the entity to repository ?

^ | v • Reply • Share ›

Abhishek Gupta ➚ Marcel Wijnands • 2 years ago

Since, we don't want to run into issues of distributed transactions, the only way to go is

asynchronous Event Sourcing.

Reply • Share ›

Matthias Noback • 7 years ago

I very much enjoyed this article.

Reply • Share ›

Mark • 7 years ago

Has the example been taken down? I am unable to see it...

Reply • Share ›

andrewgunn • 7 years ago • edited

Lev, great post!

I've been thinking about applying DDD principles to my next project. I like the idea of interacting with entities via methods instead of property setters. In this post, I can see you expose public getters and have private setters. That works until you have a child collection exposed via an ICollection<T> property. In your example, there is a collection of invoices against a purchase order. This property also has a private setter, but because the type is ICollection<Invoice>, it is still possible to add/remove invoices, thus bypassing any of your methods and more importantly, any validation and business logic (e.g. raise an event).

Is this acceptable in the world of DDD? Is there an alternative that exposes an IEnumerable<T> instead of ICollection<T>?

Reply • Share ›

Lev Gorodinski Mod → andrewgunn • 7 years ago

Good point. From what I've seen, this is usually

deemed acceptable simply because the correct way of doing it is a a bit more involved than simply marking a property as readonly. Take a look here: http://gorodinski.com/blog/... where I talk about how to fully encapsulate access to collections. In a nutshell, publicly you expose a read-only collection which wraps a field containing a mutable collection.

▲ | ▼ • Reply • Share ›

andrewgunn ➚ Lev Gorodinski
• 7 years ago

After reading your other post and implementing a simple example, it doesn't work. Entity Framework throws a exception because the collection is read-only.

I tried changing the navigation property to be protected, which did work (almost). EF was able to recognise the navigation property but it couldn't handle eager loading - something that is critical to my application.

I then went back to a public property but prefixed it with an underscore but EF threw another exception because it didn't like the name.

sigh

▲ | ▼ • Reply • Share ›

Lev Gorodinski Mod ➚
andrewgunn • 7 years ago

I'm not familiar with EF specifics, so it may very well be that the NHibernate solution may not apply

NHibernate solution may not apply.

• Reply • Share ›

Garvice Eakins ➚ andrewgunn
• 6 years ago

One of several ways we have overcome this shortcoming in EF is to have a public property on the Entity for use by EntityFramework only, which is not exposed on the interface.
Then having a Public property which exposes a read only collection to fulfill the interface.
Both of which manipulate the private field on the entity itself.

17 • Reply • Share ›

Reynaldo Zabala ➚ Garvice Eakins • 2 years ago

This is resolved in EntityFrameworkCore 2, private fields can work

1 • Reply • Share ›

cfs • 6 years ago

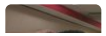
Great article!! One of the best explanations I've read about this subject.

• Reply • Share ›

Rajesh Kumar • 6 years ago

Is PurchaseOrderService is a domain service called from Application service? Can i get a sample code

• Reply • Share ›



Rajesh Kumar • 6 years ago

Can you please provide me a answer
http://stackoverflow.com/qu...

1 • Reply • Share ›

Guest • 6 years ago • edited

Say you had a WCF Service, it could theoretically be the component that implements the coordination logic of using the core application, but, you could also decide to encapsulate such logic in an Application Service, which could then be used by WCF Services and other ports to use the application core. Do I understand this correctly? It's therefore a kind of indirection principle. Also, this means your Application Services are a part of your core?

• Reply • Share ›

Lev Gorodinski Mod ➚ Guest • 6 years ago

Yes. In this way, the WCF service is really just an adapter between you application service and HTTP/TCP or whatever protocol you bind to with WCF. You can think of this in terms of the ports/adapters architecture or onion architecture. And yes, the application service is part of your domain which is the core of your application. You then create adapters between this core and the "outside" world - repositories, HTTP services, UIs, etc.

1 • Reply • Share ›

DOmega • 6 years ago

Wow, what a fantastic read.

Quick question: Would you within any of the different types of services put a fluent API and/or make the service classes accumulate their data via a hash in a common ancestor, rather than simply "returning the data"?

ancestor, rather than simply "returning the data".

Example: https://gist.github.com/atr...

This isn't an approach I espouse, but have seen it in practice and have found it to contravene composition of services, rules of least astonishment and not really presenting any value to the consumer except to obscure desired data.

^ | v • Reply • Share ›

Lev Gorodinski Mod ➚ DOmega • 6 years ago

I wouldn't do that for the same reasons - it adds complexity. However, if you find that your service operations require a significant number of "cross-cutting" concerns such as logging, transactions, etc. it might be worthwhile to use the command handler pattern and exploiting the ICommandHandler<tcommand> abstraction. The complex moving part in this case would be the dispatcher, but it only has to be done once. Better yet, use a functional language like F# where you get these things for free - https://github.com/eulerfx/...

1 ^ | v • Reply • Share ›

DOmega ➚ Lev Gorodinski
• 6 years ago • edited

Another quick question:

Assuming someone who proposed my last example did so because they were under the impression that service calls must somehow return more than one piece of data. Either via a custom container object or tuple of some kind being returned.

Would you consider that a breaking of separation of concerns given that the service layer is now expecting the caller to be prepared to correctly handle more than what it was originally asking for?

Is it not true that when writing services, they should do one thing, and do it well and *only* return what's asked of them? Is there ever a scenario where "out of band" status meta can be sent back from a service method?

^ | v • Reply • Share ›

Lev Gorodinski Mod ➚ DOmega
• 6 years ago

Returning a tuple from a service doesn't imply that its out of band data - it is still one piece of data which happens to be represented by a tuple. The scope of a service, and specifically what it returns is up to the service designer. Status meta-data can be sent in various ways, depending on the underlying infrastructure. For example HTTP supports various ways to provide status metadata via headers, status codes.

1 ^ | v • Reply • Share ›

DOmega ➚ Lev Gorodinski
• 6 years ago

That's awesome advice, thank you! I'm definitely going to take it to heart.

• Reply • Share ›

John Hunter • 6 years ago

Absolutely the best description of this I've seen and uses an excellent example that can be easily translated into real world scenarios. Well done!

• Reply • Share ›

Rafał Łużyński • 5 years ago

Hello, really good overview about service. I finally understood what infrastructure service is (I thought it's a service that calls application service, in example view in a framework). I have a question. I have a model where there is Unit entity and Order VO that can be given to unit and stored within it, so it can "follow it" later. There are various kinds of orders, in example MoveToPositionOrder. Unit has method .followOrders() that is called every step. Problem is that some orders needs data from repository, in particular "MoveToPositionOrder" needs "Stage" AR so it can find the shortest path to the point, how can I pass it there? I suppose I can't, so how can I solve this?

• Reply • Share ›

Thorbs • 5 years ago

Like the fact you drop in a reference to GRASP patterns. Underated and under used IMO.

• Reply • Share ›

Cristian • 5 years ago

Great post it gives lot of insights. Still I have a simple question: Where does PurchaseOrderService.Invoice() being called? There certainly is a sequence between PurchaseOrder.Invoice() -> DomainEvents.Raise() and

the PurchaseOrderService.Invoice(). Thanks!

^ | v • Reply • Share ›

Andrei Gavrila • 5 years ago

Hi Lev and thank you for your clarifications.

Could you please tell us why this:
“load an account entity with id A from account repository,
load an account entity with id B from account repository,
call the debit method on the account A entity…”

is not a domain service that has the responsibility to transfer the money ?
Why is it so clear it is an application service ?

We are implementing business logic here - right ?

^ | v • Reply • Share ›

hkimc ➚ Andrei Gavrila • 4 years ago

+1

^ | v • Reply • Share ›

Constantin Gica ➚ hkimc • 4 years ago

It does not contains any business logic, it is only preparing the actors and the put them to work. I think that the idea is that in the domain layer you have only high level code implementing business rules and we must not clutter it with details like loading something from a repository. I like to think that the domain layer contains only code that a non-programer business specialist would understand.

^ | v • Reply • Share ›

DenisPshenov • 4 years ago