API Orchestration in the Cloud

It's just regular API Orchestration

Drew Olson

May 24th, 2021

About Me

Hi, I'm Drew Olson

- Chief Architect at GoFundMe
- Previously Chief Architect at Braintree
- https://drewolson.org

Agenda

Agenda

- API Orchestration
- GraphQL
- AWS + API Orchestration

API Orchestration

API Orchestration

API Orchestration is about decoupling your API from your architectural decisions.

API Orchestration

We'd like to expose a simple, coherent API to our users regardless of how we choose to build the service(s) that power our application.

API Orchestration

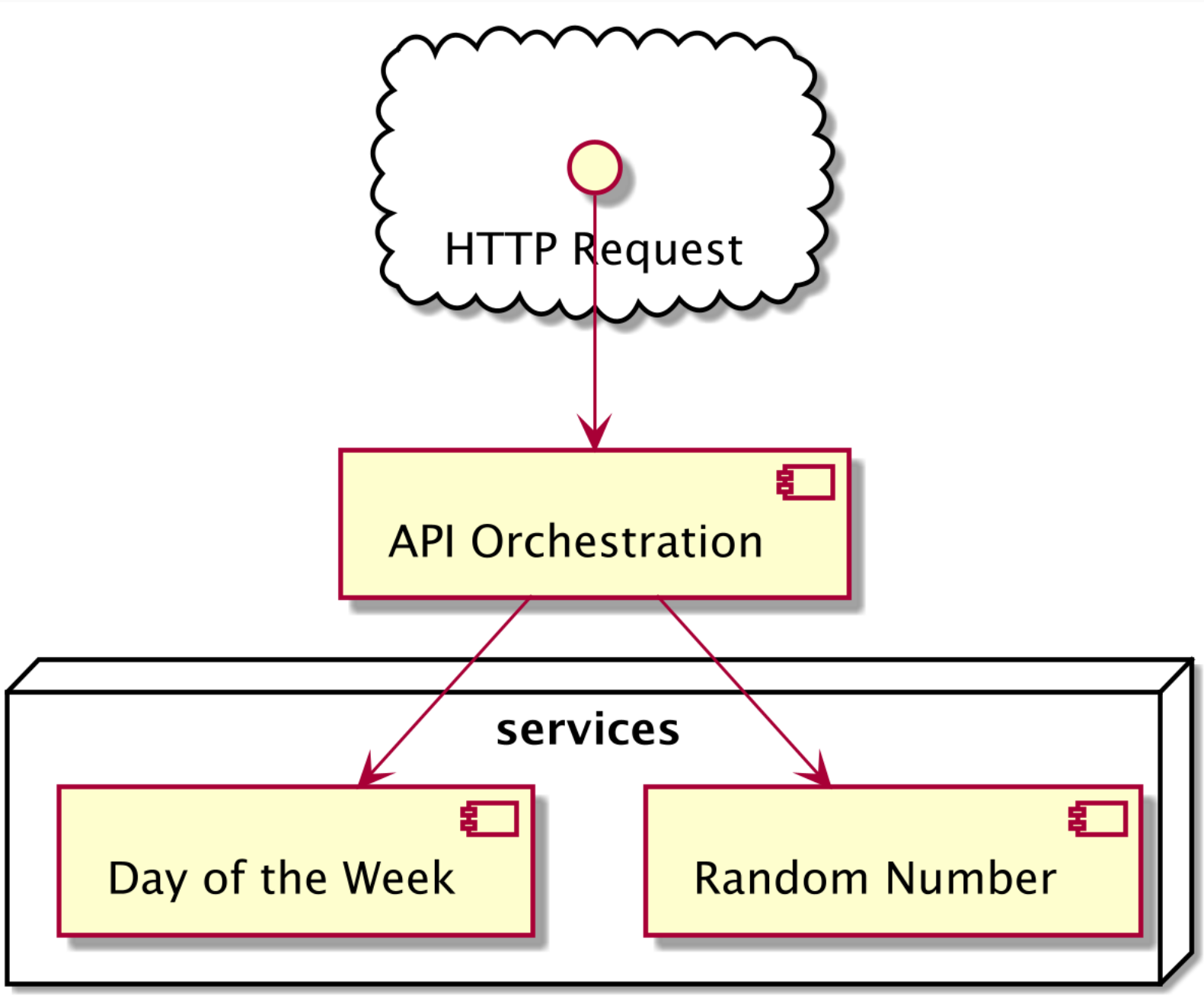
Suppose we have two simple services within our application:

1. Generate a random number between 0 and 9
2. Return the current day of the week

API Orchestration

We can build an API Orchestration layer that provides the capabilities of each of these services by delegating to them and composing their responses as an API response to our user.

API Orchestration



API Orchestration

Let's assume initially these "services" are just functions within our application (please never start with microservices). We can build this "orchestration layer" quite simply.

API Orchestration

Here's our `random_number` function:

```
def random_number():
    return randrange(0, 10)
```

API Orchestration

Here's our `day_of_the_week` function:

```
def day_of_the_week():
    response = requests.get(
        "http://worldclockapi.com/api/json/cst/now"
    )
    body = response.json()

    return body.get("dayOfTheWeek")
```

API Orchestration

Because this code exists locally, our API orchestration layer is very simple:

API Orchestration

Because this code exists locally, our API orchestration layer is very simple:

```
app = FastAPI()

@app.get("/")
def index():
    return {
        "number": random_number(),
        "dayOfTheWeek": day_of_the_week(),
    }
```

API Orchestration

```
$ curl localhost:8000
{"number":9,"dayOfTheWeek":"Sunday"}%
```

API Orchestration

If we eventually chose to extract `random_number` and `day_of_the_week` to separate services, our orchestration layer gets more complicated.

API Orchestration

```
app = FastAPI()

@app.get("/")
def index():
    return {
        "number": requests
                    .get("http://random.service")
                    .json()
                    .get("number"),
        "dayOfTheWeek": requests
                    .get("http://day-of-the-week.service")
                    .json()
                    .get("number"),
    }
```

API Orchestration

Now we're making two HTTP requests to downstream services.

API Orchestration

Now we're making two HTTP requests to downstream services.

If one service call fails, the whole API request fails.

API Orchestration

Now we're making two HTTP requests to downstream services.

If one service call fails, the whole API request fails.

If either service call is expensive, we pay this penalty for every API request.

API Orchestration

We can do better.

GraphQL

GraphQL

GraphQL is a specification for an API query language and a set of executor libraries written in several languages.

GraphQL

GraphQL is *not*:

- A database
- A serialization format
- Super complicated

GraphQL

Our clients make a JSON request to our server in the following format:

```
{
  "query": "<graphql query>"
}
```

GraphQL

We pass the query to an executor, which returns a response in the following format:

```
{
  "data": { ... },
  "errors": null | [{ ... }]
}
```

GraphQL - Hello world

We'll be using `python`'s `graphene` library to execute our GraphQL queries.

GraphQL - Hello world

```
class Query(graphene.ObjectType):
    ping = graphene.String()

    def resolve_ping(self, info):
        return "pong"
```

GraphQL - Hello world

And we'll use `fastapi` to accept queries via HTTP + JSON. This code never changes across our examples.

GraphQL - Hello world

```
schema = graphene.Schema(query=Query)

app = FastAPI()

class GraphQLRequest(BaseModel):
    query: str

@app.post("/")
def index(request: GraphQLRequest):
    result = schema.execute(
        request.query,
    )

    return result.formatted
```

GraphQL - Hello world

We can now make an HTTP request with the following query:

```
{
  ping
}
```

```
$ curl -X POST -d '{"query": "{ ping }"}' localhost:8000
{"data":{"ping":"pong"},"errors":null}%
```

GraphQL - Providing Arguments

```
class Query(graphene.ObjectType):
    hello = graphene.Field(
        graphene.String,
        name=graphene.String(required=True),
    )

    def resolve_hello(self, info, name):
        return f"Hello, {name}!"
```

GraphQL - Providing Arguments

Request:

```
{
  hello(name: "Drew")
}
```

Response:

```
{
  "data": {
    "hello":"Hello, Drew!"
  },
  "errors":null
}
```

GraphQL - Nested Fields

```
class Greeting(graphene.ObjectType):
    hello = graphene.String()
    goodbye = graphene.String()

class Query(graphene.ObjectType):
    greeting = graphene.Field(
        Greeting,
        name=graphene.String(required=True),
    )

    def resolve_greeting(self, info, name):
        return {
            "hello": f"Hello, {name}!",
            "goodbye": f"Goodbye, {name}!",
        }
```

GraphQL - Nested Fields

Request:

```
{
  greeting(name: "Drew") {
    hello
    goodbye
  }
}
```

GraphQL - Our Example

```
def random_number():
    return randrange(0, 10)

def day_of_the_week():
    response = requests.get(
        "http://worldclockapi.com/api/json/cst/now"
    )
    body = response.json()

    return body.get("dayOfTheWeek")
```

GraphQL - Our Example

```
class Query(graphene.ObjectType):
    day_of_the_week = graphene.String()
    random_number = graphene.Int()

    def resolve_day_of_the_week(self, info):
        return day_of_the_week()

    def resolve_random_number(self, info):
        return random_number()
```

GraphQL - Our Example

Request:

```
{
  randomNumber
  dayOfTheWeek
}
```

GraphQL - Our Example

Response:

```
{
  "data": {
    "randomNumber": 6,
    "dayOfTheWeek": "Saturday"
  },
  "errors": null
}
```

GraphQL

BUT WAIT THERE'S MORE!

GraphQL

BUT WAIT THERE'S MORE!

In GraphQL, the server will only resolve the **fields you ask for**. This means if you only ask for `randomNumber`, the server **will not** make an external HTTP request to fetch the current day.

GraphQL

Request:

```
{
  randomNumber
}
```

GraphQL

Response:

```
{
  "data": {
    "randomNumber": 8
  },
  "errors": null
}
```

GraphQL

Client-driven responses allow our orchestration layer to be far more efficient. We do only the work required to return the **exact fields** our client requests.

GraphQL

Client-driven responses allow our orchestration layer to be far more efficient. We do only the work required to return the **exact fields** our client requests.

This is especially important if each of our fields is resolved by calling another service.

GraphQL

GraphQL also gives us **partial responses** for free.

GraphQL

GraphQL also gives us **partial responses** for free.

This means that if we fail to resolve one of our fields but succeed in resolving a second, we will send the client the data we were able to resolve and an error representing the failure.

GraphQL

GraphQL also gives us **partial responses** for free.

This means that if we fail to resolve one of our fields but succeed in resolving a second, we will send the client the data we were able to resolve and an error representing the failure.

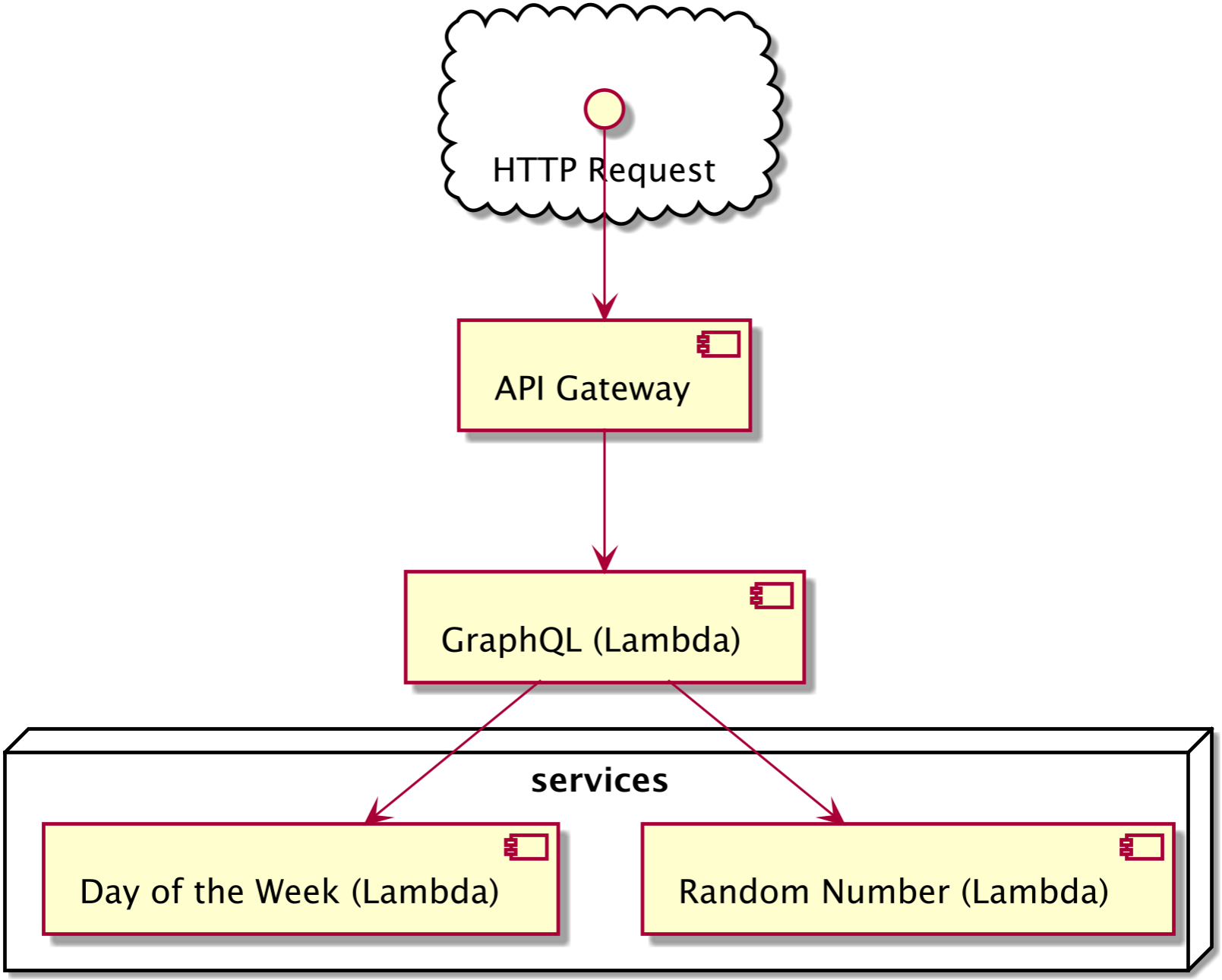
A single resolution failure **does not** fail the whole request.

AWS + API Orchestration

“Cloud” Orchestration

We’re going to implement an orchestration layer identical to our first demo, but using AWS’s API Gateway and Lambda.

“Cloud” Orchestration



“Cloud” Orchestration

LIVE DEMO THAT I HOPE WON’T FAIL!

Fin

Thanks! Questions?

(Also you should play bridge)