Software Engineering Stack Exchange is a question and answer site for professionals, academics, and students working within the systems development life cycle. It only takes a minute to sign up.

Sign up to join this community

Anybody can ask a question

Anybody can answer

The best answers are voted up and rise to the top

SOFTWARE ENGINEERING

Conceptual mismatch between DDD Application Services and REST API

Asked 5 years, 2 months ago Active 4 years, 5 months ago Viewed 5k times

23

15

I'm trying to design an application that has a complex business domain and a requirement to support a REST API (not strictly REST, but resource-oriented). I have some trouble coming up with a way to expose domain model in a resource-oriented manner.

In DDD, clients of a domain model need to go through procedural 'Application Services' layer to access any business functionality, implemented by Entities and Domain Services. For example there's an application service with two methods to update a User entity:

```
userService.ChangeName(name);
userService.ChangeEmail(email);
```

The API of this Application Service exposes commands (verbs, procedures), not state.

But if we also need to provide a RESTful API for the same application, then there is a User resource model, that looks like this:

```
{
```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

The resource-oriented API exposes *state*, not *commands*. This raises the following concerns:

- each update operation against a REST API can map to one or more Application Service procedure call, depending on what properties are being updated on the resource model
- each update operation looks like atomic to REST API client, but it is not implemented like that. Each Application Service call is designed as a separate transaction. Updating one field on a resource model could change validation rules for other fields. So we need to validate all resource model fields together to ensure that all potential Application Service calls are valid before we start making them. Validating a set of commands at once is much less trivial that doing one at a time. How do we do that on a client that doesn't even know individual commands exist?
- calling Application Service methods in different order might have a different effect, while REST API makes it look like there's no difference (within one resource)

I could come up with more similar issues, but basically they all are caused by the same thing. After each call to an Application Service, state of the system changes. Rules of what is valid change, the set of actions an entity can perform next change. A resource-oriented API tries to makes it all look like an atomic operation. But the complexity of crossing this gap must go somewhere, and it seems huge.

In addition, if the UI is more command-oriented, which often is the case, then we'll have to map between commands and resources on the client side and then back on the API side.

Questions:

1. Should all this complexity just be handled by a (thick) REST-to-AppService mapping layer?
2. Or am I missing something in my understanding of DDD/REST?
3. Could REST simply be not practical for exposing functionality of domain models over a certain (fairly low) degree of complexity?

architecture rest domain-driven-design

edited Apr 17 '15 at 8:40

asked Apr 17 '15 at 4:45
astreltsov
704 4 14

3 I personally don't consider REST that necessary. However it is possible to shoehorn DDD into it: infoq.com/articles/rest-api-on-cqrs programmers.stackexchange.com/questions/242884/… blog.42.nl/articles/rest-and-ddd-incompatible – Den Apr 17 '15 at 12:42

Think of the REST client as a user of the system. They care absolutely nothing about HOW the system performs the actions that it performs. You would no more expect the REST client to know all the different actions on the domain than you would expect a user to. As you say this logic has to go some where, but it would have to go some where in any system, if you weren't using REST you would be just moving it up into the client. Not doing that is precisely the point of REST, the client should know only that it wants to update state and should have no idea how you go about that. – Cormac Mulhall Apr 17 '15 at 14:52

I understand that REST is supposed to help decoupling Clients from the API. Everything is good for the

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

DDD is not a coding technique; it's a design technique. – Robert Harvey Apr 17 '15 at 15:55

2 @astr The simple answer is that resources are not your model, so the design of resource handling code should not effect the design of your model. Resources are an outward facing aspect of the system, where as the model is internal. Think of resources the same way you might think of the UI. A user might click a single button on the UI and a hundred different things happen in the model. Similar to a resource. A client updates a resource (a single PUT statement) and a million different things might happen in the model. It is an anti-pattern to couple your model closely to your resources. – Cormac Mulhall Apr 20 '15 at 14:08

@CormacMulhall I agree that resource should not drive the design of the domain model. But I have a problem with the premise that 'Resources' are not my 'Model'. By model I don't mean a database schema, I mean a domain model (mainly, domain entities). A well-designed domain model reflects the domain in exactly the same terms that users use and care about. A good API is designed with the same approach. So the resources do intersect with domain entities/aggregates, even if they are not exactly the same thing. – astreltsov Apr 20 '15 at 15:14

Your model and your resources have two different goals. The model models the business logic of domain you are simulating. The resources present states to clients in a fashion that is easy to implement a RESTful system (URIs, content types, state updates). There will of course be overlap, but coupling the two closely will end up with a system that is trying to do two different things. The model should not be considered with representing itself as different resources in different states to clients, and the resources should not be limited to how the model is behaving. – Cormac Mulhall Apr 20 '15 at 15:26

Your resources will be represented by content types that should be very well defined and change little. You should not be constantly changing the content types of your resources. On the other hand your model can be quite fluid, changing with the business, but this is hidden from clients. Your resources can be considered to be an external representation of your model if you like, but again coupling the two too closely can produce problems. The resources are simply one view into your model, and the structure of that view should change little. – Cormac Mulhall Apr 20 '15 at 15:29

Well, I know that when our core business changes, the API will pretty much have to change accordingly too, otherwise it becomes obsolete. Anyway, I guess it's pretty irrelevant if both models are the same or not. There is a resource-oriented model that reflects state. There is a procedure-oriented model that implements behavior. The question how to connect them remains. I'll probably edit the question a bit to make more focused. – astreltsov Apr 20 '15 at 15:37

When your business changes what will be updated is the content and the content type of the resources, not the resources themselves. Think of the (theoretical) example of the New York Times front page. The resource "newyorktimes.com/frontpage" resource has not changed in 20 years. What has changed is its content and its content type (html v1 to html v5). But a web browser from 20 years ago can use HTTP to go and still understand how to communicate with this resource (render html v1). This is the power of REST. 30 years from now a client made today should still work with your API. – Cormac Mulhall Apr 22 '15 at 11:29

1 This is a good talk about treating actions in your domain as side effects of REST state changes, keeping your domain and the web separate (fast forward to 25 min for juicy bit) yow.eventer.com/events/1004/talks/1047 – Cormac Mulhall Apr 22 '15 at 11:50

@CormacMulhall nice talk, very funny. I get the reasoning behind HTTP and resources being a different space than the domain, however this seems to conflict with the concept of task-based UI where the user's intent is precisely supposed to be captured at the Presentation level. I also wonder about the discoverability of such a system where a single visible resource can fan out to a multitude of domain operations. – guillaume31 Apr 24 '15 at 12:34

1 I'm also not sure about the whole "user as a robot/state machine" thing. I think we should strive to make our user interfaces much more natural than that... – guillaume31 Apr 24 '15 at 12:40

3 Answers

Active Oldest Votes

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

10

```
/users/1  (contains basic user attributes)
/users/1/email
/users/1/activation
/users/1/address
```

So I've basically split the larger, complex resource into several smaller ones. Each of these contain somewhat cohesive group of attributes of the original resource which is expected to be processed together.

Each operation on these resources is atomic, even though it may be implemented using several service methods - at least in Spring/Java EE it's not a problem to create larger transaction from several methods which were originally intended to have their own transaction (using REQUIRED transaction propagation). You often still need to do extra validation for this special resource, but it's still quite manageable since the attributes are (supposed to be) cohesive.

This is also good for HATEOAS approach, because your more fine-grained resources convey more information on what you can do with them (instead of having this logic on both client and server because it can't be easily represented in resources).

It's not perfect of course - if UIs is not modelled with these resources in mind (especially data-oriented UIs), it can create some problems - e.g. UI presents big form of all attributes of given resources (and its subresources) and allows you to edit them all and save them at once - this creates illusion of atomicity even though client must call several resource operations (which are themselves atomic but the whole sequence is not atomic).
Also, this split of resources is sometimes not easy or obvious. I do this mainly on resources with complex behaviors/life cycles to manage its complexity.

answered Apr 24 '15 at 8:59
qbd
2,428 9 15

That's what I've been thinking as well - create more granular resource representations because they are more convenient for write operations. How do you handle querying of resources when they become so granular? Create read-only de-normalized representations as well? – astreltsov Apr 24 '15 at 9:05

1 No, I don't have read-only de-normalized representations. I use jsonapi.org standard and it has a mechanism to include related resources in the response for given resource. Basically I say "give me User with ID 1 and also include its subresources email and activation". This helps with getting rid of extra REST calls for subresources and it doesn't affect complexity of client dealing with the subresources if you use some good JSON API client library. – qbd Apr 24 '15 at 9:12

So a single GET request on the server translates into one or more actual queries (depending on how many sub-resources are included) which are then combined into a single resource object? – astreltsov Apr 26 '15 at 17:44

What if more than one level of nesting is necessary? – astreltsov Apr 26 '15 at 17:46

Yes, in relational dbs this will probably translate to multiple queries. Arbitrary nesting is supported by JSON API, it is described here: jsonapi.org/format/#fetching-includes – qbd Apr 26 '15 at 18:00

@Den's comment on the original question provides alternative for this. Personally, I prefer that solution. REST and CQRS are more harmony and nature. This answer is good technique if our design doesn't have

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

0

The key issue here is, how is business logic invoked transparently when a REST call is made? This is a problem that is not directly addressed by REST.

I have solved this by creating my own data management layer over a persistence provider such as JPA. Using a meta model with custom annotations, we can invoke the appropriate business logic when the entity state changes. This ensures that irrespective of how the entity state changes the business logic is invoked. It keeps your architecture DRY and also your business logic in one place.

Using the above example, we can invoke a business logic method called validateName when the name field is changed using REST:

```
class User {
      String name;
      String email;

      /**
       * This method will be transparently invoked when the value of name is changed
       * by REST.
       * The XorUpdate annotation becomes effective for PUT/POST actions
       */
      @XorPostChange
      public void validateName() {
        if(name == null) {

          throw new IllegalStateException("Name cannot be set as null");
        }
      }
    }
```

With such a tool at your disposal, all you will need to do is annotate your business logic methods appropriately.

edited May 20 '15 at 12:57

answered May 19 '15 at 23:04

codedabbler

159 6

0

> I have some trouble coming up with a way to expose domain model in a resource-oriented manner.

You shouldn't be exposing the domain model in a resource oriented manner. You should be exposing the application in a resource oriented manner.

> if the UI is more command-oriented, which often is the case, then we'll have to map between commands and resources on the client side and then back on the API side.

Not at all - send the commands to application resources that interface with the domain model.

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Yes, although there is a slightly different way to spell this that may make things simpler; each update operation against a REST api maps to a process which dispatches commands to one or more aggregates.

> each update operation looks like atomic to REST API client, but it is not implemented like that. Each Application Service call is designed as a separate transaction. Updating one field on a resource model could change validation rules for other fields. So we need to validate all resource model fields together to ensure that all potential Application Service calls are valid before we start making them. Validating a set of commands at once is much less trivial that doing one at a time. How do we do that on a client that doesn't even know individual commands exist?

You are chasing the wrong tail here.

Imagine: take REST out of the picture completely. Imagine instead that you were writing a desktop interface for this application. Let's further imagine that you have really good design requirements, and are implementing a task based UI. So the user gets a minimalist interface that is perfectly tuned for the task they are working; the user specifies some inputs then hits the "VERB!" button.

What happens now? From the perspective of the user, this is a single atomic task to be done. From the perspective of the domainModel, it's a number of commands being run by aggregates, where each command is run in a separate transaction. Those are completely incompatible! We need something in the middle to bridge the gap!

The something is "the application".

On the happy path, the application receives some DTO, and parses that object to get a message that it understands, and uses the data in the message to create well formed commands for one or more aggregates. The application will make sure each of the commands it dispatches to the aggregates are well formed (that's the anti-corruption layer at work), and it will load the aggregates, and save the aggregates if the transaction completes successfully. The aggregate will decide for itself if the command is valid, given its current state.

Possible outcomes - the commands all run successfully - the anti-corruption layer rejects the message - some of the commands run successfully, but then one of the aggregates complains, and you've got a contingency to mitigate.

Now, imagine that you have that application built; how do you interact with it in a RESTful way?

1. The client begins with a hypermedia description of the its current state (ie: the task based UI), including hypermedia controls.
2. The client dispatches a representation of the task (ie: the DTO) to the resource.
3. The resource parses the incoming HTTP request, grabs the representation, and hands it off to the application.

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

- the application successfully updated all of the aggregates: the resource reports success to the client, directing it to a new application state
- the anti-corruption layer rejects the message: the resource reports a 4xx error to the client (probably Bad Request), possibly passing along a description of the problem encountered.
- the application updates some aggregates: the resource reports to the client that the command was accepted, and directs the client to a resource that will provide a representation of the progress of the command.

Accepted is the usual cop-out when the application is going to defer processing a message until after responding to the client - commonly used when accepting an asynchronous command. But it also works well for this case, where an operation that is supposed to be atomic needs mitigation.

In this idiom, the resource represents the task itself - you start a new instance of the task by posting the appropriate representation to the task resource, and that resource interfaces with the application and directs you to the next application state.

In ddd, pretty much any time you are coordinating multiple commands, you want to be thinking in terms of a process (aka business process, aka saga).

There's a similar conceptual mismatch in the read model. Again, consider the task based interface; if the task requires modifying multiple aggregates, then the UI for preparing the task probably includes data from a number of aggregates. If your resource scheme is 1:1 with aggregates, that's going to be difficult to arrange; instead, provide a resource that returns a representation of the data from several aggregates, along with a hypermedia control that maps the "start task" relation to the task endpoint as discussed above.

See also: REST in Practice by Jim Webber.

answered Jan 30 '16 at 4:39
VoiceOfUnreason
23.6k 1 31 57

If we're designing the API to interact with our domain as per our use cases.. Why not to design things in such a way that Sagas are not required at all? Maybe I'm missing something but by reading your response I truly believe REST is not a good match with DDD and it's better to use remote procedures (RPC). DDD is behavior-centric while REST is http-verb centric. Why not to remove REST from the picture and expose the behavior (commands) in the API? After all, probably they were designed to satisfy use cases scenarios and prob are transactional. What's the advantage of REST if we own the UI? – diegosasw Dec 2 '19 at 8:41

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.