API MODALITY

Like user experiences and machine interfaces, APIs can have modes.

An API endpoint is **MODELESS** when it returns the same result, given equivalent input, against the same dataset, **regardless of state**.

An API endpoint is **MODELESS** when it returns the same result, given equivalent input, against the same dataset, **regardless of state**.

Let's convert that sentence into a behavioral model:

- Given
  - The same input
  - The same dataset
  - And multiple / different states (i.e. a user's role)
- When
  - A request is made to an endpoint
- Then
  - It should always return the same result

WHY IS API MODALITY IMPORTANT?

A modeless API should result in fewer experienced errors by the clients (the code or service consuming the API). It is more flexible than a modal API, so it can serve more purposes.

How could that be?

Modeless APIs reduce the logic necessary for the API to operate. The clients have greater control over their workflow. Modeless APIs improve predictability: we are more likely to receive the responses that we expect for a given endpoint.

In Example

Let's say we have an e-commerce site like Amazon. In that site, we have an object called `Order`, which stores information about a given purchase. Let's assume that Users make Orders, and that Users can belong to Companies, so an Order can be associated with both Users and Companies. Our Order might look something like this:

```
{
    "id": "c7e98dc656b4c13883316fdb",
    "ActorId": "a3dceb32dd2780ae936db946",
    "CompanyId": "d2795b53fd1b35de884d1826",
    "items": []
}
```

For these Orders, we need to provide a way to:

- list all existing orders
- list all of a user's orders
- list all of a company's orders
- list all of a user's orders, for a given company

A **MODAL** Approach

Given a single endpoint, `"/orders"`, if we filter the results by the user's role, our API will have modes based on who the user is:

- If the current user is an app admin, it returns all orders
- If the current user is a company admin, it returns all of the company's orders
- If the current user has no administrative role, it returns all of the current user's orders

This approach has several problems:

- We can't easily predict the results, because the API is making decisions that we know too little about
- An app admin can't list the orders by user or company
- The company admin can't list company orders by user
- Neither of them can list their own orders

A **MODELESS** Approach

To solve this problem with a **modeless** API, we can break it into smaller pieces:

`“/orders"`
- Always returns all existing orders (likely paginated)
- Unless the current user isn’t an app admin. Then it returns a `401`

`"/users/{{user_id}}/orders"`
- Always returns all of a user's orders
- Unless the current user isn't an app admin, and they are not the given user. Then it returns a `401`

`"/companies/{{company_id}}/orders"`
- Always returns all of a company's orders
- Unless the current user isn't an app admin, and they are not an admin for the given company. Then it returns a `401`

`"/companies/{{company_id}}/users/{{user_id}}/orders"`
- Always returns all of a user's orders for a given company
- Unless the current user isn't an app admin, and they are not an admin for the given company, and they are not the given user. Then it returns a `401`

A **MODELESS** Approach
(continued)

The endpoints always return the same result, given equivalent input, against the same dataset, regardless of state. The intent of each endpoint is clear.

These **modeless** endpoints are more flexible than our modal ones. There is a path for an admin and a company admin to see orders by users, including themselves. There is a path for an admin to see orders by company. It also affords a use case where the user's can belong to multiple companies.

The modes that do exist are necessary - we can't expect an application to give us something we should not have access to. We should expect that it will tell us when that is the case.

Shortcuts

When using a RESTful, modeless approach, we may introduce unnecessary complexity in the routes, make the URL harder to navigate, and force clients to perform unnecessary computations. For instance, let's say we have a 32 character identifier for the `user_id`.

That would turn this:
"/users/{{user_id}}/orders"
into this:
"/users/egnp98nu24533onipnjv4t980ubtw0by/orders"

Perhaps that's necessary for an administrator to access another user's orders, but for a user accessing their own orders, this may not be ideal. For this, we can add shortcut resources.

Suppose we declare the resource, `my`. That resource can map to a particular resource: the currently authenticated user. It acts as shorthand for "/users/{{user_id}}".

So we can use this:
"/my/orders"
instead of this:
"/users/egnp98nu24533onipnjv4t980ubtw0by/orders"

An Alternative to Multiple Endpoints

Sometimes it's not desirable to provide multiple endpoints in order to achieve **modelessness**. The criteria for filtering results might be fluid. In the examples in A Modeless Approach, we have four different adaptations of the order endpoint. In reality, we might have forty. In these scenarios, a single endpoint may prove the better solution.

Remember: in our behavioral model for **modeless** endpoints, one of the givens is: "the same input". So, as long as we accept input for the filter criteria, we can achieve **modelessness**.

`"/orders"`

- Always returns all existing orders (likely paginated)
- Unless the current user isn't an app admin. Then it returns a `401`.

`"/orders?user_id={{user_id}}"`

- Always returns all of a user's orders
- Unless the current user isn't an app admin, and they are not the given user. Then it returns a `401`.

`"/orders?company_id={{company_id}}"`

- Always returns all of a company's orders
- Unless the current user isn't an app admin, and they are not an admin for the given company. Then it returns a `401`.

`"/orders?company_id={{company_id}}&user_id={{user_id}}"`

- Always returns all of a user's orders for a given company
- Unless the current user isn't an app admin, and they are not an admin for the given company, and they are not the given user. Then it returns a `401`.

An Alternative to Multiple Endpoints
(continued)

Benefits

- It is **modeless**
- We can support a large number of filters; even ones we don't know about yet

Drawbacks

- It requires more logic on the server
  - e.g. to apply the query string parameters to a model or a query
- It usually requires more effort to achieve security
  - e.g. to make sure we only query on permitted fields, via white or black list
  - e.g. there may be added complexity to avoid query injection
  - Because more effort and code is required, it is harder to understand, which increases risk for bugs and gaps
- It tends to be more error prone
- It doesn't communicate intention as well as the example in A Modeless Approach. For instance, when both a company id and a user id are supplied, will the endpoint return a company's user's orders, or a user's company's orders?

An Alternative to Multiple Endpoints

Query String Security

Query strings are not appropriate for Personally Identifiable Information (PII). Because URLs are stored in web server logs, browser history, and are transferred as Referrer headers, the data in the query string is essentially not secured, even over HTTPS.

In the above example, if we were to replace the `user_id`, with `username`, which is PII, it would not pass a privacy audit. To safeguard PII, we need to use the request body, and `POST` our request.

NOTE: It's confusing to developers when an API expects both query strings and a request body to be present. Always choose one or the other, and always use the request body when the request is not a `GET`.

`POST /orders` with body: `{}`

- Always returns all existing orders (likely paginated)
- Unless the current user isn't an app admin. Then it returns a `401`.

`POST /orders` with body: `{"username":"{{username}}"}`

- Always returns all of a user's orders
- Unless the current user isn't an app admin, and they are not the given user. Then it returns a `401`.

`POST /orders` with body: `{"companyId":"{{companyId}}"}`

- Always returns all of a company's orders
- Unless the current user isn't an app admin, and they are not an admin for the given company. Then it returns a `401`.

`POST /orders` with body: `{"companyId":"{{companyId}}","username":"{{username}}"}`

- Always returns all of a user's orders for a given company
- Unless the current user isn't an app admin, and they are not an admin for the given company, and they are not the given user. Then it returns a `401`.

Modelessness Across Endpoints

Some architects like to take **modelessness** a step further then the example in A Modeless Approach, and provide even more consistency by wrapping all of their responses.

A successful response might have `status`, `payload`, and empty `messages`:

```
{
    "status": 200,
    "payload": [{
        "id": "c7e98dc656b4c13883316fdb",
        "ActorId": "a3dceb32dd2780ae936db946",
        "CompanyId": "d2795b53fd1b35de884d1826",
        "items": []
    }, {
        "id": "b5dad61b9cec3113aac7ea1e",
        "ActorId": "a3dceb32dd2780ae936db946",
        "CompanyId": "d2795b53fd1b35de884d1826",
        "items": []
    }],
    "messages": []
}
```

A fail response might have `status`, an empty `payload`, and `messages`:

```
{
    "status": 401,
    "payload": null,
    "messages": ["Access Denied"]
}
```

Modelessness Across Endpoints
(continued)

The level of consistency across endpoints is an architect's preference. Often times, a wrapped response comes with benefits and drawbacks.

Potential Benefits

- Greater consistency across endpoints
- The domain-specific language and conventions (DSL) for describing success and failure may be more apparent

Potential Drawbacks

- Wrapping responses often results in non-standard, proprietary, opinionated responses. For instance, it's not uncommon for all results to return a 200, unless there is a network failure. This forgoes following HTTP Status Code standards, and nests the outcome in a proprietary request body. Ultimately, that's a poor developer experience.
- The data we really care about is nested under another property called `payload`, `data`, or something. We've seen developers really spin their wheels wondering what's wrong, only to find out that they were trying to access `response.id` and `response.title`, instead of `response.payload.id` and `response.payload.title`.

HTTP Status Codes

HTTP Status Codes are industry standard codes (numbers) which are specified and returned in HTTP responses from servers that support the HTTP protocol. These codes help consumers, such as browsers and API clients, react to the response and to branch their reaction to responses based on the code.

For example, if a request succeeds without error, we expect the server to respond with a **2xx** response, such as `200 OK`. If the request succeeds and results in a new record being added to the database, we expect the server to return: `201 Created`.

In the event that the request doesn't meet the servers expectations, we expect a **4xx** response, such as `400 Bad Request`. If the request body contains invalid values, we expect the server to return `400 Bad Request`. If the request requires that the consumer be authenticated, and it is not, we expect the server to return: `401 Unauthorized`. If the request requires a role or claim that the currently authenticated user doesn't have, we expect the server to return: `403 Forbidden`. And so on...

Often, a response body is also provided which offers more detailed information. For instance, in the event that a record was created and we receive a `201`, we might also expect the ID of the new record to be present in the response body. If our request body had invalid values and we receive a `400`, we might also expect the response body to indicate which values were unacceptable and what is expected.

HTTP Status Codes

in Microservice Architectures

The developer experience is best when the APIs in a microservice architecture are consistent in how they issue HTTP Status Codes, and when they provide consistent response body formats. For instance, it's not difficult to describe most, if not all, **4xx** responses using modlessness across endpoints.

```
Status Code 400
{
    "reason": "INVALID_REQUEST_BODY",
    "messages": ["The Username is required", "You must be 18 to register"]
}
```

```
Status Code 401
{
    "reason": "UNAUTHORIZED",
    "messages": ["Please login to continue"]
}
```

Further Reading

- List of Http Status Codes, Descriptions, and Examples
- Wikipedia List of HTTP Status Codes
- MDN List of Http Status Codes
- IANA HTTP Status Code Registry