How to abstract away implementation details and simplify your life: A Story

Author: L Wallmark

Claps: 3

Date: Sep 1, 2021

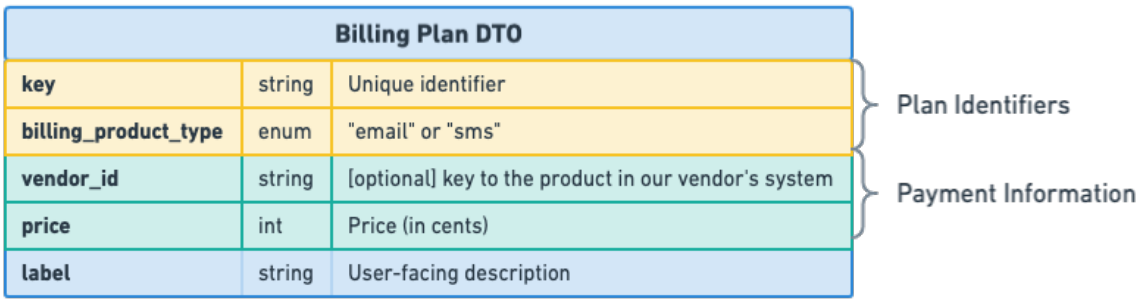
This past spring, Klaviyo started up a subscription model for SMS messaging, to provide flexibility for our SMS customers and to bring our SMS billing offerings into parity with our Email ones. As part of this, several teams across the organization pitched in to update billing features and SMS message sending.

For the Billing team, we were faced with a new paradigm: instead of a single plan for each Klaviyo company, we wanted to support multiple plans, each for a different billing type. This change (as you might imagine) had an impact throughout our code. Suddenly we couldnâ€™t rely on a single price for a single plan for our Klaviyo company objects.

This one-to-many relationship change required a number of considerations. We needed to think about what data would be stored in our databases. We needed to think about how the data would be structured within our 3rd party payment provider. And we needed to think about how this altered set of data would be passed around internally, from system to system, from Klaviyoâ€™s billing service to the code controlling SMS message sending and even to the UI for our Klaviyo customers.

At this point, it became overwhelmingly obvious that there were a lot of moving parts. We needed to start defining some clear data objects and APIs.

The magic answer for us turned out to be data transfer objects. Data transfer objects (DTOs) hold the specific information you need to pass around â€” no more, no less. They have no dependencies and function as little blobs of easy-to-understand data, distilled down to the key details. When you pass around a DTO, you donâ€™t need to worry about matching up function signatures to different parameters. Since the Billing team needed to pull together information from several different sources and then send it around to multiple places, having a set of DTOs defined made everything easier.



Ok, so now we knew what data was needed and where â€” the next step was to define how and when it would be passed from place to place. We started to jot down some function signatures for the API calls we knew we would need. This included billing-specific flows (e.g. change your plan and pay the difference) as well as those that needed the extra information we store within the plan

system (e.g. what are your current plan limits and can we manage them for multiple billing product types).

The important part of this all was to get clear data contracts established, so that we could start parallelizing some of the work. Once the contract was known and documented, other engineers, such as those working directly in the SMS code, could start using it immediately. After all, the Mobile team doesnâ€™t need to know how we end up storing the data with our 3rd party vendor, they just need Billing to provide the right information at the right time. In fact, weâ€™d all prefer other systems not interact with the Billing data stores at all, so that no unexpected dependencies are introduced in systems outside of Billingâ€™s control.

We created contracts going both ways â€” if the SMS code didnâ€™t need to know the ins and outs of Billing, the Billing code also didnâ€™t need to know the ins and outs of how SMS usage was processed. So what goes into a generic usage service? Keep in mind that Klaviyo will now need to support not only two product types, *email* and *SMS*, but also two usage types, *profiles* and *message sends*. Letâ€™s take the following example - this usage service needs to support the following request:

```
service.get_usage_limits(
    company,
    billing_product_type,
    usage_limit_type
)
```

This function would be available (agnostically), but ultimately would need to pull together the following information:

1. Current limits, according to your billing plan (Billing code)
2. Current usage for this month (Product-specific code, either SMS or Email)

In collaboration with the Mobile team, we set the expectation that the SMS code would include a function for each usage type that was supported, for example:

```
service.get_current_cycle_sent_message_count(
    company,
)
```

Knowing this function would be available, the Billing code could incorporate it into the original â€œget_usage_limitsâ€ function.

The decision to separate interface from implementation turned out to be useful. For instance, we needed to move our plan information to a more flexible database table source in order to support SMS plans. However, we needed to keep our old email data source in play for a little while, until we could fully phase it out. This meant our billing plan service needed to be smart enough to support multiple data sources without changing functionality for the callers. We created new code paths to return our brand new Plan DTO, and each data source (old and new) had its own implementation for these functions.

Another happy outcome of this approach was in testability. When writing unit and integration tests, we were able to easily mock different sources of data. We were also able to have very predictable and understandable expected results. As a result, we were able to establish good baseline coverage of our Billing functionality and maintain it as we made changes.

As with anything, our DTOs and APIs were an iterative process. As new surprises showed up, we adjusted fields here and there. We changed how we used the vendor_id column to support custom internal plans. We redefined the ownership lines for the usage service (and got assistance from our friends on the Mobile team to build some of the Billing parts). But any time there was a point of confusion or a question about how things should work, we could refer back to our documented decisions and use that as a jumping off point to move forward.

Weâ€™ve been able to build off of this starting point as well. Since launching the SMS subscription project, the Billing team consolidated to a single data source for billing plans, without interfering with existing API calls. The Mobile team expanded to more geographic areas and updated their usage system to handle the changes, all without impacting any Billing systems. And any time a change is needed in the backend implementation, we have the confidence (and the automated tests) to know we arenâ€™t going to immediately break all the things.