Goldman Sachs | TRANSACTION BANKING

Designing Better APIs for Modern Treasuries

Introduction

Application programming interfaces (APIs) have already dramatically reshaped the user experience in a variety of platforms and industry sectors.

Payment APIs in particular have found widespread use across the consumer and corporate landscape, making it easy to send money, accept a credit card, or get a loan for a big purchase. Corporate treasurers have begun to adapt similar APIs for their own payments, bookkeeping, cash flow, and reporting, among other needs. Further, they are exploring the other benefits of these information-connecting and -sharing hubs for more dramatic uses: automating their treasury function, gaining more accurate views into the future, and enhancing their own internally facing user experience.

As an organization building our own transaction banking product from scratch, we decided that a strong, resilient external API platform would be at the heart of our offering. We wanted to make it easy for a treasurer (or procurement, payment, or fintech professional) to tap directly into our tech with APIs.

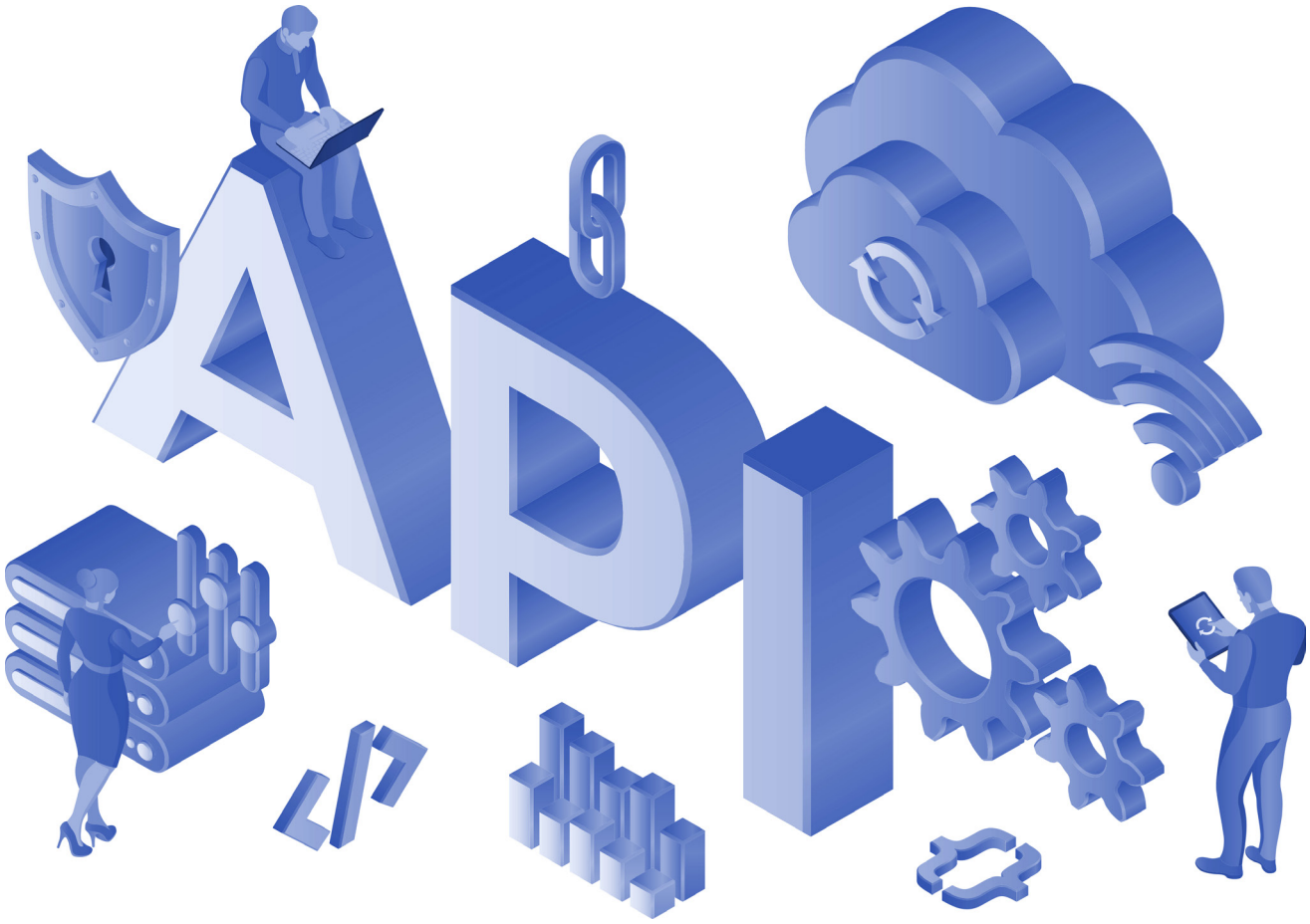
"APIs promise a lot of benefits, and having gone through the process of building them, we have a road map for who needs to be around the table, what they should do, when they should do it, and how the process works when it works best," said Chad Wallace, Global Head of Digital. Our initial position was a strong one—we had the advantage of starting out with a clean technology slate and thus were able to imagine and build APIs without the overhang of legacy systems. As such, our APIs offer strengths and features we wanted to emphasize and that we believe will provide a strong foundation for treasury departments going forward: 24/7 uptime, cloud services, and modern tooling. But this didn't come without significant forethought and design. In fact, the fulcrum of success in building our APIs was the technical platform that lies beneath and the process of polishing each API before rolling it out externally. Building an API is relatively straightforward; building a thoughtful set of APIs on a strong technical platform is challenging and expensive.

Since APIs fundamentally depend on what lies underneath, their design, creation, and development—especially in the context of existing legacy systems—require careful consideration and technical choices. The level of technical complexity can multiply these choices, and APIs are no panacea for that complexity. In fact, APIs are like form-fitting Lycra: They may improve performance and provide the appearance of speed and efficiency, but they ultimately can't hide the fitness of the underlying tech stack, for better or worse.

"The attractiveness of building APIs is undeniable, but getting to smooth-running, relatively glitch-free APIs requires work you don't see as a user—and that's the point. You don't want the user to do the work," said Luc Teboul, Global Head of Engineering. ■

Building an API is relatively straightforward; building a thoughtful set of APIs on a strong technical platform is challenging and expensive.

Reviewing the Technical Choices



The process of sorting through the many considerations and technical choices begins with a simple question: What do you want the API to do?

The answer to that question determines what information it collects, what it connects to, and what happens as a result. Importantly, the user must be involved in answering all these questions. Users and developers, working together, can avoid complexity and choice paralysis, which tends to derail the development of APIs or undermine them once they're rolled out.

The avoidance of complexity can be a guidepost when considering different choices. For example, when designing our payment API, we had to decide whether to allow a client to pay via different payment rails with one or multiple APIs (e.g., one for Fedwire, one for ACH). In the end, we saw that it didn't make sense for a client to have to relearn a new payment API just to access a different rail—so we built a single API for this function. In another case, we created separate payment template APIs for "funding accounts" and "payment accounts" despite their functional similarities. We knew from users they would be utilized in different ways and in different contexts—one creates an account you can pull money from while the other creates an account you can pay money to. ■

Feedback & Refine

The process for reaching these conclusions was deliberate: We would design an API first, then get feedback and refine.

Rather than borrowing an existing internal API (which is often confined to a single system), we tried to design one first that ignored the complexity of internal systems. Then, a structured review of design, following a set of standard questions, forced the initial design to account for some of the most critical technical considerations and challenges likely to emerge later.

Then, users and user input could be integrated into the review process, driving further improvements. By steadily grinding down and polishing the APIs in this manner, we reached insights we would not have recognized in a design-only phase. For example, due to user feedback, we put foreign-exchange details in our payment status API—something we hadn't done initially. We better distinguished funding from payment templates due to their very different sign-up processes. We recognized the need for some changes to our data platform to give easier access to new types of data we had not originally planned to include. These alterations were not initially imagined, but became necessary after we began soliciting client and user feedback.

By building in a feedback phase, we were able to place the user at the center of our design process. In certain cases, we would create personas for our clients, recognize their unique qualities and needs, and seek out approaches that would work for them. For example, we designed our APIs to be as payment rail–agnostic or rail–neutral as possible, so that they could be understood easily by both corporate treasurers and technology companies.

The process of designing and developing APIs revolves around a set of technological issues that can determine user uptake, business impact, complexity, and business execution. ■

Specific Technical Issues

The best-designed APIs follow a set of patterns—naming, response codes, pagination, and so on. The reason is simple: Consistency and patterns make for a better user experience. Consider the challenge of naming, the words used to describe the API.

An inconsistent or disjointed naming system may not appear to make much of a difference at first, but eventually those knots will create confusion and slow the process of scaling up. For example, when creating a single API for all payment rails, the goal of a consistent naming structure forces a designer to confront the fact that "paymentAmount", "Amt," "Amount," and "payment_amount" are all used to describe how much someone is paying someone else. The challenge is further compounded when a particular word has multiple meanings, depending on context. For example, in the domain of foreign-currency exchange, "quantity" can refer to the amount of particular currency. Which is right—the payments version (amount) or the FX version (quantity)? It is best to settle those issues at the outset and answer them firmly.

Our own experience has been shaped by our clean-slate approach. Because we created new APIs without a legacy of naming systems and protocols, we built from the ground up. We created standard,

English descriptions of all of our fields, and, when we were in doubt as to the correct direction on a naming question, we asked professionals in corporate treasury departments: How would they describe it? The answers shaped our nomenclature. This exercise has given us a guidepost to use so that we can add more code without adding more complexity.

Clients want to know the resource types and objects we offer in our system so they can be sure our vision for APIs aligns with their reality. For example, is a payment a single payment, or a collection of payments, over a specific rail? Does it include the "pay from" and the "pay to" or just references to them? Clients want to know how and when they will receive various responses and how their system should react to them. Getting this right is challenging, and we are eager to share our documentation to show what we have learned.

We had the advantage of starting from a relatively small base of business and therefore could build APIs at the center of cash management, including reports, payments, and status tracking. Then, we started making payments for the rest of the bank—a classic "eat your own food" exercise. Then we beta-tested a client-facing API and gauged the response, which helped us prioritize features clients cared about. And because of this sensing mechanism, we were able to see how APIs could evolve and follow a natural life cycle—development, market launch, maturation, and retirement.

This highlighted the critical question of how do we best handle upgrades to the APIs for clients. We recognized that we can't demand that a client upgrade their platform every time we do, so we had to adopt an approach that allowed us to enhance the user experience without dropping clients who had locked in on prior versions of our APIs. The solution became clear through a process of versioning and automated testing so that we know when something will break before others do.

We had to adopt an approach that allowed us to enhance the user experience without dropping clients who had locked in on prior versions of our APIs.

Ultimately, this focus on the user has simplified several other technical issues. For example, consider documentation: An unwritten but valuable rule of graphical user interface is that if you need documentation to use it, the design may be subpar. Layouts, navigation, affordances, icons, fonts, styles, and animations give graphical design inherent clarity. However, in API documentation, there are fewer ways to easily communicate to a user what an API feature does and can do. As a result, we are investing heavily in making our APIs well documented within our design process. ■

Conclusion: The Team Matters

The authority, experience, commitment, and composition of the team designing and developing APIs matters. We have recognized meaningful benefits by creating API ownership across a team of dedicated engineers, product managers, and sales and business stakeholders. Rather than seeing the API as a technical plug-in or product extension, we have treated APIs as a product in their own right and therefore built them like we would build a stand-alone product. That focus, combined with the natural advantages of building off of a brand-new foundation, provided us with the freedom and room to create APIs that deliver what they are supposed to do. ■

We have treated APIs as a product in their own right and therefore built them like we would build a stand-alone product.