Where to call repository.update in DDD?

Asked 7 years, 3 months ago Active 7 years, 3 months ago Viewed 4k times

9

4

I have a real scenario that is a perfect Domain Model design. It is a field that has multiple quadrants with different states on every quadrant. So my aggregate root is the field. Now i have one important question: I want to have a persitant ignorat domain model, which i think makes sense. so where should i call the update on the repository methods? not in the domain model, right? So how should the aggregate root child entities update in the database when there is no change tracking proxy of this objects and the repository should not be called in the entities? Or do i misunderstand the domain model pattern?

is my question clear? :) thank you in advance best laurin

c# domain-driven-design ddd-repositories

asked Jan 10 '13 at 10:51
LaurinSt

792 ● 7 ● 20

4 Answers

Active Oldest Votes

8

> So where should i call the update on the repository methods?

In a stereotypical DDD architecture the repository is usually called by an **application service**. An application service is a class which serves as a facade encapsulating your domain and implements domain uses cases by orchestrating domain objects, repositories and other services.

I'm not familiar with your domain, but suppose there is a use case which shifts a `State` from one `Quadrant` in a `Field` to another. As you stated, the `Field` is the AR. So you'd have a `FieldApplicationService` referencing a `FieldRepository`:

```
public class FieldApplicationService
{
    readonly FieldRepository fieldRepository;

    public void ShiftFieldState(int fieldId, string quadrant, string state)
    {
          // retrieve the Field AR
          var field = this.fieldRepository.Get(fieldId);
          if (field == null)
              throw new Exception();

          // invoke behavior on the Field AR.
          field.ShiftState(quadrant, state);

          // commit changes.
          this.fieldRepository.Update(field);
    }
}
```

The application service is itself very thin. It does not implement any domain logic; it only orchestrates and sets the stage for execution of domain logic which includes accessing the repository. All code dependant of your domain, such as the presentation layer or a service will invoke domain functionality through this application service.

The repository could be implemented in a variety of ways. It can be with an ORM such as NHibernate, in which case change tracking is built in and the usual approach is to commit all changes instead of calling an explicit update. NHibernate provides a Unit of Work as well allowing changes to multiple entities can be committed as one.

In your case, as you stated, there is no change tracking so an explicit call to update is needed and it is up to the repository implementation to handle this. If using SQL Server as the database, the `Update` method on the repository can simply send all properties of a `Field` to a stored procedure which will update the tables as needed.

answered Jan 10 '13 at 17:36
eulerfx
31.8k ●5 ●56 ●77

Question: What if it is *unclear which aggregates were modified*? E.g. application service calls `bookLendingService.LendBookToClient(book, client)`. Save book? Save client? How can we be certain that the application service *knows* and that this never changes? – Timo Mar 5 '18 at 15:20

I have heard of a rule of mutating only one aggregate per transaction, which might be one answer to my question above. If so, what is the reason for this rule? And how does one deal with a scenario that requires changes to multiple aggregates? – Timo Mar 5 '18 at 15:23

5

The Aggregate Root (AR) is updated somwehere. Using a message driven architecture, that somewhere is a command handler, but let's say for general purpose that is a service. THe service gets the AR from a repository, calls the relevant methods then saves the AR back to repository.

The AR doesn't know about the repository, it is not its concern. The Repository then saves all the AR modficications as a unit of work (that is all or nothing). How the Repo does that, well, that depends on how you decided your persistence strategy.

If you're using Event Sourcing, then the AR generates events and the Repo will use those events to persist AR state. If you take a more common approach, that AR should have a state data somewhere exposed as a property perhaps. It's called the memento pattern. The repository persist that data in one commit.

Bu one thing is certain: NEVER think of the persistence details, when dealing with a Domain object. That is don't couple the Domain to an ORM or to some db specific stuff.

answered Jan 10 '13 at 13:26
MikeSW
14.6k ●2 ●30 ●48

Thank you to all of you guys! – LaurinSt Jan 11 '13 at 7:21

The "application code" should call the repository. How the application code is hosted is an infrastructure concern. Some examples of

5

how the application code might be hosted are WCF service, as a Winforms/WPF application, or on a Web server.

The repository implementation is responsible for tracking changes to the aggregate root and its child entities as well as saving them back to the db.

Here is an example:

Domain Project

```
public DomainObject : AggregateRootBase //Implements IAggregateRoot
{
    public void DoSomething() { }
}

public IDomainObjectRepository : IRepository<DomainObject>, IEnumerable
{
    DomainObject this[object id] { get; set; }
    void Add(DomainObject do);
    void Remove(DomainObject do);
    int IndexOf(DomainObject do);
    object IDof(DomainObject do);
    IEnumerator<DomainObject> GetEnumerator();
}
```

Implementation Project

```
public SqlDomainObjectRepository : List<DomainObjectDataModel>, IDomainObjectRepository
{
    //TODO: Implement all of the members for IDomainObjectRepository
}
```

Application Project

```
public class MyApp
{
    IDomainObjectRepository repository = //TODO: Initialize a concrete
SqlDomainObjectRepository that loads what we need.
    DomainObject do = repository[0]; //Get the one (or set) that we're working with.
    do.DoSomething(); //Call some business logic that changes the state of the aggregate
root.
    repository[repository.IDof(do)] = do; //Save the domain object with all changes back
to the db.
}
```

If you need to transactionalize changes to multiple aggregate roots so the changes are made on an all or nothing basis, then you should look into the Unit of Work pattern.

Hope this helps clarify things!

answered Jan 10 '13 at 16:57
Aaron Hawkins
2,120 ● 16 ● 24

Thanks for your valueable answer :) – LaurinSt Jan 11 '13 at 7:22

Generally you should not modify more than one aggregate in one transaction. See dddcommunity.org/wp-content/uploads/files/pdf_articles/…. – Tuukka Haapaniemi Sep 3 '15 at 13:38

1 That's true in general, but I love how the document your link points to lists 4 distinct reasons as to why you may want to do exactly this. – Aaron Hawkins Sep 4 '15 at 20:31

4

My solution is the aggregate root will raise some events to event handlers outside. Those event handlers will call repository to update the database. You will also need a ServiceBus to register and dispatch events. See my example:

```
public class Field: AggregateRoot
{
    public UpdateField()
    {
         // do some business
         // and trigger FieldUpdatedEvent with necessary parameters
         ....
         // you can update some quadrants
         // and trigger QuadrantsUpdatedEvent with necessary parameters
    }
}

public class FieldEventHandlers: EventHandler
{
    void Handle (FieldUpdatedEvent e)
    {
         repository.Update(e.Field);
    }
}

public class QuadrantEventHandlers: EventHandler
{
    void Handle (QuadrantsUpdatedEvent e)
    {
        repository.Update(e.Quadrant);
    }
}
```

answered Jan 10 '13 at 12:54

phnkha
6,587 ● 1 ● 18 ● 28

Thanks for your usefull answer :) – LaurinSt Jan 11 '13 at 7:21