# Rest API and DDD

Asked 4 years, 3 months ago Active 3 months ago Viewed 20k times

48

29

In my project using DDD methodology.

The project has the aggregate(entity) Deal. This aggregate has many of use cases.

For this aggregate I need to create a rest api.

With standard: create and delete no problem.

1) **CreateDealUseCase**(name, price and many another params);

```
POST /rest/{version}/deals/
{
   'name': 'deal123',
   'price': 1234;
   'etc': 'etc'
}
```

2) **DeleteDealUseCase**(id)

```
DELETE /rest/{version}/deals/{id}
```

But what to do with the rest of the use cases?

- HoldDealUseCase(id, reason);
- UnholdDealUseCase(id);
- CompleteDealUseCase(id, and many another params);
- CancelDealUseCase(id, amercement, reason);
- ChangePriceUseCase(id, newPrice, reason);
- ChangeCompletionDateUseCase(id, newDate, amercement, whyChanged);
- etc(total 20 use cases)...

What are the solutions?

1) **Use verbs**:

```
PUT /rest/{version}/deals/{id}/hold
{
   'reason': 'test'
}
```

But! Verbs can not be used in the url(in REST theory).

```
PUT /rest/{version}/deals/{id}/holded
{
   'reason': 'test'
}
```

Personally for me it looks ugly. Maybe I'm wrong?

3) **Use 1 PUT request for all operations:**

```
PUT /rest/{version}/deals/{id}
{
   'action': 'HoldDeal',
   'params': {'reason': 'test'}
}

PUT /rest/{version}/deals/{id}
{
   'action': 'UnholdDeal',
   'params': {}
}
```

It is difficult to handle in the backend. Moreover, it is difficult to document. Since 1 action has many different variants of requests, from which is already dependent on specific responses.

All solutions have significant drawbacks.

I have read many articles about the REST on the internet. Everywhere only a theory, how to be here with my specific problem?

api rest domain-driven-design

asked Feb 29 '16 at 13:05
stalxed
923 1 8 11

1 I don't want to state the following as an answer so perhaps others can give their opinion in the event that it is a terrible idea. How about: `/rest/{version}/dealsheld/` , `/rest/{version}/dealscompleted/{id}` , etc. Since one would need to know which state you are dealing with in any event. Would a scheme such as that make sense? – Eben Roux Mar 1 '16 at 4:31

## 4 Answers

Active Oldest Votes

38

> I have read many articles about the REST on the internet.

Based on what I see here, you really need to watch at least one of Jim Webber's talks on REST and DDD

> But what to do with the rest of the use cases?

Ignore the API for a moment - how would you do it with HTML forms?

You'd presumably have a web page the presents a representation of `Deal`, with a bunch of links on it. One link would take you to the `HoldDeal` form, and another link would take you to the `ChangePrice` form, and so on. Each of those forms would have zero or more fields to fill in, and the forms would each post to some resource to update the domain model.

Would they all post to the same resource? Perhaps, perhaps not. They would all have the same media type, so if they were all posting to the same web endpoint, you would have to decode the content on the other side.

Given that approach, how do you implement your system? Well, the media type wants to be json, based on your examples, but there really isn't anything wrong with the rest of it.

> 1) Use verbs:

That's fine.

> But! Verbs can not be used in the url(in REST theory).

Um... no. REST doesn't care about the spelling of your resource identifiers. There's a bunch of URI best practices that claim that verbs are bad - that's true - but that's not something that follows from REST.

But if people are being so fussy, you name the endpoint for the command instead of the verb. (ie: "hold" isn't a verb, it's a use case).

> Use 1 PUT request for all operations:

Honestly, that one isn't bad either. You won't want to share the URI though (because of the way the PUT method is specified), but use a template where the clients can specify a unique identifier.

Here's the meat: you are building an API on top of HTTP and HTTP methods. HTTP is designed for *document transfer*. The client gives you a document, describing a requested change in your domain model, and you apply the change to the domain (or not), and return another document describing the new state.

Borrowing from the CQRS vocabulary for a moment, you are posting commands to update your domain model.

```
PUT /commands/{commandId}
{
   'deal' : dealId
   'action': 'HoldDeal',
   'params': {'reason': 'test'}
```

Justification - you are putting a specific command (a command with a specific Id) into the command queue, which is a collection.

```
PUT /rest/{version}/deals/{dealId}/commands/{commandId}
{
   'action': 'HoldDeal',
   'params': {'reason': 'test'}
}
```

Yeah, that's fine too.

Take another look at RESTBucks. It's a coffee shop protocol, but all of the api is just passing small documents around to advance the state machine.

edited Mar 3 at 12:34

answered Feb 29 '16 at 19:42

VoiceOfUnreason
**31.5k** 3 26 58

10 I seem that you invented the remote procedure calls based on REST. – xfg Jun 19 '17 at 10:17

1 But what if you dont want to build 20 endpoints that would follow the behavior of domain model? 20 endpoint is extremely difficult to maintain. What if you have one endpoint and an additional layer between application and domain layers that would compare and process posted data in order to trigger proper domain behaviour? – mko Oct 9 '17 at 11:45

The complexity exists either way. You either put it in the rest client or in the server. In the server you reduce the places your domain rules are. – JDPeckham May 8 '19 at 12:34

1 it is not REST anymore the way you put it. agree with @xfg – ses Jun 5 at 14:55

---

17

Design your rest api independently of the domain layer.

One of the key concepts of domain driven design is *low coupling* between your different software layers. So, when you design your rest api, you think about the best rest api you could have. Then, it's the role of the application layer to call the domain objects to perform the required use case.

I can't design your rest api for you, because I don't know what you're trying to do, but here are some ideas.

As I understand it, you have a Deal resource. As you said, creation/deletion are easy :

- POST /rest/{version}/deals
- DELETE /rest/{version}/deals/{id}.

Then, you want to "hold" a deal. I don't know what that means, you have to think about what it changes in the resource "Deal". Does it change an attribute? if yes, then you're simply modifying the Deal resource.

PUT /rest/{version}/deals/{id}

```
    holdReason: "something",
    ...
}
```

Does it add something? Can you have several holds on a Deal? Sounds to me that "hold" is a noun. If it's ugly, find a better noun.

POST /rest/{version}/deals/{id}/holds

```
{
    reason: "something"
}
```

another solution: forget REST theory. If you think your api would be clearer, more efficient, simpler with the use of verbs in the url, then by all means, do it. You can probably find a way to avoid it, but if you can't, don't do something ugly just because it's the norm.

Look at twitter's api: lots of developers say that twitter has a well designed API. Tadaa, it uses verbs! Who cares, as long as it's cool and easy to use?

I can't design your api for you, you're the only one who knows your use cases, but i'll say again my two advices:

- Design the rest api by itself, and then use the application layer to call the appropriate domain objects in the right order. That's exactly what the application layer is here for.
- Don't follow the norms and theories blindly. Yes, you should try to follow good practices and norms as much as possible, but if you can't then leave them behind (after careful consideration of course)

answered Feb 29 '16 at 13:31
Kaidjin
**1,233** 1 10 17

3 Domain-driven design is about domain. API clients should be designed with domain in mind too. Otherwise you lose most of benefits of DDD. – Eugene Griaznov Dec 6 '16 at 10:47

3 Yeah but that doesn't mean you should expose all the complexity of your domain to the consumers of the API. The API could expose a subset of the functionalities of the domain layer for example. – Kaidjin Dec 6 '16 at 14:43

You shouldn't expose the logic of command handlers or other internals. But it comes about aggregate commands. It is not all the complexity, it is public shape of the domain. – Eugene Griaznov Dec 6 '16 at 14:58

3

The article Exposing CQRS Through a RESTful API is a detailed approach addressing your problem. You can check the the prototype API. A few comments:

- It is a sophisticated approach, so you likely don't need to implement everything from the article: the Event Sourcing concurrency through HTTP's ETag and If-Match is such an

- It is an opinionated approach: The DDD command type is sent via the media type header, not via the body. Personally, I find it interesting... but not sure to implement this way though

answered Oct 18 '18 at 2:21
Mathieu François
**652** 5 14

0

I separate the use cases (UCs) in 2 groups: commands and queries (CQRS), and I have 2 REST controllers (one for commands and another one for queries). The REST resources doesn't have to be model objects to perform CRUD operations on them as a result of POST/GET/PUT/DELETE. Resources can be whatever object you want. Indeed in DDD you shouldn't expose domain model to the controllers.

**(1) RestApiCommandController:** One method per command use case. The REST resource in the URI is the command class name. The method is always POST, because you create the command, and then you execute it through a command bus (a mediator in my case). The request body is a JSON object that maps the command properties (the args of the UC).

For example: `http://localhost:8181/command/asignTaskCommand/`

```
@RestController
@RequestMapping("/command")
public class RestApiCommandController {

private final Mediator mediator;

@Autowired
public RestApiCommandController (Mediator mediator) {
    this.mediator = mediator;
}

@RequestMapping(value = "/asignTaskCommand/", method = RequestMethod.POST)
public ResponseEntity<?> asignTask ( @RequestBody AsignTaskCommand asignTaskCommand ) {
    this.mediator.execute ( asigTaskCommand );
    return new ResponseEntity ( HttpStatus.OK );
}
```

**(2) RestApiQueryController:** One method per query use case. Here the REST resource in the URI is the DTO object that the query returns (as the element of a collection, or just one alone). The method is always GET, and the params of the query UC are params in the URI.

For example: `http://localhost:8181/query/asignedTask/1`

```
private final Mediator mediator;

@Autowired
public RestApiQueryController (Mediator mediator) {
    this.mediator = mediator;
}

@RequestMapping(value = "/asignedTask/{employeeId}", method = RequestMethod.GET)
public ResponseEntity<List<AsignedTask>> asignedTasksToEmployee (
@PathVariable("employeeId") String employeeId ) {

    AsignedTasksQuery asignedTasksQuery = new AsignedTasksQuery ( employeeId);
    List<AsignedTask> result = mediator.executeQuery ( asignedTasksQuery );
    if ( result==null || result.isEmpty() ) {
        return new ResponseEntity ( HttpStatus.NOT_FOUND );
    }
    return new ResponseEntity<List<AsignedTask>>(result, HttpStatus.OK);
}
```

**NOTE:** Mediator belongs to the DDD application layer. It's the UC boundary, it looks for the command/query, and execute the appropiate application service.

answered Jun 29 '17 at 23:18
choquero70
**2,632** 2 19 34