17 April 2012

REST and DDD: incompatible?

The past years, we've seen two new terms become popular: REST (REpresentational State Transfer) and DDD (Domain Driven Design). However, where DDD is often used to prevent an Anemic Domain (now considered an anti-pattern), a domain model for REST is often anemic. This blog post explains how we can reconcile these opposites.

REST and DDD: opposites

The year 2000 marks the birth of REST — REpresentational State Transfer. REST is (among other things) a way of looking at the data of a webservice and using a limited set of verbs to perform operations on it. The main concept in a RESTful webservice is that of resources, which are identified by a Uniform Resource Identifier (URI). Starting from a root resource, other resources are discoverable by following the links returned in the resources. This in contrast to a SOAP webservice, which are defined in terms of operations in a WSDL.

Using the 'big' four methods defined in the HTTP standard, GET, PUT, DELETE and POST, the resources of a REST webservice commonly resemble documents. That is, a PUT request overwrites the entire resource, and properties that are not present in the request are removed or reset to null. Specifically, a partial update is often frowned upon (and optionally implemented as a new verb, PATCH).

A few years later (in 2003) DDD was convieved. It's main focus is not on writing applications, but on modelling the domain and the domain logic required to build an application. Usually this is done by modelling the domain logic inside the domain, thus preventing the anti-pattern of an Anemic Domain.

And herein lies the rub: a set of documents is just that: data. It contains no logic. This is the opposite of a domain that contains logic.

Reconciling a resource model with a domain model

Search

Search

Tags

Adobe aggregate query aggregations Agile Airport AJAX Amsterdam Angular AngularJS Apache API Application Applications Architecture AspectJ Atlassian back-end bamboo Based BeanMapper Bibliotheek Bindings Blog posts Bonfire Bridge Browser Browser cache C++ Caching Canvas check CISS Clarion Class Cloudie Clover Code Coding Command line Compilation Component Components Conclusion Conference Config Confluence Constraints Cross Crowd Crucible CSRF CSS csv curl Cycle.js Database dependency Design Developers Development EasyGson EasyMock ebase ebXML Eclipse EDI-omgeving encryption Enterprise architectuur ES6 Eval Expression eXtensible Business Reporting Language Flash Flux Foundations Fragile Framework Framework. HTML5 Frameworks front-end Frontend Github GOTO Grails Greasemonkey Groovy Gson Hackers Hamcrest Hibernate hsql hsql db hsqldb HTML HTML5 IE6 IFRS informatiesysteem Installer Integration integration test integration testing Internet Explorer isolation Izpack JARB Java java owasp zap security proxy passive cookie headers http java spring security java.util.Random Javascript javaswift JCommander JDBC JIRA JMockit JOSS JPA jQuery JSON JVM Language Library License Liquibase maven Microsoft Migration Mock Mockingbird Mockito Mongo MongoDB mvc Ontwerp Open Source OpenStack Optimization Origin owasp performance Persistence Post-MVC PowerMock Prince2 Properties Property path property path traversal QueryDSL Random RDBMS Reactive Programming Redux Refactoring Release Repositories Repository Resource REST RESTful Review rta20 Runtime RxJS Schiphol Scrum Security Sharing SOA SOAP Software Sonar Sourceforge.net Speakeasy Spring Spring Data Spring MVC Spring Security Spring Web Services SQL sql injection sqli Stateless SVG Tables TDD Techniek testing Toplink TPS TypeScript ui-router UX Virtual machine Web Apps Web services Webontwikkeling WebOS Webservices Wicket Wiki Wiztools WOFF Wrapper XBRL XML

Blog Archive

► 2017 (1)

► 2016 (10)

these verb restrictions.

The first step in doing this is to reduce the number of functions to call on the domain model, by designing the domain logic to consist of data only, and not actions. Part of this can be achieved by changing actions like `parcel.shipTo(location)` into a property change: `parcel.setLocation(location)`. And although it is generally not possible to remove all function calls this way, it usually suffices. Especially with a KISS attitude towards application design — with the added benefit that a simple application is easier to build and generally easier to make successful.

This KISS attitude is important, even if it is the most difficult aspect of our profession. It also helps with the final mile: CQRS

A step further: CQRS

When building a domain model "without" business logic, our best help contists of our common sense, and *DDD*. This seems a contradion, but
DDD is generally associated with a technique called Command Query Responsibility Segregation (CQRS). And it is CQRS that can help us remove
the last vestiges of business functions from our model.

At its heart, CQRS is the simple notion of using a separate model to update our domain model, and to query our domain model. The querying part is usually easy: REST excels at this, as the decades of experience with the HTTP protocol (at least the GET requests) have resulted in a plethora of proxies, caches, load balancers, etc. to help us with that. That leaves the update part.

Another advantage of CQRS is that, with a bit of care, the update model can also be used for advanced functionality such as events, change auditing and undo operations. It is especially this advanced functionality that can help us.

In order to use events, undo and such we need to explicitly model changes. This means objects. Objects which we can POST to resources, to add them to their history. Of course we can immediately process and discard these action objects as soon as we recieve them, but that's not really RESTful. A better result from a POST request is the *204 Created* response that we also recieve when we create a new resource.

The net result of such modelling is that we have a model that:

- contains only data,
- accepts only the four HTTP operations GET, PUT, DELETE and POST,
- has an audit trail / history of changes,

► October (3)
► September (2)
► June (2)
► May (4)
▼ April (2)
Making web applications available offline
REST and DDD: incompatible?
► March (1)
► February (3)
► January (2)
► 2011 (43)
► 2010 (19)
► 2007 (10)
► 2004 (2)
► 2001 (2)
► 2000 (1)

So there you have it: a REST Resource Model that is also a Domain Model with its domain logic.

You must be a mathematician...

> That sounded like a conclusion. And altough it seems conceptually ok, this can't be realistically applied, right?

Actually, it can. A prime example (both of the strengths and pitfalls) is accounting. The basics are simple: you need two object types; accounts (with a debit/credit flag and linked to parent accounts) and journal entries. The accounts have a balance as determined by the journal entries and can be displayed on a balance sheet. Technically this is very easy.

But there's more to it, of course. The journal entries I mentioned for example, are only needed because we need to know the history. In fact, the history of the balance is so important, that there are several dozens of educations related to it, ranging from product sellers to registered accountants to fuinalcial modellers. Also around the world the most difficult laws, tax laws, are centered around the history of account balances.

The strengths of this simple model are such, that companies can all adhere to the same tax laws, and still have vastly different business models. In fact, the model is so powerful, that we only need one addition: the ability to (preferably automatically) add transactions whenever a transaction is entered that matches some filter criteria. Examples are VAT and income taxes (and the extra transactions are from and to the tax office).

But accounting is an exceptional case

Acually, no. But because accounting is so important to a company (tax laws), there are always accounts. And due to the fact that for all important tasks people want to get paid, there is always a money trail. It is because our record keeping in other areas of our expertise is so severely lacking, that this money trail is used as a substitute.

We should look upon it as a badge of pride if the money trail needs only be used as an extra, superfluous, method of verification.

requires an awful lot of documentation, especially for the available URI's and in documenting the restrictions on POSTing command objects. We're missing something.

A REST interface is not complete without links. After all, one of the tenets of REST is hyperlinking. This used to be called HATEOAS (Hypertext As The Engine Of Application State), but many simply hate the acronym. But as it essentially means that the resources of a REST interface contain hyperlinks to related resources, we really have a Hyperlink API. The best implementations resemble web pages (but are more easily interpretable by computers): they use a known format, and apart from the initial URI, all URI's are discoverable.

Such a Hyperlink API can reduce development costs by a combination of interpreting metadata of links in a resource (so a client knows what to do with them), and delegating the interpretation of the
semantics of the links to the users. The latter is a powerful mechanism, as it reduces the amount of functionality needed.

What's needed for such a discoverable resource model is a data format that contains links with metadata. Examples are HTML/XHTML and HAL+JSON & HAL+XML.

Conclusion

A REST Resource Model and a DDD Domain Model seem opposites. And in fact, a naive/partial implementation of either is, except in the simplest of cases. But embrace both of them to their fullest extend, and both the REST and DDD aspects they are a very good match in accomplishing everything for even the strictest of requirements.

Posted by Anonymous

1 comment:

Iván 1/6/20 22:27

Thanks for sharing your thoughts. I think that "In order to reconsile this with a DDD Domain Model, we must identify a domain that conforms to these verb restrictions." is not reconciliation. On the contrary, it is subverting DDD, which sits at the core of the application, to REST, which is part of -A- delivery mechanism. Moreover "domain logic to consist of data only" sounds very much like an anemic model, which is something we would like to avoid, not embrace. Cheers.

Reply

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)