Designing Web APIs for Long Term Success

Or: Lessons Learned in Web API Design.

Jeff Valore
Sr. Consultant NimblePros

Examples from 2 companies

B2B Enterprise SaaS vendor.

APIs for several Products converged into 1.

Business critical for clients. No downtime.

External clients often unwilling or unable to change.

Releases scheduled days in advance.

4 name changes, acquired twice in 8 years.

Internal APIs for Microservices.

Build by different teams.

Other teams often unaware of API changes.

Releases can happen as needed.

Document API Conventions (beyond Swagger)

Preferably organization-wide

Drive API consistency across teams and products.

Customers (even internal ones) should know how to use the API.
Support and Professional Services teams should be able to help customers build integrations.

Should define conventions:

- How to use Authentication tokens.
- Common query string parameters.
- Common property names.
- API response format.
- Error handling.

JSON:API Example

8.5 Sorting

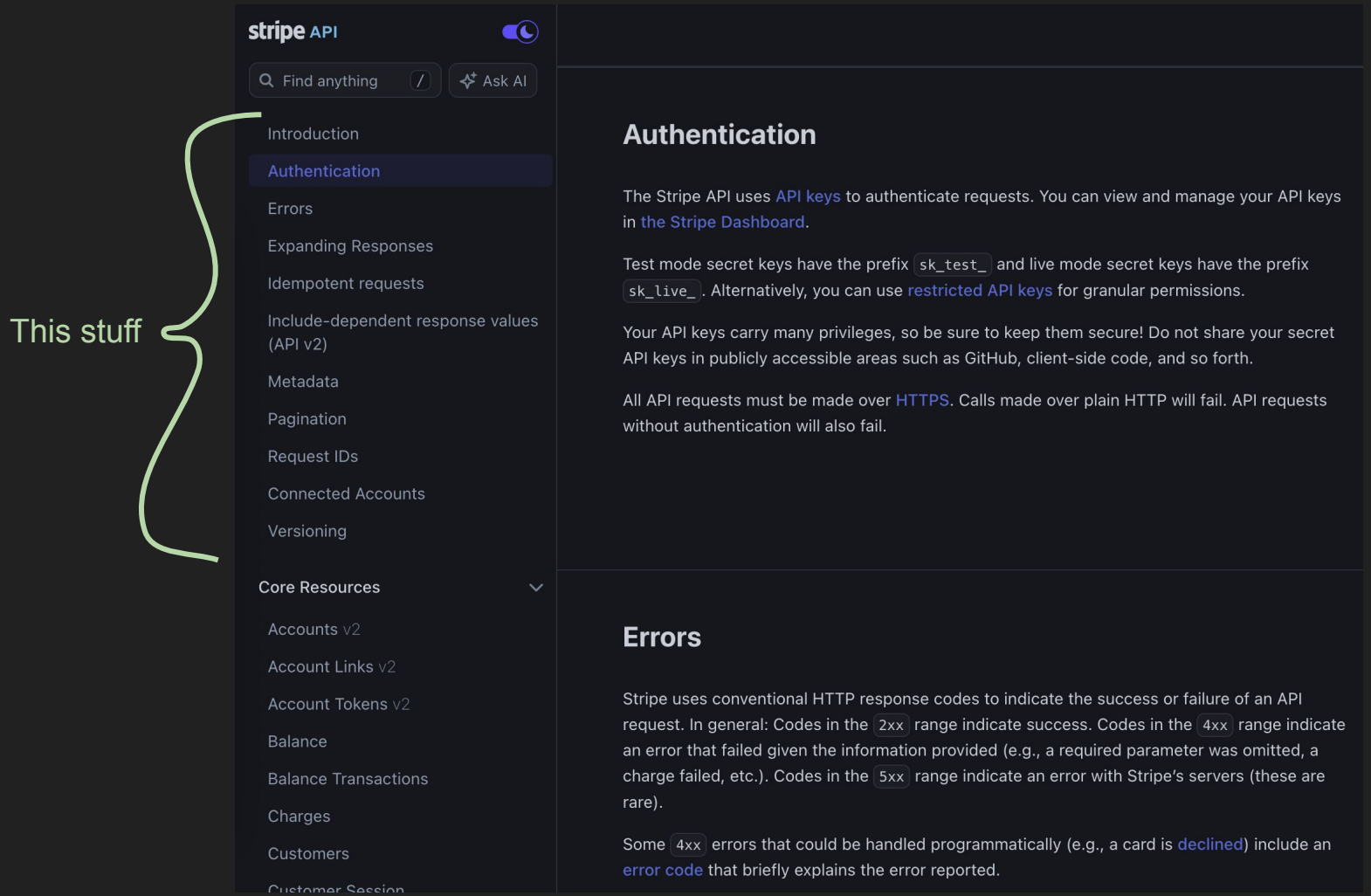
A server **MAY** choose to support requests to sort resource collections according to one or more criteria (“sort fields”).

An endpoint **MAY** support requests to sort the primary data with a sort query parameter. The value for sort **MUST** represent sort fields.

```
GET /people?sort=age HTTP/1.1
Accept: application/vnd.api+json
```

An endpoint **MAY** support multiple sort fields by allowing comma-separated (U+002C COMMA, “,”) sort fields. Sort fields **SHOULD** be applied in the order specified.

```
GET /people?sort=age,name HTTP/1.1
Accept: application/vnd.api+json
```


stripe API
Find anything /
Ask AI
Introduction
Authentication
Errors
Expanding Responses
Idempotent requests
Include-dependent response values (API v2)
Metadata
Pagination
Request IDs
Connected Accounts
Versioning
Core Resources
Accounts v2
Account Links v2
Account Tokens v2
Balance
Balance Transactions
Charges
Customers
Customer Session
This stuff
Authentication
The Stripe API uses API keys to authenticate requests. You can view and manage your API keys in the Stripe Dashboard.
Test mode secret keys have the prefix sk_test_ and live mode secret keys have the prefix sk_live_. Alternatively, you can use restricted API keys for granular permissions.
Your API keys carry many privileges, so be sure to keep them secure! Do not share your secret API keys in publicly accessible areas such as GitHub, client-side code, and so forth.
All API requests must be made over HTTPS. Calls made over plain HTTP will fail. API requests without authentication will also fail.
Errors
Stripe uses conventional HTTP response codes to indicate the success or failure of an API request. In general: Codes in the 2xx range indicate success. Codes in the 4xx range indicate an error that failed given the information provided (e.g., a required parameter was omitted, a charge failed, etc.). Codes in the 5xx range indicate an error with Stripe's servers (these are rare).
Some 4xx errors that could be handled programmatically (e.g., a card is declined) include an error code that briefly explains the error reported.

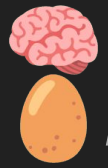
Prior Art

JSON:API - https://jsonapi.org/

OpenAPI - https://spec.openapis.org/oas/v3.1.0

Stripe - https://docs.stripe.com/api

For an easier starting point, extend one of these.



"Unless otherwise specified, refer to JSON:API"

No guidelines

Return an Object, not an Array

```
[
  1, 2, 3, 4, 5
]
```

```
{
  "data": [1, 2, 3, 4, 5],
  "pageNumber": 1,
  "pageSize": 5,
  "total": 10,
  "error": null
}
```

```
curl --request GET --url "https://api.github.com/user/repos"
```

```
[
  {
    "id": 28718607,
    "name": "Code-Workshop",
    …
  }
]
```

```
curl --request GET --url "https://api.github.com/user/repos" -s | jq ".[0].name"
```

```
"Code-Workshop"
```

```
curl --request GET --url "https://api.github.com/user/repos" -s | jq ".[0].name"
jq: error (at <stdin>:5): Cannot index object with number
```

```
curl --request GET --url "https://api.github.com/user/repos" -s | jq ".[0].name"
```

```
jq: error (at <stdin>:5): Cannot index object with number
```

Response for an invalid / expired auth token:


```
{
  "message": "Bad credentials",
  "documentation_url": "https://docs.github.com/rest",
  "status": "401"
}
```

💬 Adding Pagination

GET /api/boards/7/remaining_images?page=2

```
[
  {…},
  {…}
]
```

💬 Adding Pagination

GET /api/boards/7/remaining_images?page=2

```
[
  {...},
  {...}
]
```

could we change `/api/boards/7/remaining_images?page=1&query=&scope=all` to return an object instead of an array, and include the pagination info?
i.e. somehting like

```
{
  total: 327, <-- total of all pages
  page_size: 100, <-- records per page
  data: [...] <-- the same thing it returns now (1 page of results)
}
```

(edited)

then could pre calculate how many pages there are

GraphQL Responses are always objects

Success:

```
{
  "data": {
    "heroes": {
      "name": "R2-D2"
    }
  }
}
```

Error:

```
{
  "errors": [
    {
      "message": "Syntax Error: Unexpected Name \"operation\".",
      "locations": [
        {
          "line": 1,
          "column": 1
        }
      ]
    }
  ]
}
```

JSON:API

7.1 Top Level

A JSON object MUST be at the root of every JSON:API request and response document containing data. This object defines a document's "top level".

A document MUST contain at least one of the following top-level members:

- data: the document's "primary data".
- errors: an array of error objects.
- meta: a meta object that contains non-standard meta-information.
- a member defined by an applied extension.

The members data and errors MUST NOT coexist in the same document.

Return Errors in a consistent way

GitHub Error:

```
{
  "message": "Bad credentials",
  "documentation_url": "https://docs.github.com/rest",
  "status": "401"
}
```

Improved:

```
{
  "errors": [ {
    "message": "Bad credentials",
    "documentation_url": "https://docs.github.com/en/rest/authentication",
    "status": "401"
  } ]
}
```

```
curl -X POST 'https://egg/api' -v /
-H 'Content-Type: application/json' /
-H 'accept: application/json' /
-d '{
  "numberField": "not a number. uhoh…"
}'
```

What should be returned?
400 Bad Request?

```
curl -X POST 'https://egg/api' -v /
-H 'Content-Type: application/json' /
-H 'accept: application/json' /
-d '{
  "numberField": "not a number. uhoh…"
}'

< HTTP/2 500
< server: awselb/2.0
< content-type: test/plain; charset=utf-8
<
Internal Server Error
```

Validation errors were returned in Rails ActiveRecord validator format…

```
class Person < ApplicationRecord
  validates :first_name, presence: true, length: { minimum: 3, maximum: 10 }
end
```

```
{
  "error": {
    "first_name": [
      "Min length is 3",
      "Max length is 10"
    ]
  }
}
```

Be Language Agnostic

Validation errors were returned in Rails ActiveRecord validator format…

```
class Person < ApplicationRecord
  validates :first_name, presence: true, length: { minimum: 3, maximum: 10 }
end
```

```
{
  "error": {
    "first_name": [
      "Min length is 3",
      "Max length is 10"
    ]
  }
}
```

Be Language Agnostic

Validation errors were returned in Rails ActiveRecord validator format…

```
class Person < ApplicationRecord
  validates :first_name, presence: true, length: { minimum: 3, maximum: 10 }
end
```

```
{
  "error": {
    "first_name": [    ← This is a Rails-specific property name. Is that good?
      "Min length is 3",
      "Max length is 10"
    ]
  }
}
```

Rely on your documented conventions, not the conventions of a specific framework.

```
{
  "data": {
     "applicationId": "...",
     "applicationNumber": "...",
     "CallExp": "...",
     "EXP_SUBCODE2": "...",
     "EventType": "...",
     …
  }
}
```

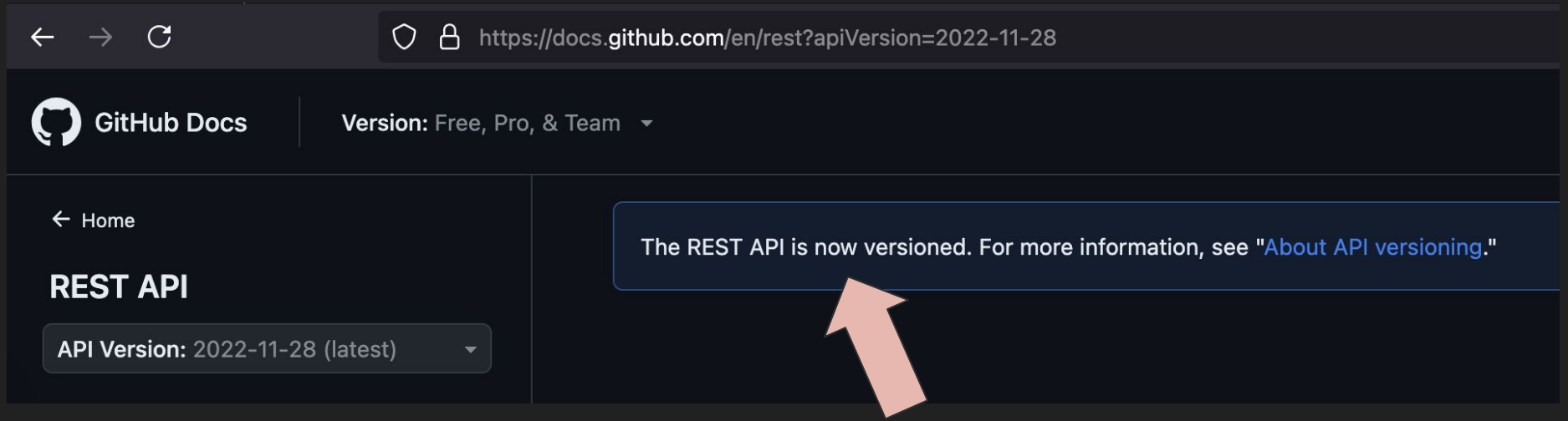
```
{
   "ApplicationNo": "...",
   "AppStep": "...",
   "Status_Attributes": {...},
   "SystemErrors": {...},
   "offers": [...],
   "product": {...},
   …
}
```

Document for Consistency

- Property name casing (camel, snake, title)
- Common property names ("applicationId", "userId", "orderId")
- Data types (is applicationId a string, a number, a guid?)
- Date formats (Numeric ms since epoch? ISO 8601 string?)

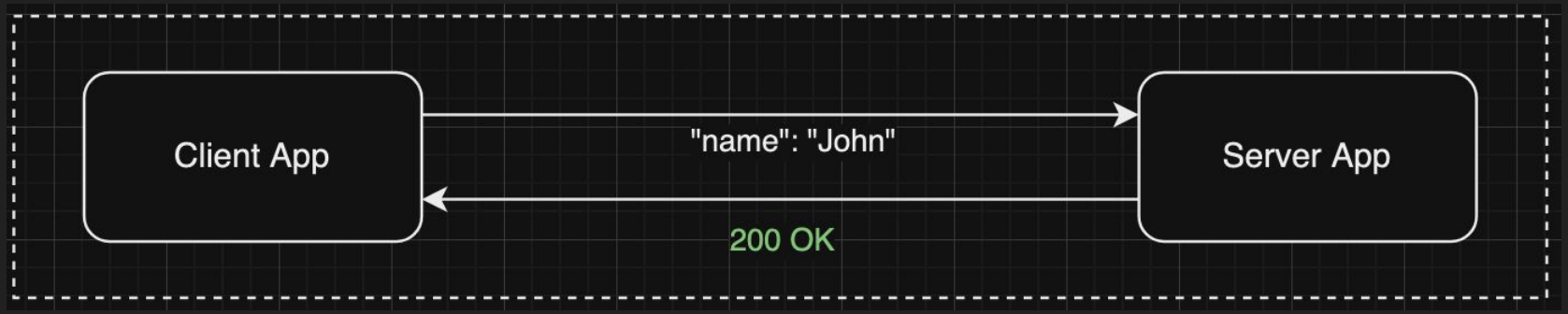
Versioning

You should actually do it. Don't worry, everyone forgets…

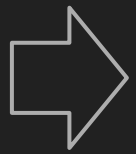


Change Version When Schema Has Breaking Changes

If an API sees extra fields it doesn't know about, it returns HTTP 400 Bad Request.

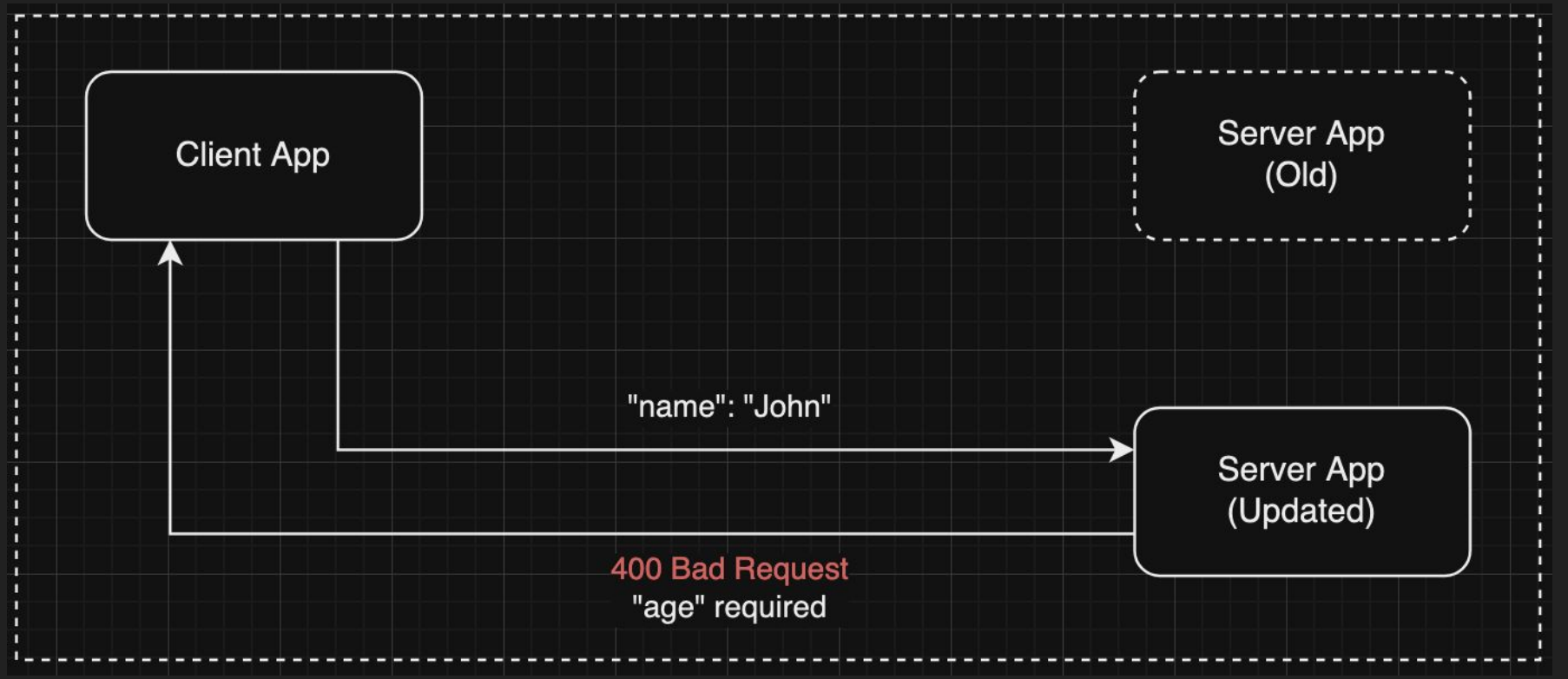



```
{
  "name": "John"
}
```

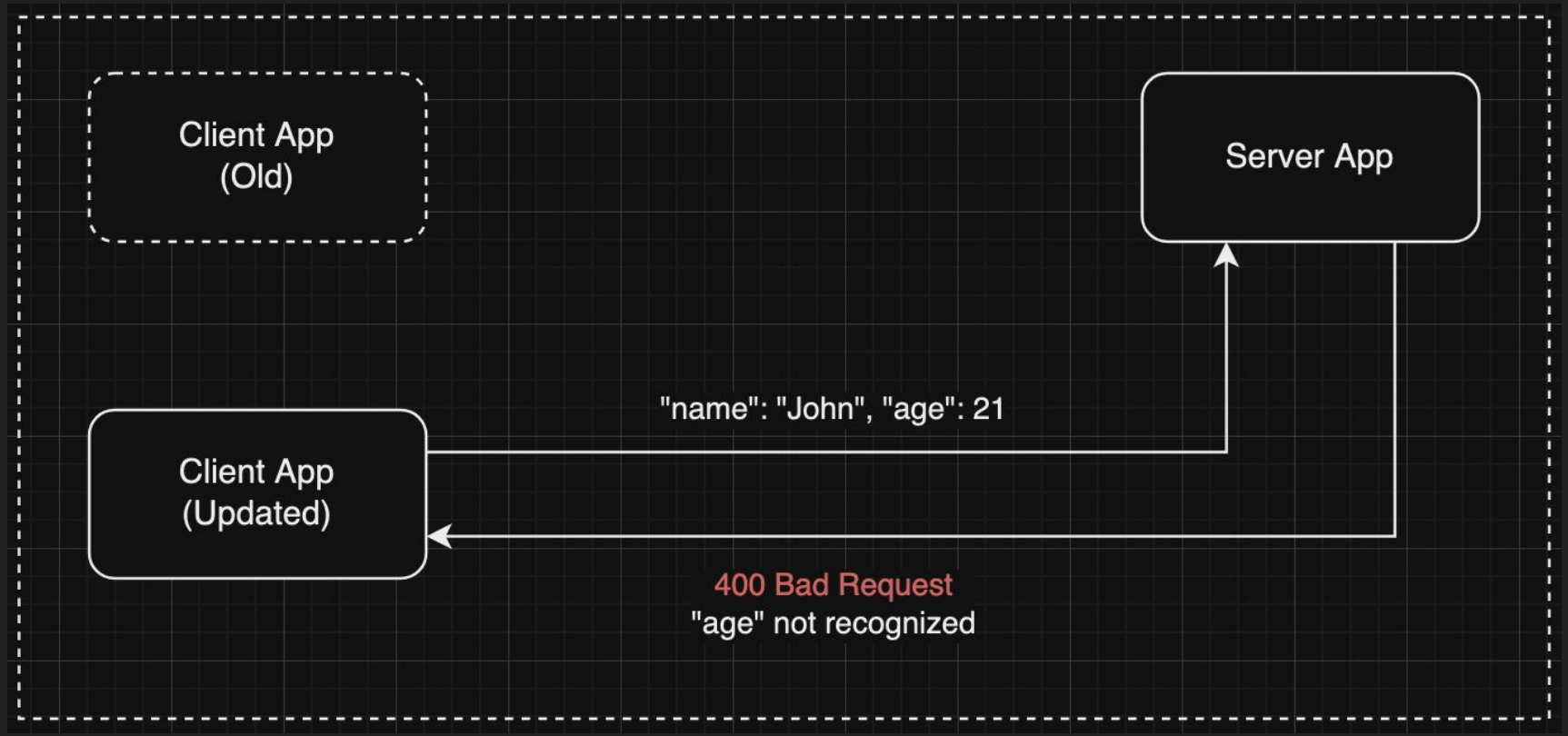


```
{
  "name": "John",
  "age": 21
}
```

Can't deploy Server change first



Can't deploy Client change first

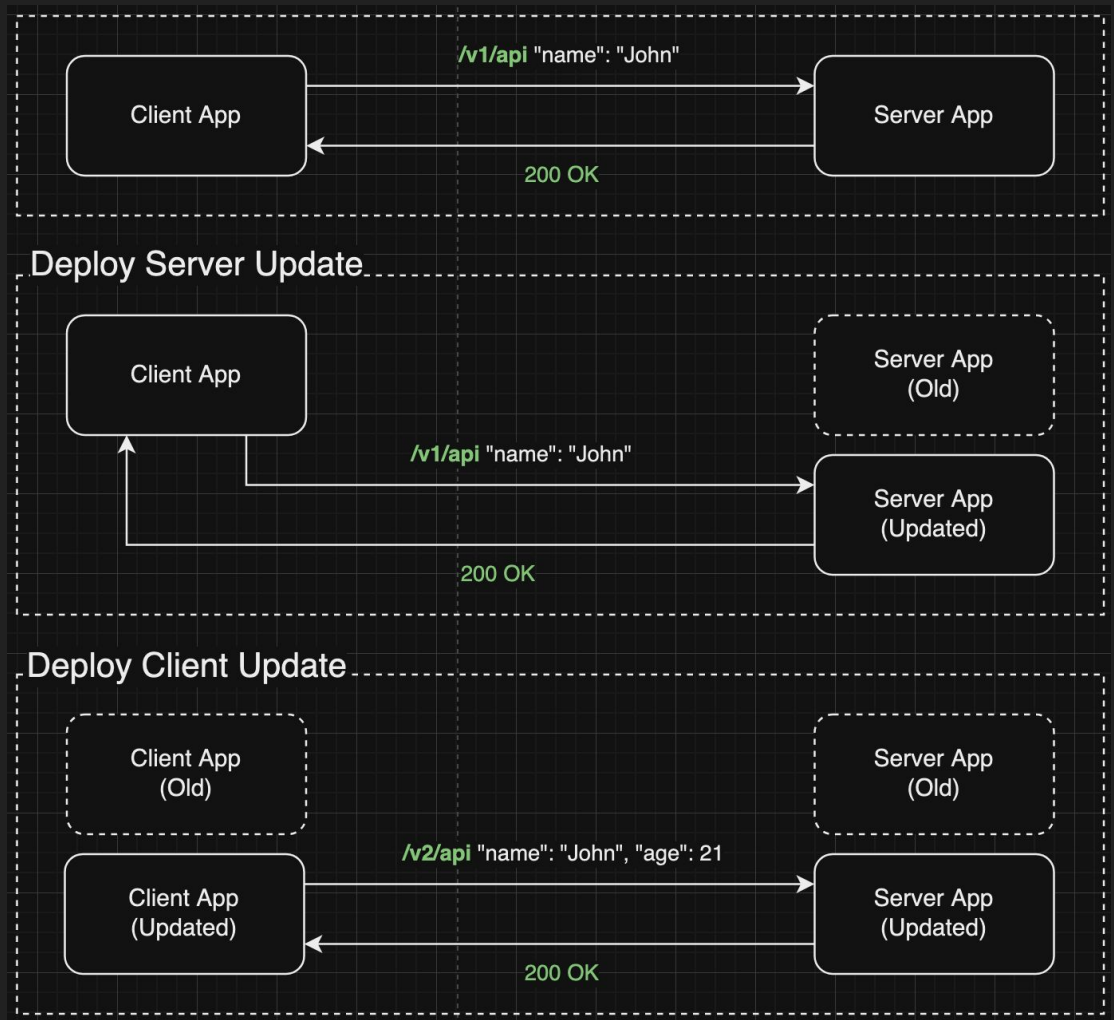


Jeff Valore Mar 11th at 10:50 AM

not sure how heavy of a lift it would be for all the services, but this seems like the kind of situation where api versions would help.

[redacted] Mar 11th at 1:04 PM

we have always been making the changes in non breaking ways


/v1/api "name": "John"
Client App
Server App
200 OK
Deploy Server Update
Client App
Server App
(Old)
/v1/api "name": "John"
Server App
(Updated)
200 OK
Deploy Client Update
Client App
(Old)
Server App
(Old)
/v2/api "name": "John", "age": 21
Client App
(Updated)
Server App
(Updated)
200 OK

Versioning Styles

URL style:

```
curl "https://www.strava.com/api/v3/activities"
```

HTTP header style:

```
curl -H "X-GitHub-Api-Version:2022-11-28"
```

Belt & Suspenders style?

```
curl "https://api.stripe.com/v1/charges" -H "Stripe-Version:2024-06-20"
```

What about GraphQL?

https://graphql.org/learn/best-practices/#versioning

“GraphQL takes a strong opinion on avoiding versioning by providing the tools for the continuous evolution of a GraphQL schema.

GraphQL only returns the data that’s explicitly requested, so new capabilities can be added via new types and new fields on those types without creating a breaking change. This has led to a common practice of always avoiding breaking changes and serving a versionless API.”

… but what about mutations, or removal of a type, or renaming of properties?

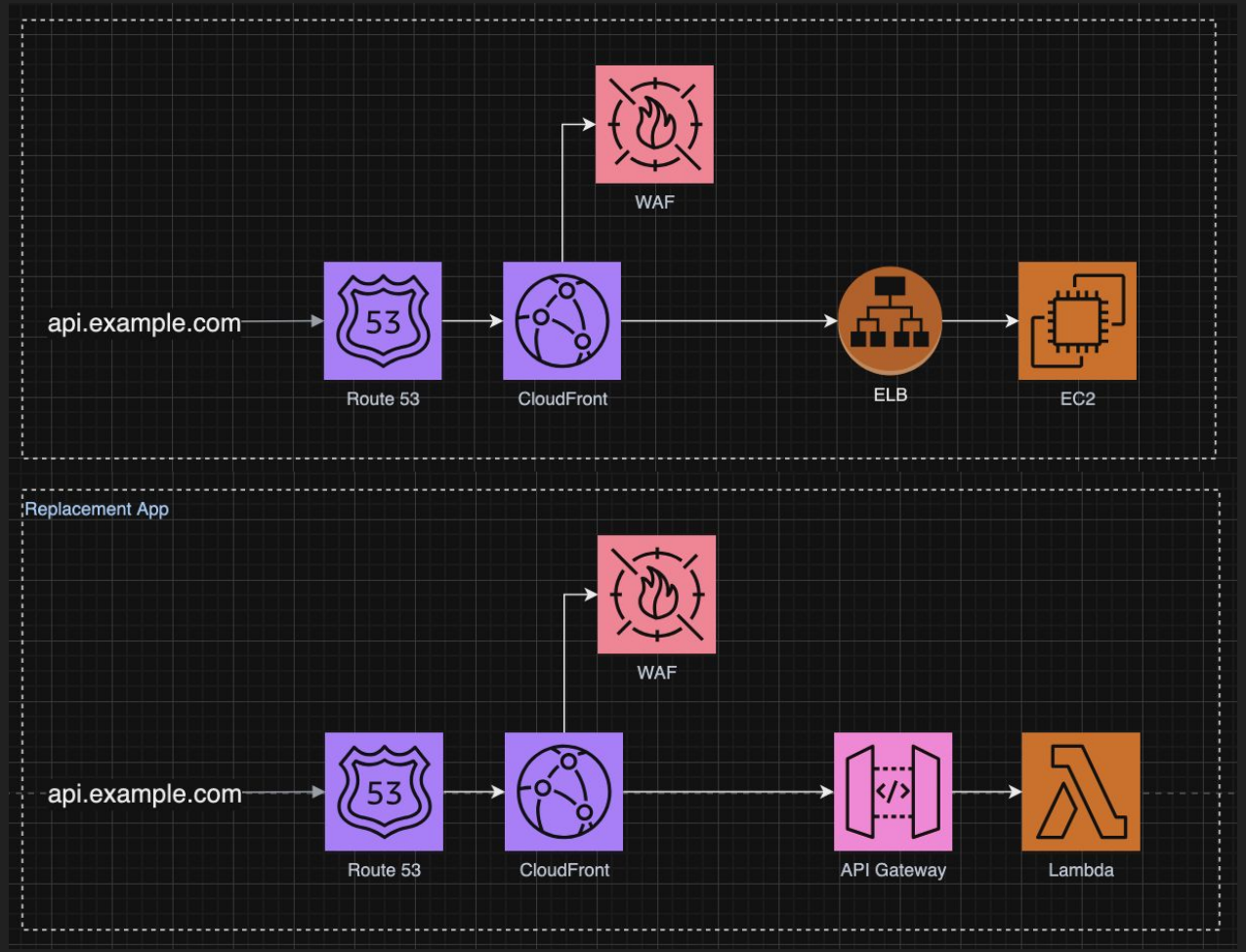
Infrastructure as Code (IaC) and DNS

```
# Example API Gateway with custom domain.
resource "aws_api_gateway_domain_name" "example" {
 certificate_arn = aws_acm_certificate_validation.example.certificate_arn
 domain_name     = "api.example.com"
}
resource "aws_api_gateway_base_path_mapping" "example" {
 api_id      = aws_api_gateway_rest_api.example.id
 domain_name = aws_api_gateway_domain_name.example.domain_name
}

# Example DNS record using Route53.
resource "aws_route53_record" "example" {
 name    = aws_api_gateway_domain_name.example.domain_name
 type    = "A"
 zone_id = aws_route53_zone.example.id

 alias {
   evaluate_target_health = true
   name                   = aws_api_gateway_domain_name.example.cloudfront_domain_name
   zone_id                = aws_api_gateway_domain_name.example.cloudfront_zone_id
 }
}
```

Remember those 4 company renames? Now prod has 4 domain names and QA has 1.

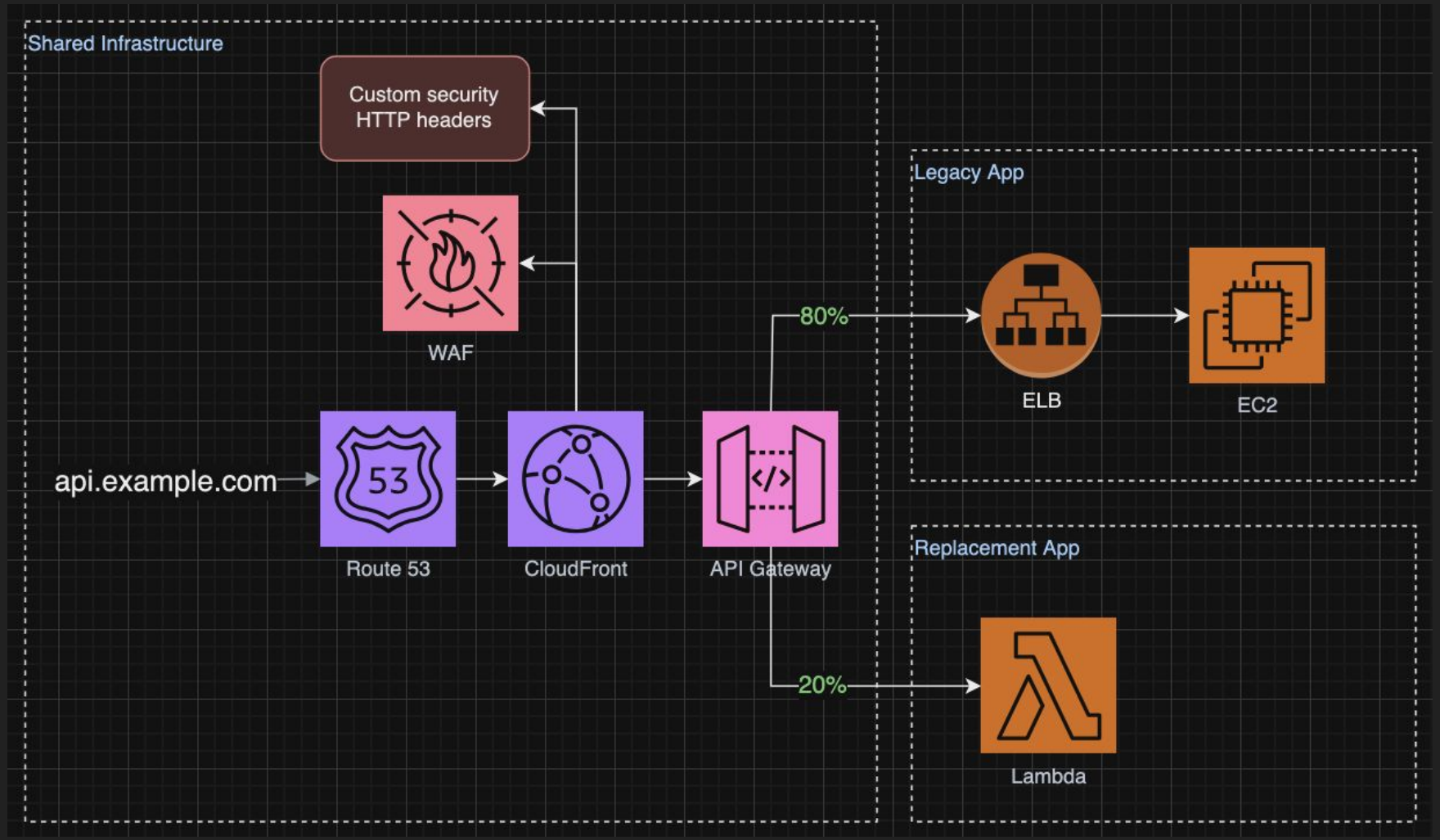
IoC differences between envs hard to manage.

App rewrites use same domain. Conflict between apps.


WAF
api.example.com
53
Route 53
CloudFront
ELB
EC2
Replacement App
WAF
api.example.com
53
Route 53
CloudFront
API Gateway
Lambda

Infrastructure as Code (IaC) and DNS

Since domain names should stay the same for any rewrite of the app or multiple deployments, DNS IaC should not be owned by the app. Move it to its own IaC “project”


Shared Infrastructure
Custom security HTTP headers
WAF
api.example.com
53
Route 53
CloudFront
API Gateway
80%
20%
Legacy App
ELB
EC2
Replacement App
Lambda

Don't assume all requests are going to the same application.
Traffic could route to different systems.

Use an API Gateway to route traffic to a backing application.

🧠 Tenants were migrated 1 at a time to a new application, with no change of URL.

Use an API Gateway to route traffic

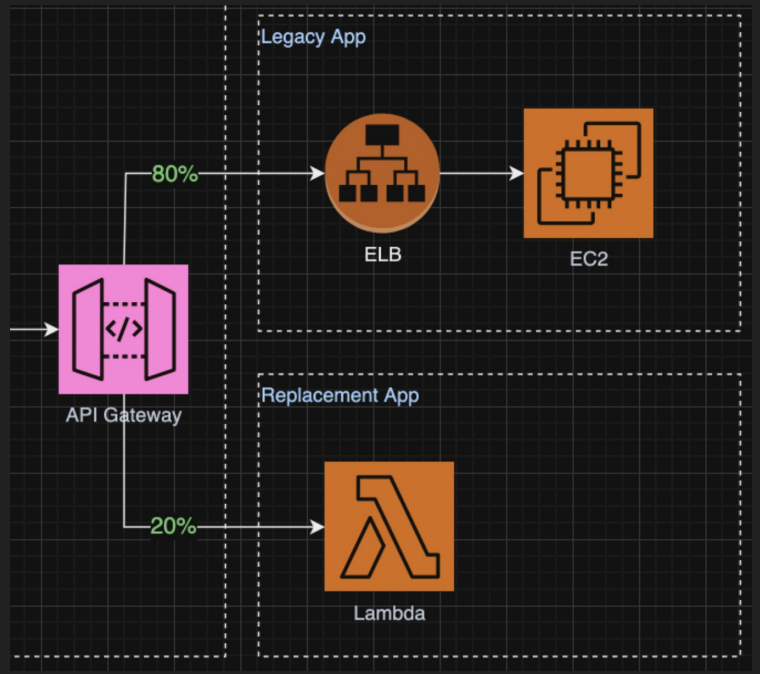
(the pattern, not necessarily the aws product)

- By Version
- By Tenant / Customer
- By A/B test groups
- By % of traffic (canary deploys)

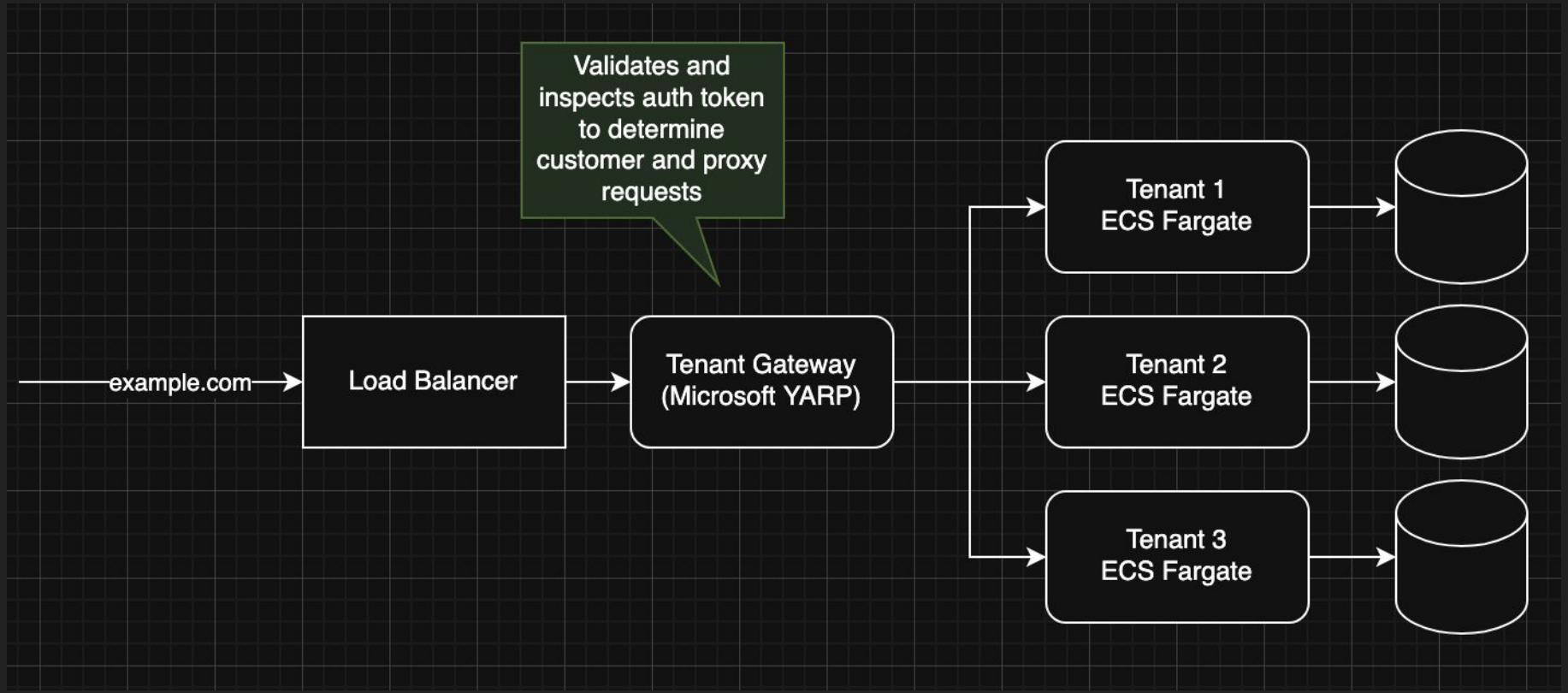
Without your clients having to change the way they call your API.

Could be implemented using:

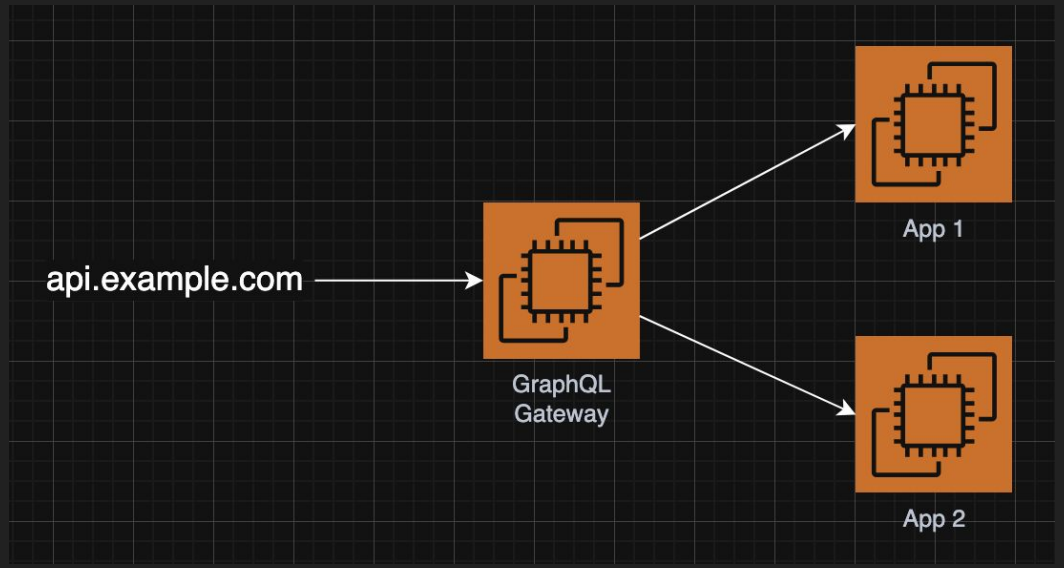
- aws api gateway
- nginx
- haproxy
- yarp



"we should deploy separate infra per tenant"



Example: Federated GraphQL



Move common functionality forward

- Authentication / Access control
- Traces and Metrics (requests rates, failure rates, duration)
- Rate Limiting
- DDoS prevention

You shouldn't have to re-implement these things for each new service.

Observability: Dashboards, Alerting, and SLAs/SLOs

How do you know your API is working?

Performance metrics & traceability built in up front.

Add dimensions to track individual clients/customers, and requests per route.

API Version as a dimension to know when an older version can be deleted.

🧠 Had good GraphQL metrics (until they were deleted) but no tracing.

🥚 Good dashboards with metrics and tracing. Easier to troubleshoot production.

Use Dimensions to slice your data

"Show me [metric] per [dimension]"

Requests	Path
Latency	Version
Errors	Tenant
	User

In ExpressJS & DataDog

```
const perfHooks = require('node:perf_hooks');
const tracer = require('dd-trace');

app.use((req, res, next) => {
 const metricTags = gatherMetricTags(req);
 const start = perfHooks.performance.now();
 try {
   next();
 } catch (error) {
   metricTags.error = true;
 } finally {
   const duration = perfHooks.performance.now() - start;
   tracer.dogstatsd.gauge('api_handler', duration, metricTags);
 }
});
```

```
function gatherMetricTags(req) {
 return {
   environment: process.env.ENV,
   path: req.originalUrl,
   version: req.headers['x-version'] || 'unknown',
   tenant: req.headers['x-tenant-id'] || 'unknown',
   user: req.userId || 'unknown',
   error: false,
 };
}
```

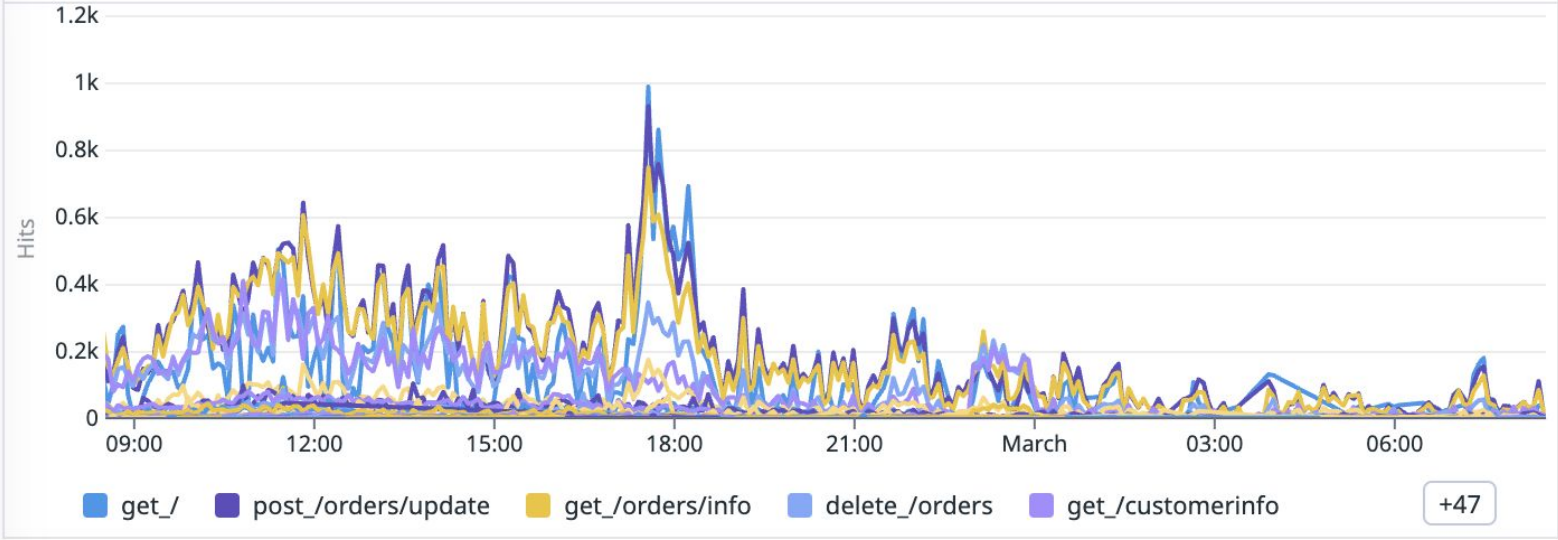
```
public class ApiAsyncTraceActionFilter : Microsoft.AspNetCore.Mvc.Filters.IAsyncActionFilter
{
 public async Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
 {
   using var _scope = Tracer.Instance.StartActive("HandleAsync");

   try
   {
     AddDataToScope(context);
   }
   catch { /* Intentionally ignoring errors. */ }

   try
   {
     await next();
   }
   catch (Exception ex)
   {
     _scope.Span.SetException(ex);
     throw;
   }
 }
```

```
private readonly string[] _autoIncludeProperties = [ "userid", "accountid", "customerid", "orderid" ];

private void AddDataToScope(ActionExecutingContext context)
{
  _scope.Span.ResourceName = context.Controller.GetType().FullName;
  _scope.Span.SetTag("Request Method", context.HttpContext.Request.Method);
  _scope.Span.SetTag("Request Path", context.HttpContext.Request.Path);

  if (context.ActionDescriptor.Parameters.Count > 0)
  {
    var requestParameter = context.ActionDescriptor.Parameters[0];
    var requestType = requestParameter.ParameterType;

    var props = requestType.GetProperties(BindingFlags.Instance | BindingFlags.Public);
    foreach (var prop in props.Where(p => _autoIncludeProperties.Contains(p.Name.ToLower())))
    {
      try
      {
        _scope.Span.SetTag(
          prop.Name,
          prop.GetValue(context.ActionArguments[requestParameter.Name])?.ToString() ?? "null"
        );
      }
      catch { /* Intentionally ignoring errors. */ }
    }
  }
}
```

API Respone Duration By Path

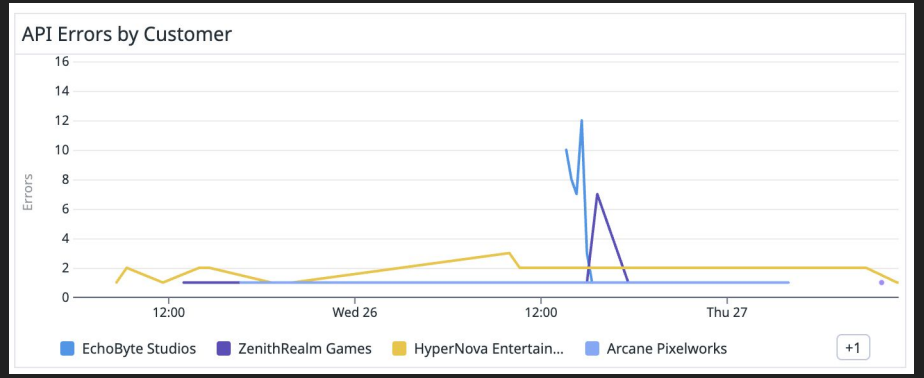
Search

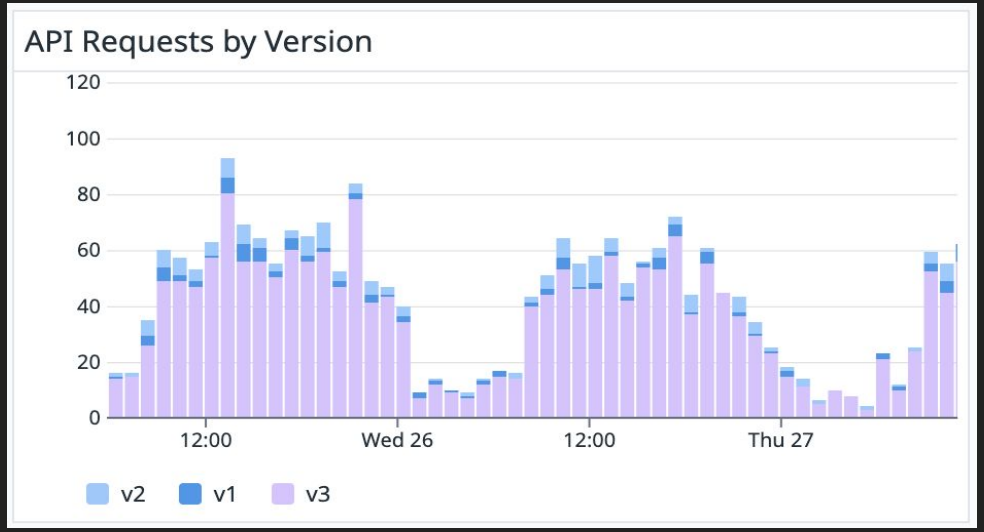
RESOURCE_NAME	REQUESTS	ERRORS	DURATION
post_/orders/update	1,505 hits	1,455 errs	1,386 ms
post_customerinfo	279 hits	—	1,340 ms
get_/customerinfo	214.75k hits	1 errs	1,335 ms
delete_/orders/cancel	279 hits	—	1,317 ms
post_/product	279 hits	—	1,276 ms
get_/products/index	279 hits	—	1,262 ms
get_/index	278 hits	—	1,245 ms



Usage By Customer (% Total Requests)

80.01	PixelForge Studios
50.82	NebulaCraft Games
1.38	Quantum Level Interactive
1.27	Arcane Pixelworks
1.13	HyperNova Entertainment
1.04	EchoByte Studios
1.04	ZenithRealm Games



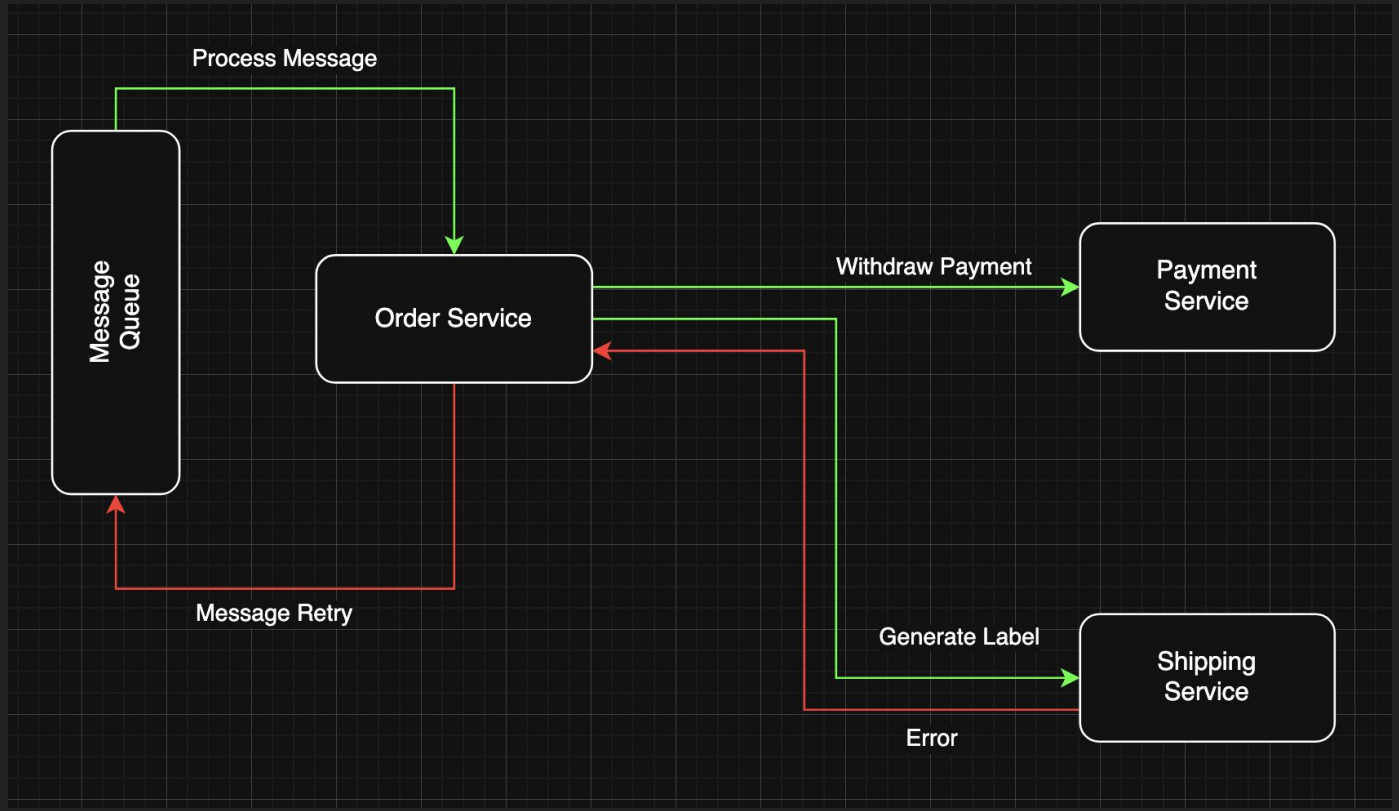


Idempotency

Support a way for customers to repeat duplicate requests and expect idempotency.

```
curl https://api.stripe.com/v1/customers \
  -H "Idempotency-Key: abc123"
```

API Call Idempotency



Designing Web APIs for Long Term Success

1. Document conventions
2. Return Objects, not Arrays
3. Return Errors in a well documented, consistent way
4. Be language agnostic
5. Include versioning
6. Handle shared infrastructure and DNS in a separate IaC project
7. Plan for future replacement. Use an API Gateway or proxy to route traffic
8. Move shared functionality forward in the stack
9. Build common Observability patterns, dashboards, & track SLAs
10. Consider Idempotency

Designing Web APIs for Long Term Success

← Scan to Give Feedback

Jeff Valore
Sr. Consultant NimblePros