DDD, where to use infrastructure services

Asked 3 years, 5 months ago Active 1 year, 8 months ago Viewed 6k times

6

2

I m trying to learn DDD and I m having some troubles dealing with the Infrastructure services.

In fact, I understand what they aim to, but I don't see where they fit inside of my application.

Actually, I m having the following stuff

- An application Service (REST API)
- Some Domain Services (managed with Specification Pattern)
- Some Infrastructure Services (Repositories, and Email sender)

Question 1 I was wondering if it was the domain service role to use (meaning, infrastructure services injected in Domain services), something like (in a very minimalist way) :

```
  constructor (userRepository, emailService) {
    this.userRepository = userRepository
    this.emailService = emailService
  }

  saveUser (user) {
    /** Apply some business rules here ...*/
    const user = this.userRepository.save(user)
    this.emailService.send('New user added : ' + user.firstName)
  }

}
```

Or should it be the role of the Application service, like this way :

```
class ApplicationService {

  constructor (userDomainService, userRepository, emailService) {
    this.userDomainService = userDomainService
    this.userRepository = userRepository
    this.emailService = emailService
  }

  postUser (user) {
    if (this.userDomainService.manageSomeRules(user)) {
      this.userRepository.save(user)
      this.emailService.send('New user added : ' + user.firstName)
      return user
    }
    return new RuleNotSatisfied()
  }

}
```

Question 2

Who's responsible to convert the received user, from POST endpoints (REST API), to a good entity ?

EDIT on question 2 : What kind of service can convert Entity to DTO and vice versa ? Where should I put them ?

design domain-driven-design

mfrachet
1,151 2 12 20

2 Answers

4

Question 1

It's ok for the domain service to use infrastructure services, if the infrastructure services accept and return domain concepts - entities and value objects.

In addition, domain services should encapsulate some part of the business logic that can't fit in an entity. So `saveUser` is not a great method for a domain service.

The question to ask is - is sending an email an important domain concept? Is `Email` an entity in your domain? If so, you may have an *interface* for an email sender defined in your domain layer, with a *separated implementation* in the infrastructure layer, and you could meaningfully do the following in a domain service - :

```
sendEmailIfWhiteListed(id) {
    Email = this.emailRepository.get(id);
    if (email.isWhiteListed) {
        this.emailSender.send(email);
    }
}
```

But this is only sensible if emails are entities in your domain. If they are not, then think about - why make sending the email the responsibility of saving the user?

In your case, it doesn't look like emails are entities in your *ubiquitous language*. So here I favour the domain events pattern. Creating a new user should raise a domain event `NewUserCreated` , and the handler for the domain event, which I would put in the application layer, would delegate to an email sender to send the email.

Question 2

Who's responsible to convert the post data to a meaningful entity? It depends what you're doing:

    1. If it's simple logic, the application service can call `new Entity()` and pass in meaningful domain arguments (valueobjects) mapped from the api `data` argument
    2. If it's complex logic, a special domain service called a factory can be used `this.entityFactory.Create()` and again, pass in meaningful domain arguments (valueobjects) mapped from the api `data` argument
    3. In both cases, the entity or the factory should not be aware of the structure of `data` as that is an api concern
2. If updating a user
    1. the application service should use a repository to obtain the existing user
    2. the user should have methods for updating relevant data. *But* I'd shy away from CRUD-style methods like `user.update` - think about *why* is the user data being updated? DDD encourages us to think about business processes and objectives.
        1. Is the user being updated because it has participated in some business process? Then model the business process explicitly, and have the user data modified as a side effect of the process
        2. Is the user being updated because of a simple CRUD interaction, like editing a profile page? Even so, I'd encourage an entity method like `user.UpdateProfile` to be explicit about the purpose of the operation.

DDD is ALL about the language - listen to the language used by domain experts, agree on a *ubiquitious* language, and use it faithfully in your code. If your domain experts don't say `persist` or `save` , then don't use those words in your domain layer.

answered Sep 7 '16 at 19:24

Chris Simon
531 4 4

I've editted my second question to be a bit clearer. I was talking about the place where the conversion should take. Who holds the translate function ? Like toDto() or toEntity() :) – mfrachet Jan 24 '17 at 16:43

2

I think the DDD answer is the first one. Put the logic in the domain object. Inject services if required.

I would have another 'hosting layer' to convert the incoming message to the domain object

The application layer would use domain objects to achieve the required effect.

However. I don't think DDD fits the service pattern very well. In my mind the logic often belongs to the service, not the domain object.

Plus the nature of these services is that they are stateless, I play a record, return the result and throw everything away.

Whereas in a desktop app DDD works better. I might have several records instantiated and perform various functions on them over time. Get from shelf, play, pause etc. DDD style code matches the users actions and thought process. 'I play this record' -> `record.Play()`

edited Jun 8 '18 at 8:37

smac89
103 5

answered Sep 7 '16 at 7:31

Ewan
52.2k 3 51 116

Thanks for your answer. I got the difference between the playin record and I agree with you concerning statelessness. My concern is on how my project would be manage, concerning dependency injection. For me, after what you've given me, I would have : Application Service <- Hosting service <- Domain service <- Infrastructure service, with Domain service managing my domain objects . (the arrows means the direction of the injection). Have I got it ? – mfrachet Sep 7 '16 at 9:04

Actually, services are an integral part of DDD. See eg. en.wikipedia.org/wiki/Domain-driven_design#Building_blocks – jhyot Sep 7 '16 at 16:53