January 27, 2019 10 min read

Bounded Context in APIs (1/2)

This is an open question: *Does it makes sense to expose the bounded contexts (from DDD) in your REST APIs?*

I think that the answer is a categorical NO. But there are colleagues that disagree so I have tried to first, gather a mountain of evidence and then second, to try to reason about it offering use cases that prove my point.

Let's see the evidence first.

Implementing DDD

The maybe most? second-most after the original book? famous book about DDD dedicates a chapter to integration of DDD and external systems, like REST APIs.

> Tempting though it may be, it is not advisable to directly expose a domain model via RESTful HTTP.
>
> This approach often leads to system interfaces that are more brittle than they need to be, as each change in the domain model is directly reflected in the system interface. There are two alternative approaches for combining DDD and RESTful HTTP.

If* we have defined a domain model in our backend, exposing it in a 1:1 relationship is a bad idea. Let's see the approaches:

> The first approach is to create a separate Bounded Context for the system's interface layer (..). This can be deemed a classic approach, as it views the system's interface as a cohesive whole that is simply exposed using resource abstractions instead of services or remote interfaces.

I like this approach: the domain in the interface is the web page/mobile app you are trying to render by REST calls. You are inside the same bounded context (that could cover several ones in your domain layer). That way you simplify the life of the frontend developer, you minimize REST calls and don't expose directly your domain.

The second approach is...

```
Another approach is appropriate when more emphasis is placed on
standard media types. (...) a domain model can be created to
represent each standard media type. Such a domain model might even
be reused across clients and servers, although some REST and SOA
proponents view this as an anti-pattern.

Note: Such an approach is essentially a Shared Kernel (3) or
Published Language (3) in DDD terms.
```

Coming from a REST background... I don't know what to think about this approach. The book uses the *ical* format as an example, but what if the media type that best represents my use case is only *application/json*? and creating new media types for the interface use cases sounds **very** fishy.

Nevertheless, that should be the end of the discussion. **We shouldn't expose our bounded contexts in our REST APIs**. But let's see more evidence.

Thoughtworks: REST API design resource modeling

This blog post talks a lot about a real use case (that coincidentally applies to my domain): rendering blog posts.

```
It is very important to distinguish between resources in REST API
and domain entities in a domain driven design. Domain driven
design applies to the implementation side of things (including API
implementation) while resources in REST API drive the API design
and contract.

API resource selection should not depend on the underlying domain
implementation details.
```

Quite clear IMHO. Resources != domain entities. This blog post continues talking about several approaches:

```
A blog post API (to create a new blog post entry) can be designed
in two ways.

The first approach is to design multiple APIs - one each for blog
post (title & textual content), picture / attachments, tags on the
content / picture, etc. This approach makes APIs more fine grained
resulting in chattier interactions between the API consumer and
provider. This approach will require API consumers to make
multiple API requests to the server. The server will end up
receiving a significantly higher number of HTTP requests -
possibly impacting its ability to serve multiple API consumers.

The second approach is to design a coarse grained API for posting
a blog (to “Posts” collection resource) that can include post
title, post content, picture and tags. This requires making just
one API request to the API provider reducing the load on the
server.

(...)

If the API consumers are expected to directly manipulate the low
level resources (using fine grained APIs), like CRUD, there will
be two big outcomes: Firstly, the API consumer to API provider
interactions will be very chatty. Secondly, business logic will
start spilling over to the API consumer.
```

So thoughtworks is slightly against chattier approaches. This can be seen from 2 fronts: don’t force frontend users to cross bounded contexts (that will make you do another request call) and think in your frontend use cases to create bounded contexts that cover your interfaces (again, I’m completely in favour of this).

Stack Overflow: Why the domain model should not be used as resources in REST API?

Why the domain model should not be used as resources in REST API?

> When using DDD, the REST API should always be separated from the domain model.
>
> The main reason for this is simplification - you don't want to leak the complexity of the domain model through the API to the clients. Otherwise, clients need to know about the nuances and intricacies of your domain, which most probably makes the API hard to use.

👏👏👏👏👏

> Because the web is a totally different world than your core domain layer. (...) If you want to expose your application via REST, you have to shoehorn your domain processes into HTTP and that usually means making compromises and designing resources that are different from your domain entities.

Again, web world priorities: fewer API calls, linked resources are NOT the same as your domain layer.

More SO: How to clearly define boundaries of a bounded context

Another Stack Overflow question, modeling a country selector:

> If your different bounded contexts understand the meaning/purpose of a country differently, then you need to model it appropriately different in each one.
>
> However, if we are speaking simply of reference data of ISO codes and names, then I believe it's pretty fair and standard to stash it wherever is convenient and make it accessible to all interested parties. For example: a database, a configuration file, a web service, etc.

This sounds like shared kernel to me :)

GOTO 2015 • DDD & REST - Domain Driven APIs for the Web • Oliver Gierke

A **really good talk** that mentions two good points:

- Aggregates (several domain models) are good resource candidates.
- Use Hypermedia :P

A blog post: REST and DDD: incompatible?

Not a very meaty blog post that can be resumed in a one-liner: use hypermedia.

Final words

This should be it, right? **We shouldn't expose our domain layer in our REST API**. That means that the bounded contexts of DDD shouldn't carry over to our REST resources. We should define another and share crossover concepts with a Shared Kernel approach.

This is what I think, focusing on what I think are frontend priorities, as a mobile developer I heavily do NOT want to make several API calls to fill out a screen (and that's one of the main reasons GraphQL is so popular these days).

I'm up for defining the business use cases of each frontend screen/app but, under those pretenses, **we should try to minimize HTTP calls**.

Notes:

- "If* we have defined a domain model in our backend" This is not our case and I think it's very unfortunate. Without doing the exercise of domain modeling you can't call your existing modules "bounded

contexts" and try to transpose them to your APIs. Trying to start to define your domain model from the APIs it's "putting the cart before the horse", IMHO. I vehemently think that: first, you have to define a domain model, then expose it with an API.

REST resources should be aggregates of existing domain models. If you just have entities (and not domain models) and try to shoehorn them into resources you risk defining "REST domain models" that are later discarded when someone defines a real domain layer.

I'm even more against fracturing your existing entities in your resources before the domain model work. If those entities are too heavy, the work of decoupling them should be done in the domain layer (and the API should decide if expose them as it is or aggregated). If you start decoupling them in your API you get the risk of fracturing them differently, annoying your users because it used to be the same "business" concept and increase complexity before it is needed.