API redesign: shopping cart and Stripe payment

ALEX XU
MAY 24, 2023 · PAID

In this issue, we continue with our hands-on exploration of the two remaining API design examples. We'll explore how to build a shopping cart API and study the Stripe payment API redesign.

Example 2 - Shopping Cart

In this section, we design a simple shopping cart.

Step 1 - Analyze the requirements and design APIs

A shopping cart needs to support the following core features:

1. Creating a cart
2. Viewing a cart
3. Adding an item to a cart
4. Viewing items within a cart

We are defining our resources as *cart* and *item*. Below, we've outlined the corresponding APIs:

1. Create a cart
`POST /v1/carts`
2. View a cart
`GET /v1/carts/{cartId}`
3. Add an item to a cart

`POST /v1/carts/mine/items`
`{`
`  "item": {`
`  "sku": "xxxxx",`
`  "qty": 1`
`  }`
`}`

4. View cart items
`GET /v1/carts/mine/items`

Note that we use *mine* as a special cart identifier because a user has only one shopping cart.

When we add an item to a cart, a Google-style API might specify the verb in the URL, like so:

```
POST /v1/carts/mine/items:add
```

There are some who aren't fans of colons in the URLs. In our view, developers should follow their organization's API specifications. When we all speak the same language, we're less prone to errors.

Step 2 - Optimizations

Consider a shopping cart filled with many items. To enhance our query experience, we'll introduce filtering, sorting, and pagination.

1. Filtering

Let's think about querying all red items in our cart. Here's what the API would look like:

```
GET /v1/carts/mine/items?filter=red
```

Note that we need to be cautious about providing broad filtering capabilities. This could potentially impact performance adversely. As a better alternative, we should offer structured filtering which limits the options to certain parameters like color and weight.

For product search pages, structured filtering becomes even more essential. They require additional filters like price, category, location, and more.

2. Sorting

Imagine sorting items in the cart based on when they were added:

```
GET /v1/carts/mine/items?sort_by=time
```

Similar to filtering, our sorting mechanism is also structured. We provide a set of predefined sorting fields for selection.

3. Pagination

When there are more items than a single page can accommodate, we use a technique called *pagination*. This lets users navigate through pages of items.

There are two popular methods for pagination. They are offset pagination and cursor-based pagination.

Offset pagination works by using a page number and the desired count of results per page as parameters. This gives users the ability to see the total pages and the exact page number. An example could look like this:

```
GET /v1/carts?page={page}&count={count}
```

The server uses the page and count parameters to calculate the exact offset and limit to request from the database. Implementing this with a relational database is simple.

For example, with page = 2 and count = 10, a database query might look like this:

```
SELECT * FROM carts
WHERE cart_id = 1234
ORDER BY item_id DESC
LIMIT 10 OFFSET 10;
```

Offset pagination is simple to implement. It gives the user the ability to jump to a specific page in the dataset.

However, offset pagination does not work well for large datasets. Performance can degrade as it requires the database to scan the table up to "offset + limit" rows. As the offset advances further in the dataset, the database still needs to perform wasteful table scans and discard many rows.

Offset pagination also does not work well for datasets that are being written too frequently. A good example is messages in a busy chat group. High data velocity can lead to duplicates or skipped results.

On the other hand, cursor-based pagination uses a pointer to a specific item. It returns results after that pointer in subsequent requests. This method is based on a unique, sequential column in the table, and it offers advantages in scalability and stability over offset pagination. It doesn't require rescanning the dataset up to the offset for each request.

An example of cursor-based pagination could look like this. It uses *maxPageSize* and *pageToken*.

```
GET /v1/carts?maxPageSize={maxPageSize}&pageToken={pageToken}
```

Response:

```
{
    results: [...],
```

```
    nextPageToken={ xxx }
}
```

In the example, *pageToken* is the cursor, and *maxPageSize* defines the maximum results returned in a response.

Cursor-based pagination addresses the drawbacks of offset-based pagination. It scales well for a large dataset. The cursor points to a specific row on a primary column, and the database can use the index to jump to that specific location quickly, without resorting to a table scan. It also stabilizes the pagination window and it works well for fast-changing dataset. However, cursor-based pagination sacrifices the ability to jump to a specific page.

Step 3 - Security

Many shopping carts allow item additions without signing in. This is known as anonymous cart functionality. These public APIs become potential DDoS attack targets. We must guard against attackers adding or removing a large number of items from tens of thousands of PCs, leading to system resource exhaustion.

When designing APIs, it's crucial to employ appropriate rate-limiting algorithms for DDoS attack prevention. This can be implemented at the firewall or API gateway level. For example, firewalls can reject recurrent requests from a single IP address, while API gateway could limit "add to or remove from shopping cart" requests to 100 per minute.

Example 3 - Stripe API Redesign

So far, we have covered two examples. Now, we're going to examine a real-life example. The diagram below shows the development of the Stripe payments API over the past 10+ years [5].

Stage 1 2011: The Stripe API, or what could be described as "7 lines of code", revolved around the *Charge* concept. At this point, it exclusively handled card payments.

Stage 2 2011-2015: the Stripe API introduced the *Token* API. Its goal was to enable its customers to avoid the complex and tedious process of adhering to PCI

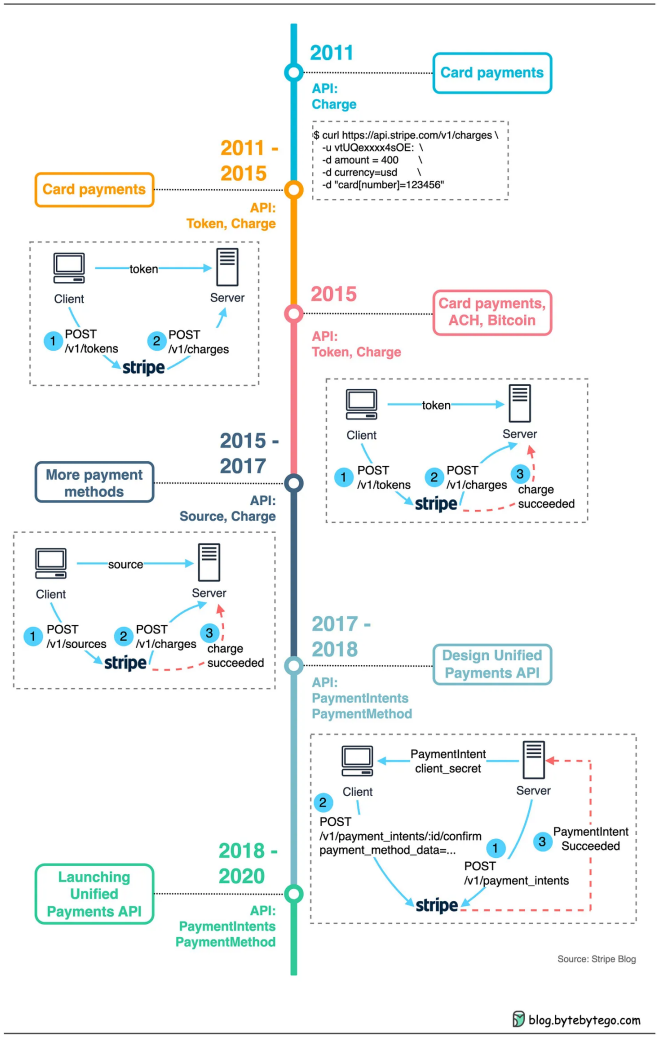
compliance requirements.

Stage 3 2015: ACH and Bitcoin payments entered the scene. Because these types of transactions needed some time to "finalize", Stripe integrated a feedback loop into the API. The change helped indicate the success of the charge.

Stage 4 2015-2017: This stage saw the addition of even more payment methods, including AliPay, WeChat Pay, and iDeal. As a result, the *Source* API was developed as an abstraction to support these varying payment methods.

Stage 5 2017-2018: As the number of supported payment methods grew, the team spent several months drafting a **Unified Payments API**, with the introduction of *PaymentIntents* and *PaymentMethod*.

Stage 6 2018-2020: Stripe invested two years to migrate their clients to the Unified Payments API.



Design Checklist

We've covered a lot in our examples, but there are still a few design principles we haven't touched on yet. Below is a checklist for your reference:

Resource URI:

- *Use plurals*
- *Keep it lowercase*

- *Don't include internal information*
- *Make sure resources are MECE (mutually exclusive, collectively exhaustive)*

Parameters:

- *Keep the number of request and response parameters manageable*
- *Don't expose authentication and authorization details*
- *Avoid confusing abbreviations*
- *Steer clear of illegal parameters*
- *Name parameters consistently*
- *Use proper HTTP status codes*
- *Use standard timestamp formats*

HTTP header:

- *Use cached information properly*
- *Apply expiration if necessary*
- *Add security information when required*

URL:

- *Use versioning*
- *Always use HTTPS*
- *Provide an API index for all available APIs*
- *Prevent unauthorized access*
- *Apply rate-limiting algorithms to resource groups when necessary*

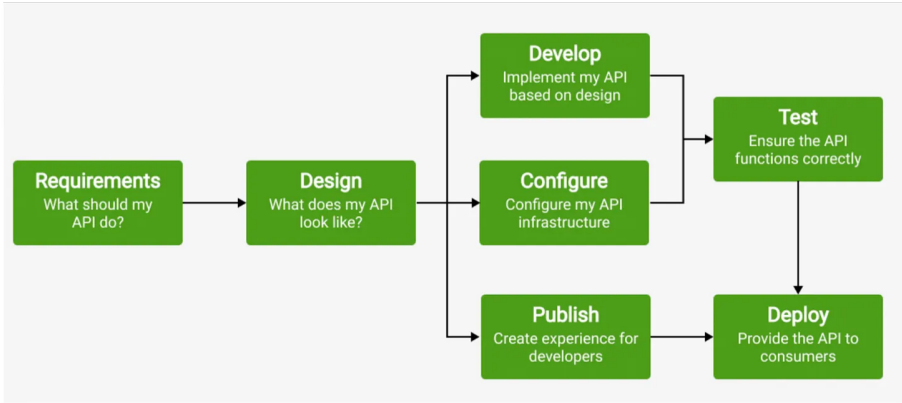
Remember, these are guidelines and might not all apply to every situation. Make sure to use your best judgment.

API Lifecycle Management

We have covered the details of REST API design. But what about managing the entire lifecycle of API development?

Last September, Gartner released their Magic Quadrant for Full Life Cycle API Management [4], highlighting key players like Apigee and MuleSoft as leaders and Postman as a visionary. Have a look at the detailed diagram below.

These robust tools facilitate various stages of our development, from designing and developing APIs to configuring, publishing, testing, and deploying them. In our daily operations, we rely on OpenAPI for defining API specifications. This ensures a shared understanding of specifications across our front-end, back-end, and QA teams. To understand how OpenAPI supports each stage, take a look at the diagram below.



Source: https://www.openapis.org/what-is-openapi

In conclusion, designing APIs holds equal importance to writing code. It calls for an in-depth understanding of business requirements and system interactions. Top-notch APIs foster interoperability and yield network effects within a large organization, similar to Amazon. Whether we are designing new APIs or fine-tuning existing ones, adherence to design guidelines is important. We should carefully consider maintainability. While we might not have the luxury to redesign APIs like Stripe did, it shouldn't stop us from making continual improvements.

We'll wrap up this issue with a noteworthy quote from the Stripe blog:

"A great API product stays out of the developer's way for as long as possible."

117 Likes · 4 Restacks

3 Comments

Write a comment...

Oleksandr May 31, 2023

GET /v1/carts/mine/items?filter=red

What kind of filter is that?