Technology

REST API Design - Resource Modeling

Prakash Subramaniam

Published: Aug 14, 2014

"The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time." - Roy Fielding's **dissertation**.

Resources form the nucleus of any REST API design. Resource identifiers (URI), Resource representations, API operations (using various HTTP methods), etc. are all built around the concept of Resources. It is very important to select the right resources and model the resources at the right granularity while designing the REST API so that the API consumers get the desired functionality from the APIs, the APIs behave correctly and the APIs are maintainable.

A resource can be a singleton or a collection. For example, "customers" is a collection resource and "customer" is a singleton resource (in a banking domain). We can identify "customers" collection resource using the URN "/customers". We can identify a single "customer" resource using the URN "/customers/{customerId}".

A resource may "contain" sub-collection resources also. For example, sub-collection

resource “accounts” of a particular “customer” can be identified using the URN “/customers/{customerId}/accounts” (in a banking domain). Similarly, a singleton resource “account” inside the sub-collection resource “accounts” can be identified as follows: “customers/{customerId}/accounts/{accountId}”.

The starting point in selection of resources is to analyze your business domain and extract the nouns that are relevant to your business needs. More importantly, focus should be given to the needs of API consumers and how to make the API relevant and useful from the perspective of API consumer interactions. Once the nouns (resources) have been identified, then the interactions with the API can be modeled as HTTP verbs against these nouns. When they don't map nicely, we could approximate. For example, we can easily use the “nouns in the domain” approach and identify low level resources such as Post, Tag, Comment, etc. in a blogging domain. Similarly, we can identify the nouns Customer, Address, Account, Teller, etc. as resources in a banking domain.

If we take “Account” noun example, “open” (open an account), “close” (close an account), “deposit” (deposit money to an account), “withdraw” (withdraw money from an account), etc. are the verbs. These verbs can be nicely mapped to HTTP verbs. For example, API consumer can “open” an account by creating an instance of “Account” resource using HTTP POST method. Similarly, API consumer can “close” an account by using HTTP DELETE method. API consumer can “withdraw” or “deposit” money using HTTP “PUT” / “PATCH” / “POST” methods. **This** tutorial explains the basic resource selection and naming very well.

However, this simplistic approach may be valid at an abstract level, but breaks down once you hit more complicated domains in practice. Eventually you run into concepts which are not covered by the usual / obvious nouns. The following sections explore the art of resource modeling and the importance of modeling resources with right granularity.

Fine-grained CRUD resources versus Coarse-Grained

Fine grained CRUD resources versus Coarse Grained resources

If we design the API around fine grained resources, we would end up with a chattier API for consumer applications. On the other hand, if we design the API based on very coarse grained resources (designed to do everything) there will not be enough variations to support all the API consumers' needs and the API may become too difficult to use and maintain. To understand this better, let us consider a blog API example:

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.

Conversely, an API consumer application should be able to "like" a blog post by making API request to "Likes" sub-collection resource ("/posts/{post_id}/likes") or add a comment to the blog by making a separate API request to "Comments" sub-collection resource ("/posts/{post_id}/comments") without having to go through the blog "Post (/posts/{post_id}" resource. Instead of these sub-collection resources, using the single coarse grained resource "Post"(/posts/{post_id}" for "liking" or "commenting" would have made the job of both API provider and API consumer difficult. With the single coarse grained "POST" resource approach, to add a comment or to like a blog post, the API provider has to provide an option for the API

consumer to indicate that the API request is meant to add a comment or to like a post - may be by specifying a separate XML element or JSON property in the payload that will indicate the payload type. In the server side, API provider has to look for these hints and decide whether the request is to add a comment or to like a post or to actually update the blog post content, etc. Same goes for updates to the blog post comments. API consumer side code also needs to handle these variations in the payload while using the single coarse grained resource that results in unwanted complexity.

Preventing migration of business logic to API consumer

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. In our blog post API example, the fine grained APIs can leave the blog post data in an inconsistent state and create maintenance problems. For example, the blogging application might have a business logic that says that attaching tags on the content is mandatory or that picture tags can be added only when the post has a picture, etc. To do this correctly, the API consumer needs to make all the required API requests in correct sequence - one request for basic post with content, another request with picture, another request with tags, etc. If the API consumer makes a request to create the blog post but does not make an API request to attach the tags, then the blog post is left with inconsistent data (when tags are mandatory in the context of application). Essentially, this implies that the API consumers need to understand and apply the business logic (such as ensuring tags are attached, ensuring the correct sequence of API requests, etc.) on the API consumer side.

Even if the API consumers understand this responsibility clearly, what happens when there are failures?. For example, the first API request goes through successfully - creates the blog post, but the second request to attach tags has failed.

This leaves the data in an inconsistent state. In this situation, there should be a very clear agreement on what the API consumer is expected to do? Can the API consumer retry? If not, who will clean up the data?, etc. This responsibility is not always well understood and very difficult to enforce. Given that business logic can undergo changes, this approach can increase maintenance efforts - especially when there are different types of API consumers (native mobile, web, etc.) and when API consumers may be unknown / more in numbers (for public APIs).

Essentially, the low level CRUD oriented approach puts the business logic in the client code creating tight coupling between the client (API consumer) and services (API) it shouldn't care about, and it loses the user intent by decomposing it in the client. Anytime the business logic changes, all your API consumers have to change the code and redeploy the system. This is impossible to do in some cases such as native mobile applications because customers will not be interested in frequently updating their mobile applications. Also, providing low level services that support chattier interactions means that API provider will be forced to support all those low level services whenever services are upgraded to maintain backwards compatibility for the API consumers.

In the case of coarse grained APIs, The business logic remains with the API provider side thus reducing the data inconsistency issues discussed earlier. The API consumer may not even know the business logic that gets applied in the server and does not need to know in many cases.

Note: When we talk about preventing business logic migration, we are talking about the control flow business logic (for example, making all the required API requests in correct sequence) and not the functional business logic (for example, tax calculation).

Coarse grained aggregate resources for business processes

How can we reconcile coarse grained interfaces that speak the language of a business capability with HTTP verbs against named resources? What do I do if I don't have enough nouns? What if my service is dealing with multiple (two or more) resources with lots of operations on these resources? How do we ensure coarse grained interactions with many nouns and few verbs? And how do we avoid the low-level, CRUD-like, nature of service interaction, and speak a language more aligned with business terms?

Let us relook at what Roy Fielding's dissertation says about resource: "...any concept that might be the target of an author's hypertext reference must fit within the definition of a resource....". Business capabilities / processes can neatly fit the definition of resources. In other words, for complex business processes spanning multiple resources, we can consider the business process as a resource itself. For example, the process of setting up a new customer in a banking domain can be modeled as a resource. CRUD is just a minimal business process applicable to almost any resource. This allows us to model business processes as true resources which can be tracked in their own right.

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.

Escaping CRUD

The way to escape low-level CRUD is to create business operation or business process resources, or what we can call as "intent" resources that express a business/domain level "state of wanting something" or "state of the process towards the end result". But to do this you need to ensure you identify the true owners of all

your state. In a world of four-verb (AtomPub-style) CRUD, it's as if you allow random external parties to mess around with your resource state, through PUT and DELETE, as if the service were just a low-level database. PUT puts too much internal domain knowledge into the client. The client shouldn't be manipulating internal representation; it should be a source of user intent. To understand this better, let us consider an example where a customer in banking domain wants to change her address. There are two ways this can be done:

1. In the first approach, a CRUD API built around the “Customer” resource can be used to directly update the address of the customer. That is, to update an existing customer’s address, a HTTP PUT request can be made to “Customer” resource (or “Address” resource if it exists). If we go with this CRUD API design, the business meaningful event data such as when was the address changed, who changed it (changed by customer or by the bank staff), what was the change, history of changes, etc. will get missed out. In other words, we will miss out the business relevant event data that may be useful at a later stage. Also, with this approach, the client code needs to have the knowledge of “Customer” domain (including Customer’s attributes, etc.). If the domain definition of “Customer” changes, the client code might require immediate updates even if the client does not use the affected attributes. This makes the client code more brittle.
2. An alternate approach that addresses the CRUD concern is to design the API around the resources that are based on the business processes and domain events. For example, to update an existing bank customer’s address, a POST request can be made to “ChangeOfAddress” resource. This “ChangeOfAddress” resource can capture the complete address change event data (such as who changed it, what was the change, etc.). This approach is especially useful in situations where the business meaningful event data will be useful either for immediate business needs (such as long running asynchronous processes that make the address changes as part of a background batch process) or from the long term perspective (such as analytics or showing historical changes for audit

long term perspective (such as analytics or showing historical changes for audit purposes, etc.). Even if there is no immediate or foreseeable business need to keep the event data, POSTing to the "intent" resource "ChangeOfAddress" can still be considered (with a cost-benefit analysis) to avoid clients from knowing the internal domain knowledge. This keeps the client code less affected by "Customer" domain definition changes. When the event data is not required for business, it is optional to persist "ChangeOfResource" event data. We may choose to directly apply the change of address without storing "ChangeOfResource" event data.

Escaping CRUD means making sure that the service that hosts the resource is the only agent that can directly change its state. This may mean separating resources out into more resources according to who truly owns the particular bit of state. Everyone then just POSTs their 'intents' or publishes the resource states they themselves own, perhaps for polling.

Nouns versus Verbs

The argument over **Nouns and Verbs is endless**. Let us consider an example - setting up a new customer in a bank. This business process can be called either EnrollCustomer, or CustomerEnrollment. In this case, the term CustomerEnrollment sounds better because it is noun-ish. It also reads better: "CustomerEnrollment number 2543 for customer xxxx". It also has the additional benefit of maintaining business relevant, independently query-able and evolving state. The business equivalent of such a resource is a typical form that we may fill out in a business, which triggers a business process. Thinking about the paper form analogy in a typical business function helps us to focus on the business requirements in a technology agnostic way as discussed by Dan North in his article **"A Classic Introduction to SOA"**.

A typical customer enrollment may involve sending a KYC (Know Your Customer) request to an external agency, registering the customer, creating an account,

request to an external agency, registering the customer, creating an account, printing debit cards, sending a mail, etc. These steps may be overlapping and the process would be long-running with several failure points. This is probably a more concrete example where we may model the process as a resource. A process like this will result in creation / updates of multiple low level resources such as Customer, Account, KYCRequest, etc. A GET for such a process will make sense, because we would get back the state of the process currently.

If we don't have the Customer enrollment process modeled as a resource, the API consumer has to then "know" the business logic that a customer enrollment involves - one request to create customer resource, one request for KYC request, one request for print card request, etc. Essentially, all your API consumers will have to understand and apply the business logic in their code. If an API consumer missed out a step such as "print card request", then you have an incomplete enrollment and an unhappy customer because she did not receive the card. This is clearly error prone.

Perhaps this can be a rule of thumb: Does the process need state of its own? Will the business be asking questions about this process such as - what is the status of the process? if it failed, why? Who initiated it and from where? how many of them happened? What are the most common reasons for failure of the process, and at which step? How long did it take on average, min, max? For most non-trivial processes, businesses want answers to these questions. And such a process should be modeled as a resource in its own right.

And this is where the noun-based approach starts getting limiting. Business Processes are of course behavior and the business language often focuses on the verb. But they are also "things" to the business. And given that we can convert most verbs into nouns, the distinction starts becoming blurred. And really it's just how you want to perceive it - any noun can be verbed and vice-versa. The question is what do you want to do with it. You may say things like "enroll Sue" rather than "make an enrollment for Sue", but when talking about a long-running process it

makes sense to say “how is Sue’s enrollment coming along?”. That’s why using a noun for any process that lasts long enough for us to want to know how it’s going looks better.

Reification of abstract concept

Dictionary meaning of Reification is “make (something abstract) more concrete or real”. In other words, reification makes something abstract (e.g. a concept) more concrete/real.

With coarse grained approach focusing on business capabilities, we are modeling many more reified abstract notions as resources. A good example of reified resource is CustomerEnrollment that we discussed previously. Instead of using the Customer resource, we are using a resource which is the equivalent of a request to enroll customer. Let us consider two other examples from the same banking domain:

1. Cash deposit in bank account: Customer deposits money to his/her account. This involves the operations such as applying business rules (example: checking if the deposited amount is within the allowed limit), updating customer’s account balance, adding a transaction entry, sending notifications to customer’s mobile or email, etc. Though we could technically use the Account resource here, a better option would be to reify the business capability / abstract concept called transaction (or money deposit) and create a new resource “Transaction”.
2. Money transfer between two bank accounts: Customer transfers money from one bank account to another bank account. This involves updating two low level resources (“from” account and “to” account), also involves business validations, creation of transaction entry, sending notifications, etc. If the “to” account is in another bank, the transfer might be channeled via a central bank or an external agency. Here the abstract concept is “money transfer”

transaction. To transfer money, we can post to /transactions or /accounts/343/transactions and create a new “Transaction” (or “MoneyTransfer”) resource. It is important to note that creation of new “Transaction” resource does not automatically imply creation of a database table for “Transaction”. API design should be independent of the underlying design concerns on API implementation and data persistence.

In both these cases, rather than using the Account resource, we are using a resource which is the equivalent to a command to deposit money or transfer money - Transaction resource (similar to CustomerEnrollment mentioned previously). This is a good idea especially if this could be a long running process (for example: money transfer might involve multiple stages before it completes). This of course doesn't preclude you from having an Account resource as well - one could be updated as a result of the “Transaction” being processed. Also, there may be genuine use cases for making API requests to “Account” resource. For example, to get the account summary/balance information, the API request should made to “Account” resource.

One of the key switches in thinking is to understand that there is an infinite URI space that you can take advantage of. At the same time, it is good to avoid resource proliferation that may add confusion to the API design. As long as there is a genuine need for the resources with clear user/consumer “intent” that fits well in the overall API design, URI space can be expanded. Reified resources can be used as the transactional boundary for your service.

There’s another aspect to this - the way you organize the server behavior is separate to how the API works. We’ve already discussed having a “Transaction” resource for money deposit, and there are many good reasons for doing so. But it’s also perfectly valid for money deposit to be handled by a post to the Account resource. The service that handles the Account is then responsible for coordinating the changes and create a Transaction resource, Notification resource, etc. (which may be in the same service, or separate services). There’s no reason for the client to have to do all

this itself. API provider needs to pick one service to handle the coordination responsibility. In our example, this responsibility is given to the service handling the Transaction resource, if that's the route to take. This is just the same as in-memory object design. When a client needs to coordinate changes over a bunch of objects a common approach is to pick one to handle the coordination.

REST without PUT and CQRS

HTTP verb **PUT** can be used for idempotent resource updates (or resource creations in some cases) by the API consumer. However, use of PUT for complex state transitions can lead to synchronous cruddy CRUD. It also usually throws away a lot of information that was available at the time the update was triggered - what was the real business domain event that triggered this update? With "**REST without PUT**" technique, the idea is that consumers are forced to post new 'nounified' request resources. As discussed earlier, changing a customer's mailing address is a POST to a new "ChangeOfAddress" resource, not a PUT of a "Customer" resource with a different mailing address field value. The last bit means that we can reduce our API consumers' expectations of atomic consistency - if we POST a "ChangeOfAddress", then GET the referenced Customer, it's clearer that the update may not have been processed yet and the old state may be still there (asynchronous API). GETing the "ChangeOfAddress" resource that was created with "201" response will return details related to the event, and a link to the resources that were updated or that will be updated.

The idea is that we no longer PUT the "new" state of an entity, instead we make our mutations be first class citizen nouns (rather than verbs), and POST them. This also plays very nicely with **event sourcing** - events are a canonical example of first class citizen nouns and help us get out of the mindset of thinking of them as "mutators" - they're domain relevant events, not just a change to the state of some object.

REST without PUT has a side-benefit of separating command and query interfaces

(**CQRS**) and forces consumers to allow for eventual consistency. We POST command entities to one endpoint (the "C" of CQRS) and GET a model entity from another endpoint (the "Q"). To expand this further, think of an API that manages Customers. When we want to view the current state of a Customer, we GET the Customer. When we want to change a Customer, we actually POST a “CustomerChange” resource. When we view the customer via the GET, this may just be a projection of the current state of the Customer built up from the series of change events related to the Customer. Or it could be that we have the “CustomerChange” resources that actually mutate the state of Customer in DB, in which case the GET is a direct DB retrieval. In the latter case, we may be destroying some data (associated with the change events) and potentially losing the intent behind the change (depending whether we choose to persist the event data or not). So REST without PUT doesn't mean we *will* get CQRS automatically, but does make it very easy if we want to.

In summary, PUT puts too much internal domain knowledge into the client as discussed earlier. The client shouldn't be manipulating internal representation; it should be a source of user intent. On the other hand, PUT is easier to implement for many simple situations and has good support in libraries. So, the decision needs to be balanced between simplicity of using PUT versus the relevance of event data for the business.

An example from the public GitHub API

GitHub API is a good example of a reasonably well designed API in the public domain with right resource granularity. For example, **creating a fork** is an asynchronous operation. GitHub supports the reified “Forks” sub-collection resource that can be used to list existing forks or create a new fork. Performing code “merge” using **merges** sub-collection resource is another example of reification of the “merge” concept and the underlying merge operation. On the other hand, GitHub also supports many low level resources such as **Tag**. Most of the real world APIs will require both coarse grained aggregate resources and also low level “nouns in the

domain" resources.

On a closing note

As with everything else, there is no single approach that will work for all the situations. As discussed earlier, there may be situations where an API that is built around low level resources may just be fine. For example, to get bank account balance information, an API that is built around "Account" resource is good enough. On the other hand, if the need is to do a money transfer or to get a bank statement, the API needs to be built around the coarse grained "Transactions" resource. Similarly, there are many situations where using HTTP PUT on low level domain resources may be appropriate and simple. There are also situations where the state transitions are complex and long running or event data is business relevant and worth capturing using HTTP POST on user/consumer "intent" resources. Resource modeling requires a careful consideration based on the business needs, technical considerations (clean design, maintainability, etc.) and cost-benefit analysis of various approaches discussed earlier so that the API design brings out the best API consumer interaction experience.

Acknowledgement

This article significantly borrows from the discussion points of ThoughtWorks employees Charles Haynes, Duncan Cragg, Evan Bottcher, Graham Brooks, James Lewis, Martin Fowler, Peter Gillard-Moss, Samir Seth, Sam Newman, Sarah Hutchins, Srinivasan Raguraman and Tarek Abdelmaguid in internal ThoughtWorks developer group discussions, which the article's author was part of. With additional input from: Jonny Leroy, Sriram Narayan and Tarek Abdelmaguid.