Migrations

The Actual Hardest Problem in Computer Science

Matt Ranney / December 2022

Hello

Matt Ranney

Principal Engineer

he/him

Active, notifications snoozed

10:03 AM local time

US

Message | Call

Contact information

Email Address
matt.ranney@doordash.com

Affiliations

Cost Center
Infrastructure Engineering

Motivation

Motivation

- Nobody wants to work on migrations, but why?

Motivation

- Nobody wants to work on migrations, but why?
- Turns out they are super hard.

Motivation

- Nobody wants to work on migrations, but why?
- Turns out they are super hard.
- You are all very wise and deserve a raise.

Migrations

Migrations

- Converting from one software system or service to another.

Migrations

- Converting from one software system or service to another.
- Decomposing an API “monolith”.

Migrations

- Converting from one software system or service to another.
- Decomposing an API “monolith”.
- Moving databases around for whatever reason.

Why Are Migrations Hard?

Why Are Migrations Hard?

- Old systems are generally not well understood. People move around.

Why Are Migrations Hard?

- Old systems are generally not well understood. People move around.
- Most systems are not built with upgrades in mind.

Why Are Migrations Hard?

- Old systems are generally not well understood. People move around.
- Most systems are not built with upgrades in mind.
- When designing software interfaces, it’s very hard to anticipate the ways in which you’ll want to change them later.

Why Are Migrations Hard?

- Old systems are generally not well understood. People move around.
- Most systems are not built with upgrades in mind.
- When designing software interfaces, it’s very hard to anticipate the ways in which you’ll want to change them later.
- Many modern software systems cannot be turned off without significant business consequences.

Making Changes to Always-On Systems

Making Changes to Always-On Systems

- Everything you do must be forward and backward compatible.

Making Changes to Always-On Systems

- Everything you do must be forward and backward compatible.
- If anything goes wrong, and it certainly will, roll it back.

DoorDash Migration Example

```
commit 3936252e85bf7a2866fbc23e95c5619ff6bcd4e8
Author: Andy Fang <andyfang@stanford.edu>
Date:   Sat Feb 2 00:19:17 2013 -0800

    add initial django project

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000000..fab8f691c0
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,2 @@
+venv
+*.pyc
\ No newline at end of file
diff --git a/Procfile b/Procfile
new file mode 100644
index 0000000000..c09a552101
--- /dev/null
+++ b/Procfile
@@ -0,0 +1 @@
+web: python manage.py runserver 0.0.0.0:$PORT --noreload
diff --git a/doorstep/__init__.py b/doorstep/__init__.py
new file mode 100644
index 0000000000..e69de29bb2
diff --git a/doorstep/settings.py b/doorstep/settings.py
new file mode 100644
index 0000000000..e5c2417617
--- /dev/null
+++ b/doorstep/settings.py
@@ -0,0 +1,155 @@
+# Django settings for doorstep project.
+
+DEBUG = True
+TEMPLATE_DEBUG = DEBUG
+
+ADMINS = (
+    # ('Your Name', 'your_email@example.com'),
+)
+
+MANAGERS = ADMINS
+
```

DoorDash Migration Example

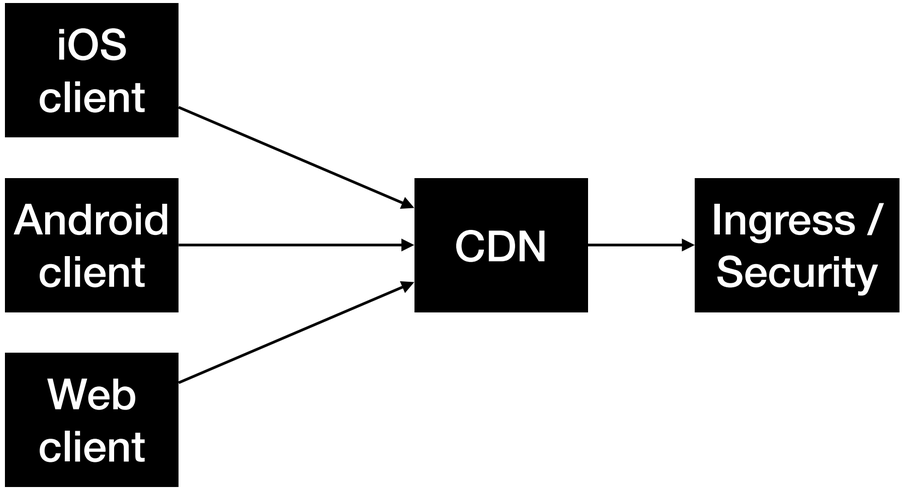
DoorDash Migration Example

iOS client

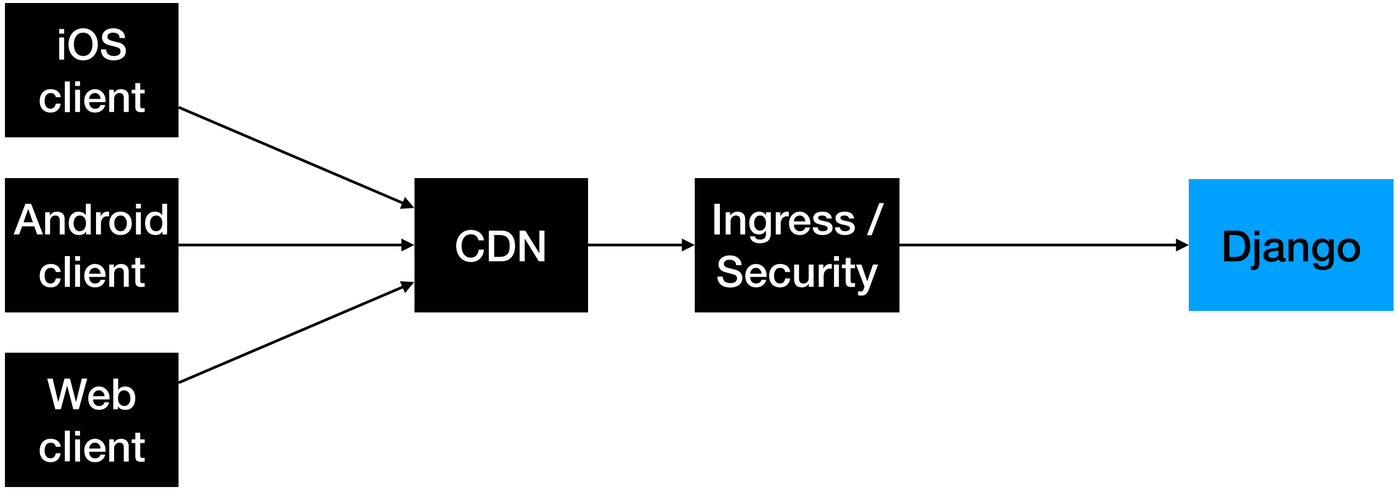
Android client

Web client

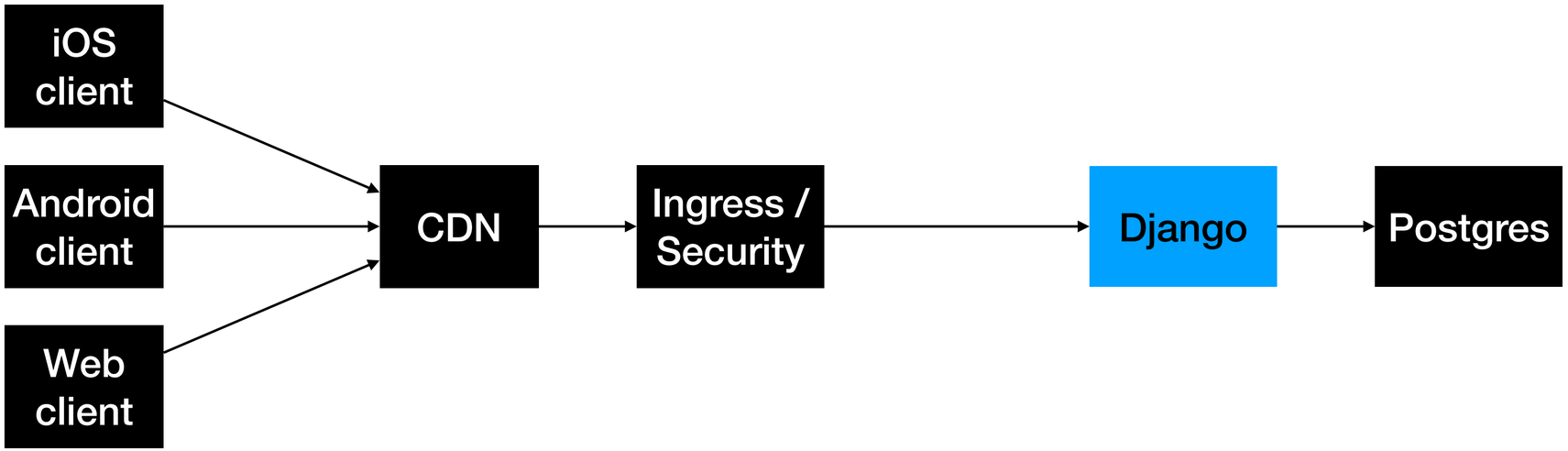
DoorDash Migration Example



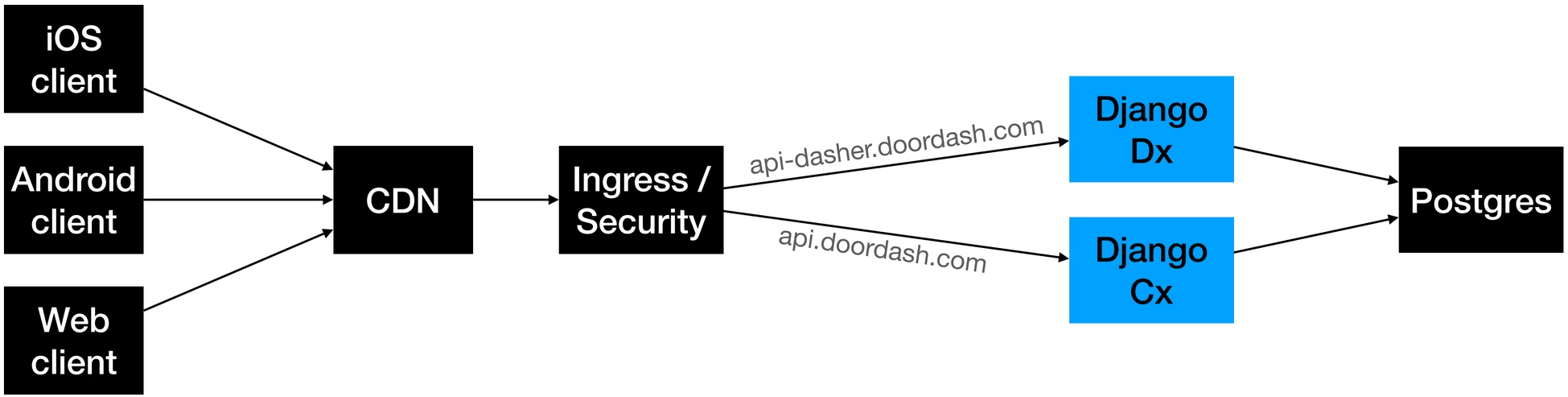
DoorDash Migration Example



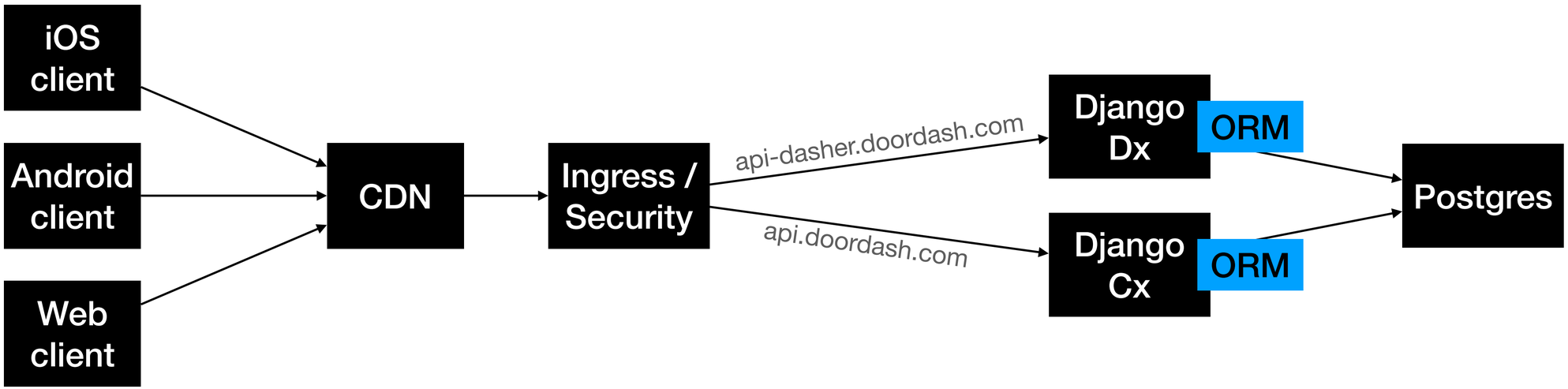
DoorDash Migration Example



DoorDash Migration Example



DoorDash Migration Example



Now We Must Decide

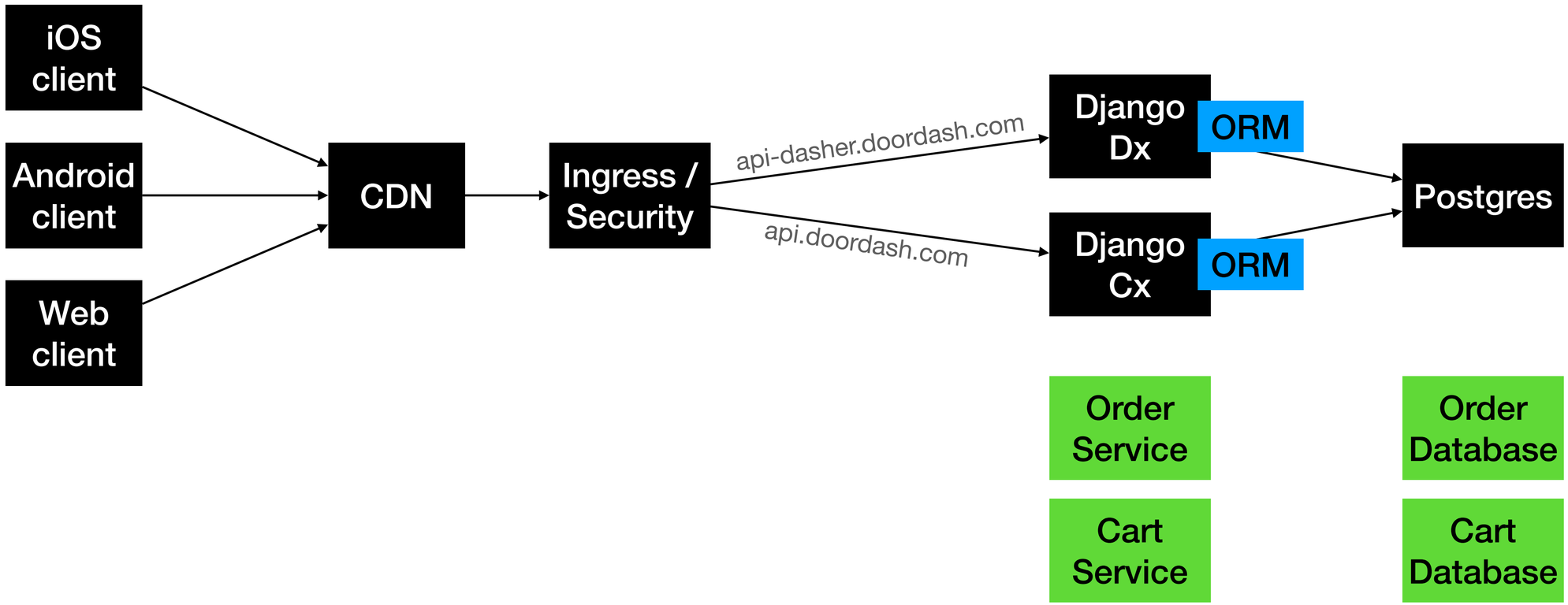
Now We Must Decide

- Rewrite the whole thing?

Now We Must Decide

- Rewrite the whole thing?
- Break off some small pieces?

DoorDash Migration Example



How To Break Off Some Small Pieces

How To Break Off Some Small Pieces

- All of these options sound bad.

How To Break Off Some Small Pieces

- All of these options sound bad.
- All of these options ARE bad.

How To Break Off Some Small Pieces

- All of these options sound bad.
- All of these options ARE bad.
- One is generally less bad, depending on the situation.

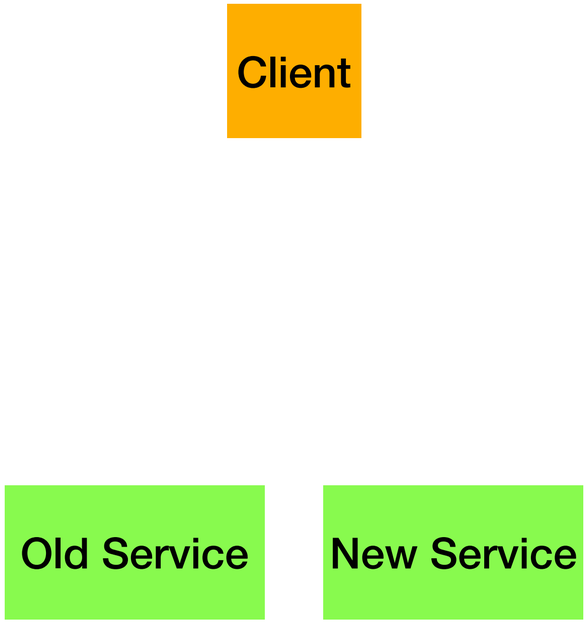
How To Break Off Some Small Pieces

- All of these options sound bad.
- All of these options ARE bad.
- One is generally less bad, depending on the situation.
- Remember that YOU are the one that had to go and have a successful business with a large team.

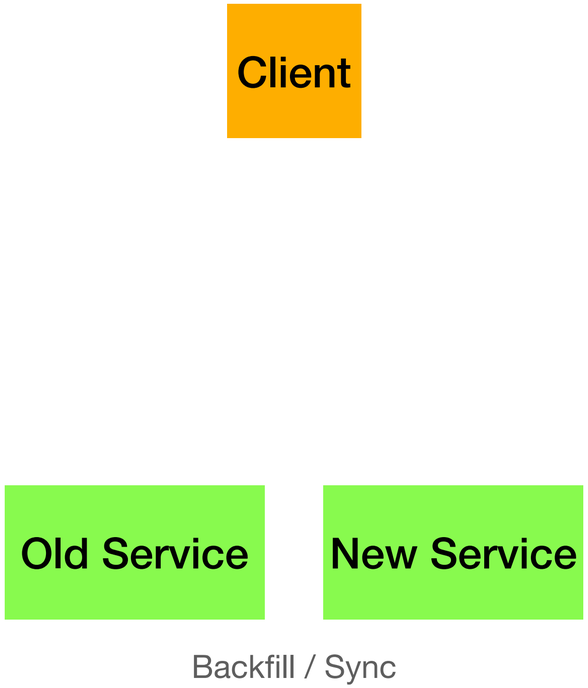
Making Changes to Always-On Systems

- Everything you do must be forward and backward compatible.
- If anything goes wrong, and it certainly will, roll it back.

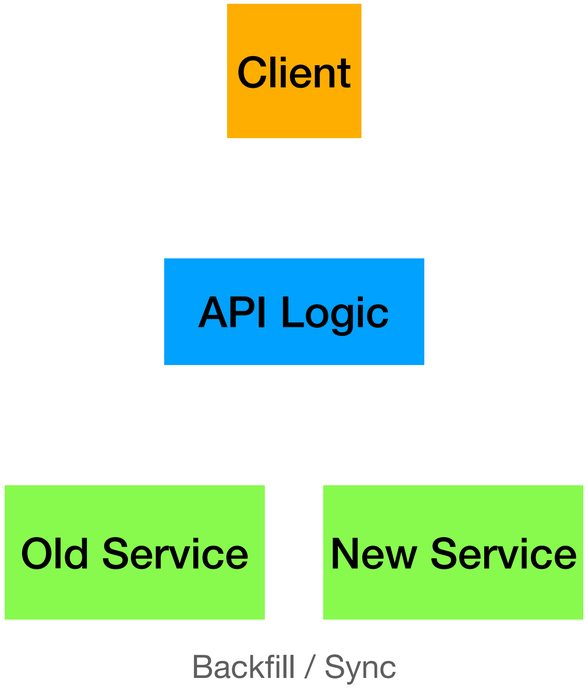
Dual Write



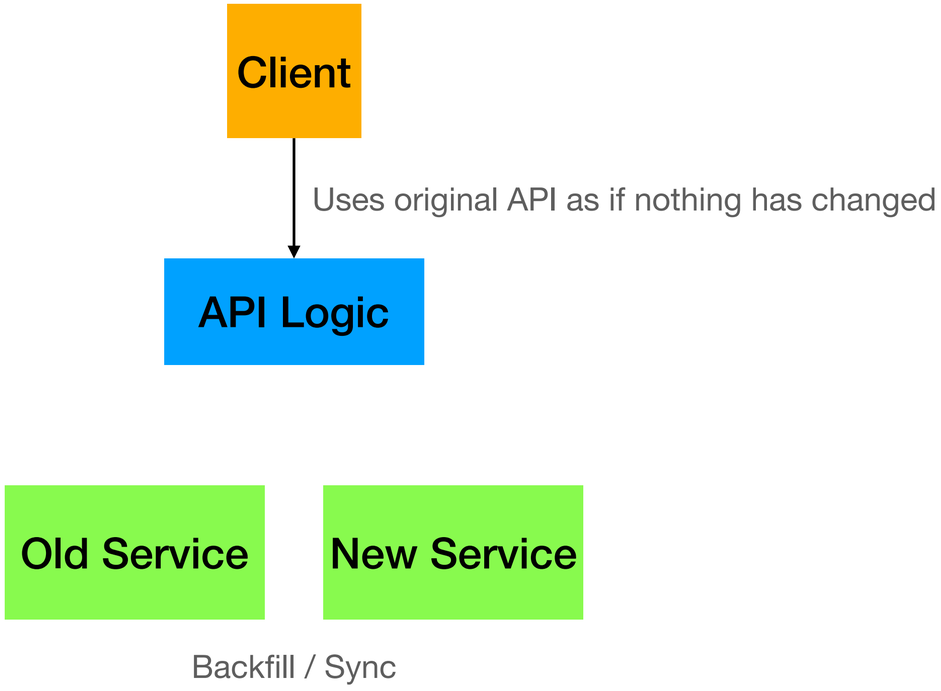
Dual Write



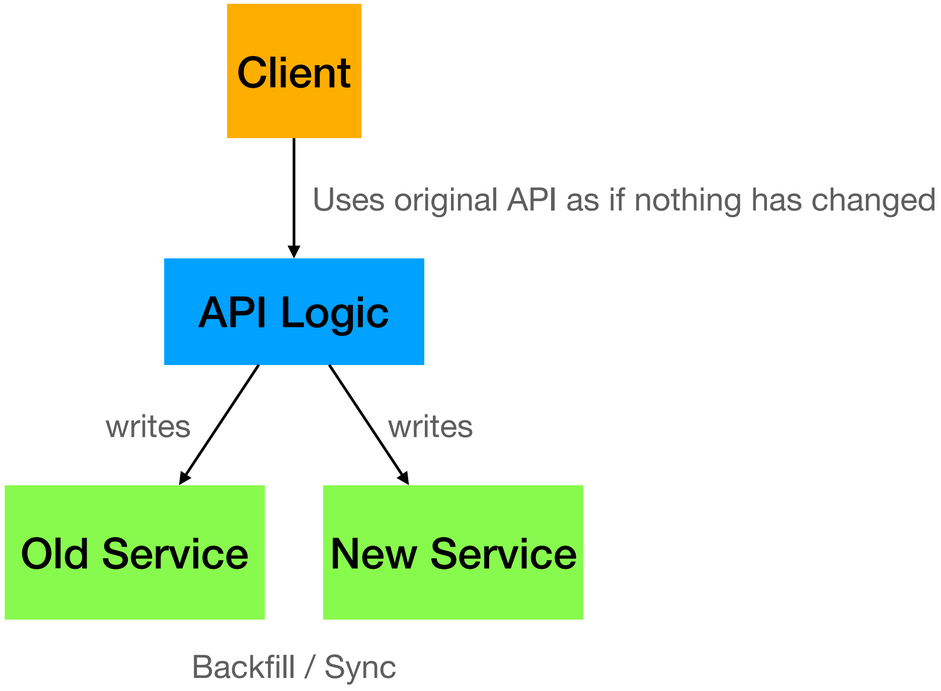
Dual Write



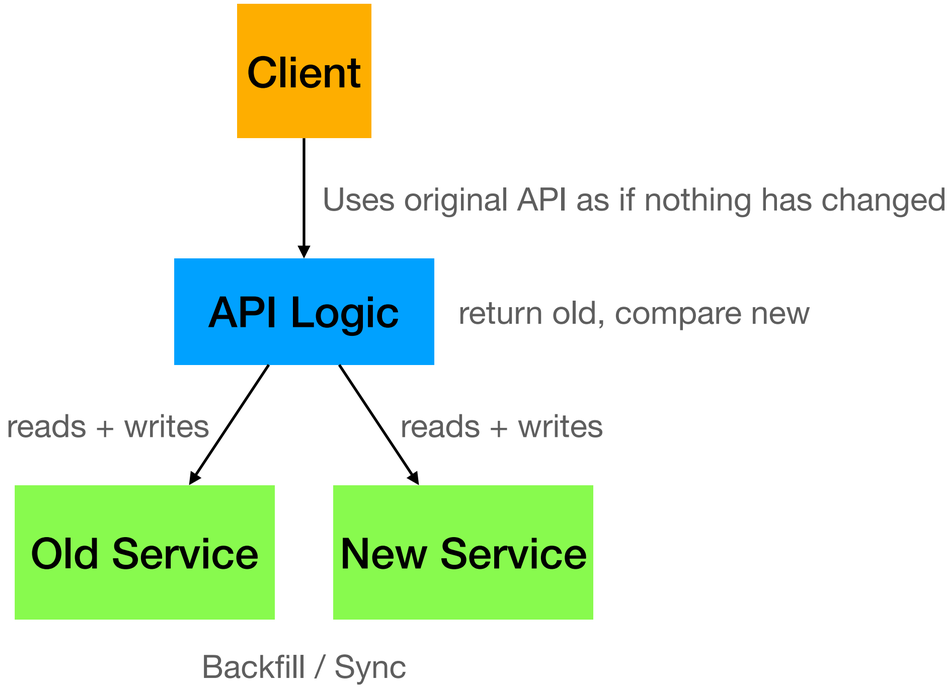
Dual Write


Client
Uses original API as if nothing has changed
API Logic
Old Service
New Service
Backfill / Sync

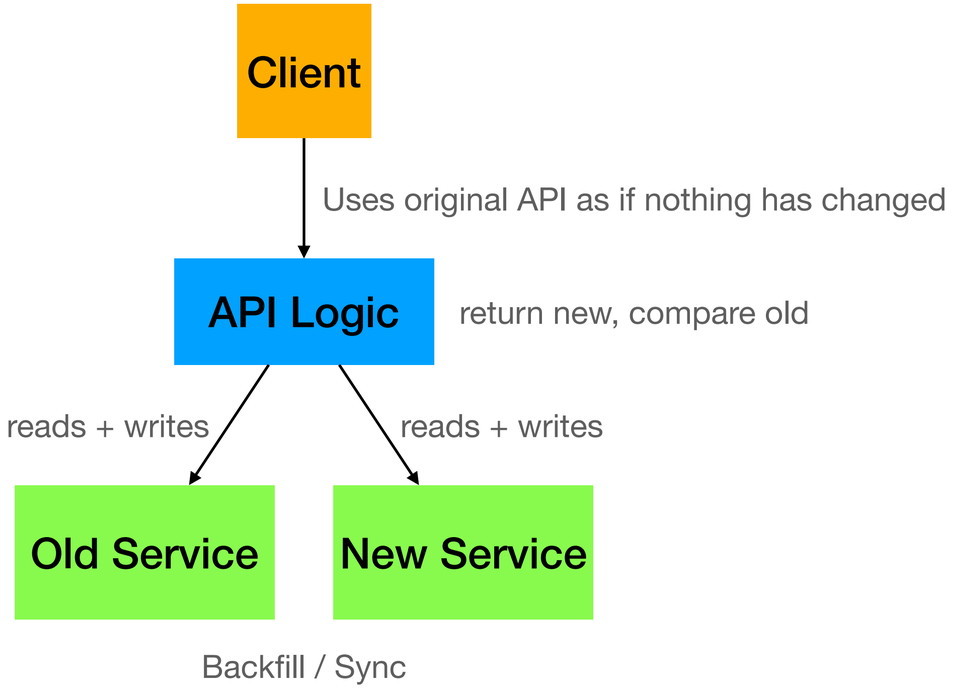
Dual Write


Client
Uses original API as if nothing has changed
API Logic
writes
writes
Old Service
New Service
Backfill / Sync

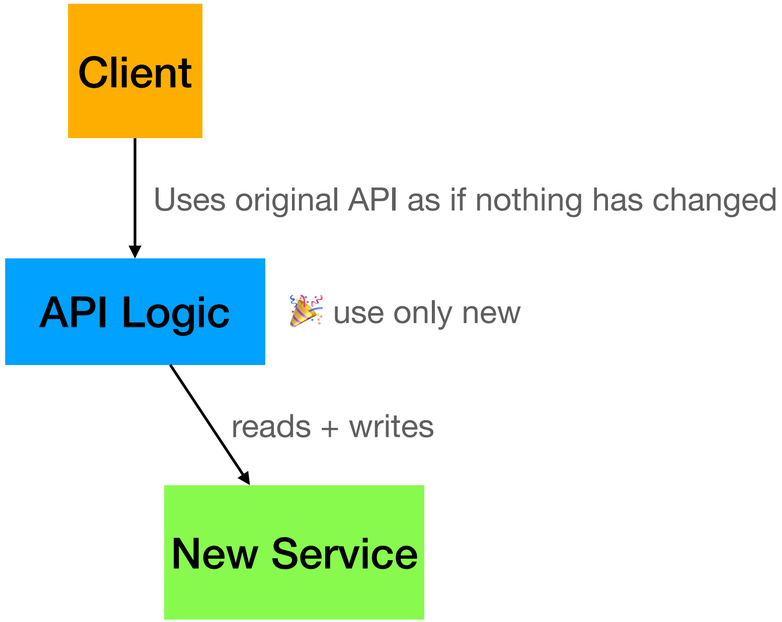
Dual Write


Client
Uses original API as if nothing has changed
API Logic
return old, compare new
reads + writes
reads + writes
Old Service
New Service
Backfill / Sync

Dual Write


Client
Uses original API as if nothing has changed
API Logic
return new, compare old
reads + writes
reads + writes
Old Service
New Service
Backfill / Sync

Dual Write


Client
Uses original API as if nothing has changed
API Logic
🎉 use only new
reads + writes
New Service

This Sounds Bad

This Sounds Bad

- What if I’m using auto-incrementing keys?

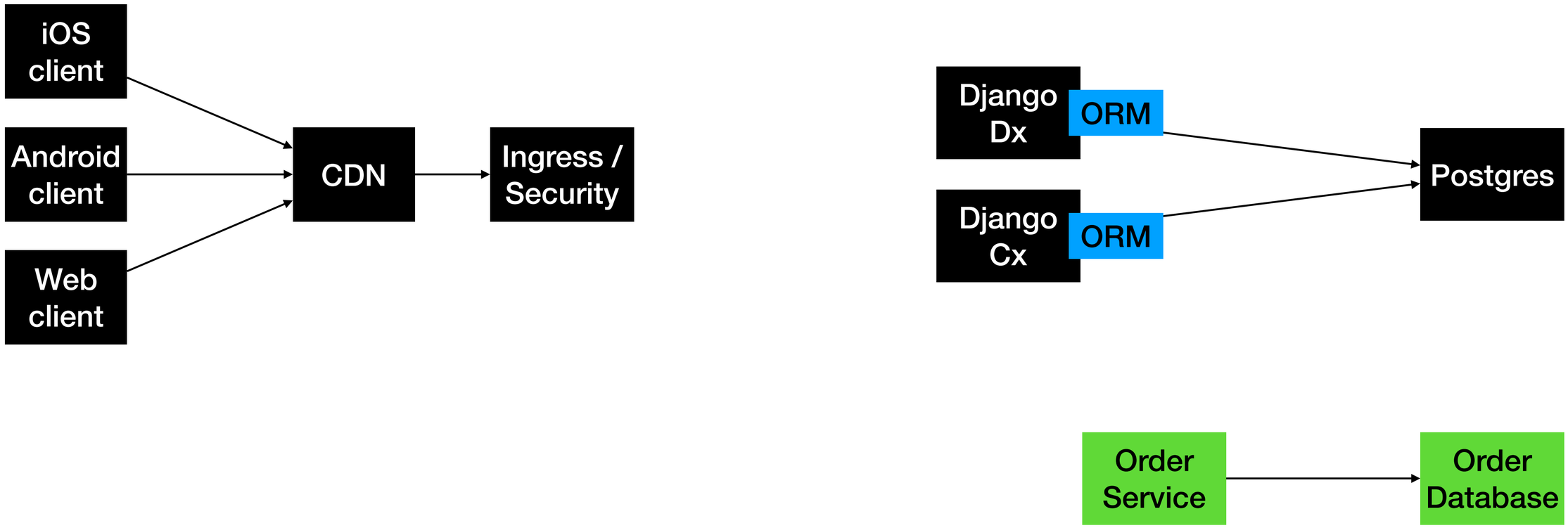
This Sounds Bad

- What if I’m using auto-incrementing keys?
- What if one write fails and the other succeeds?

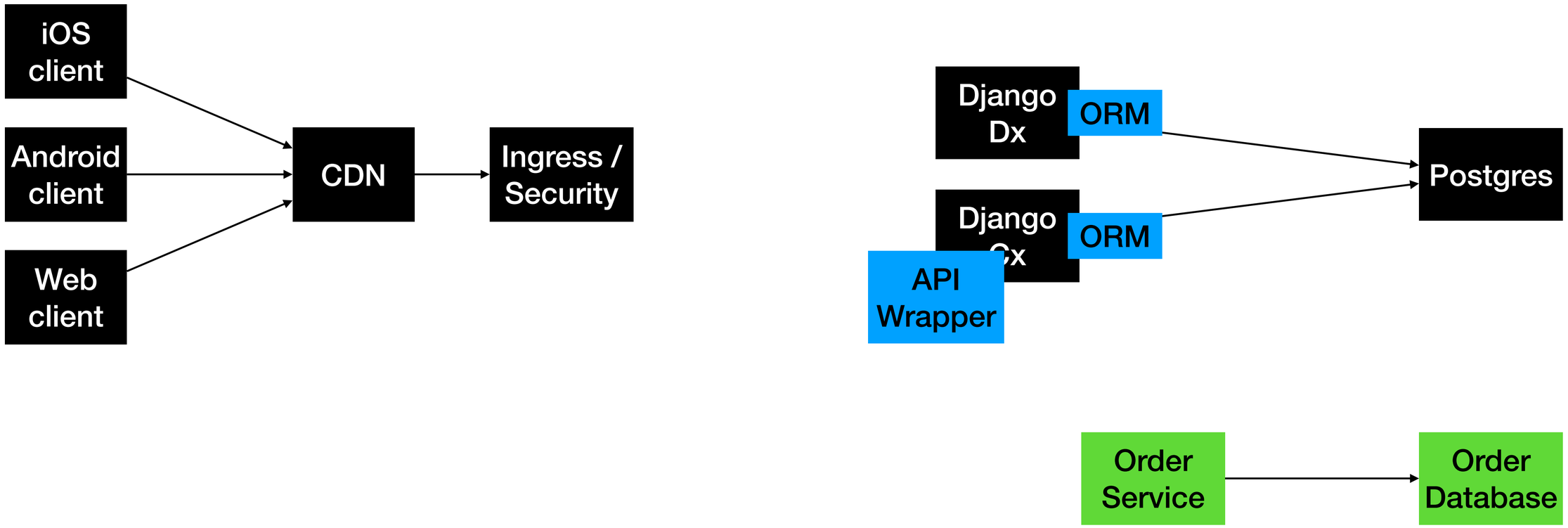
This Sounds Bad

- What if I'm using auto-incrementing keys?
- What if one write fails and the other succeeds?
- My API is super important and I NEED transactions.

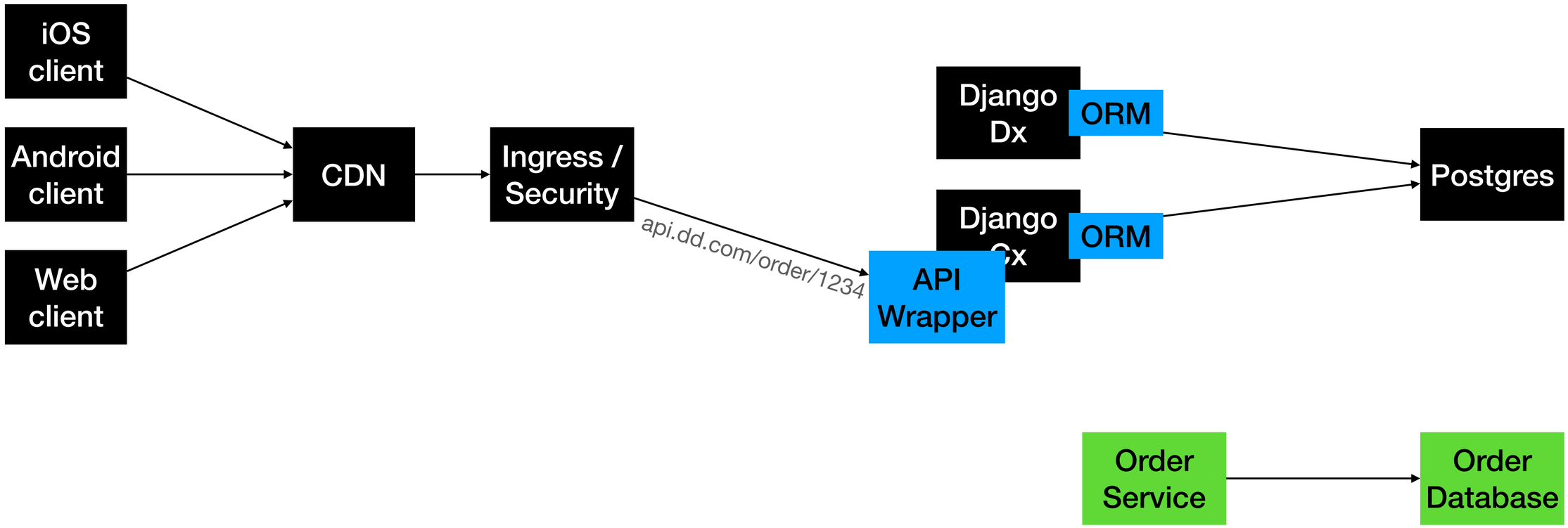
API “Dual Write”



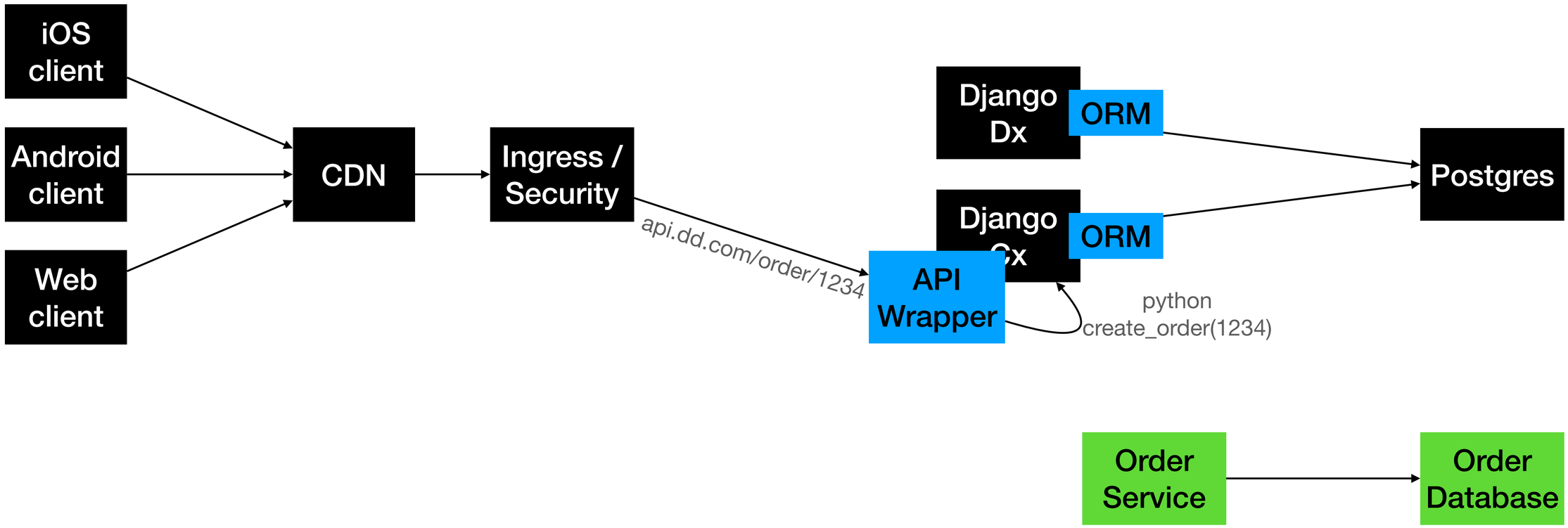
API “Dual Write”



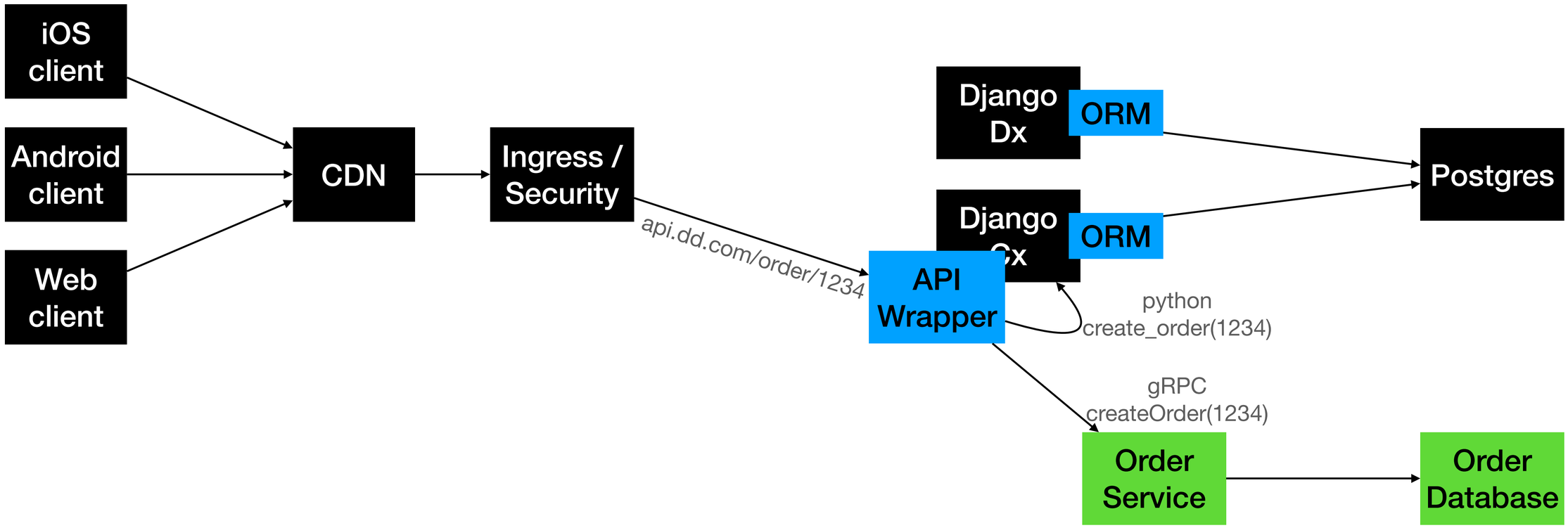
API “Dual Write”



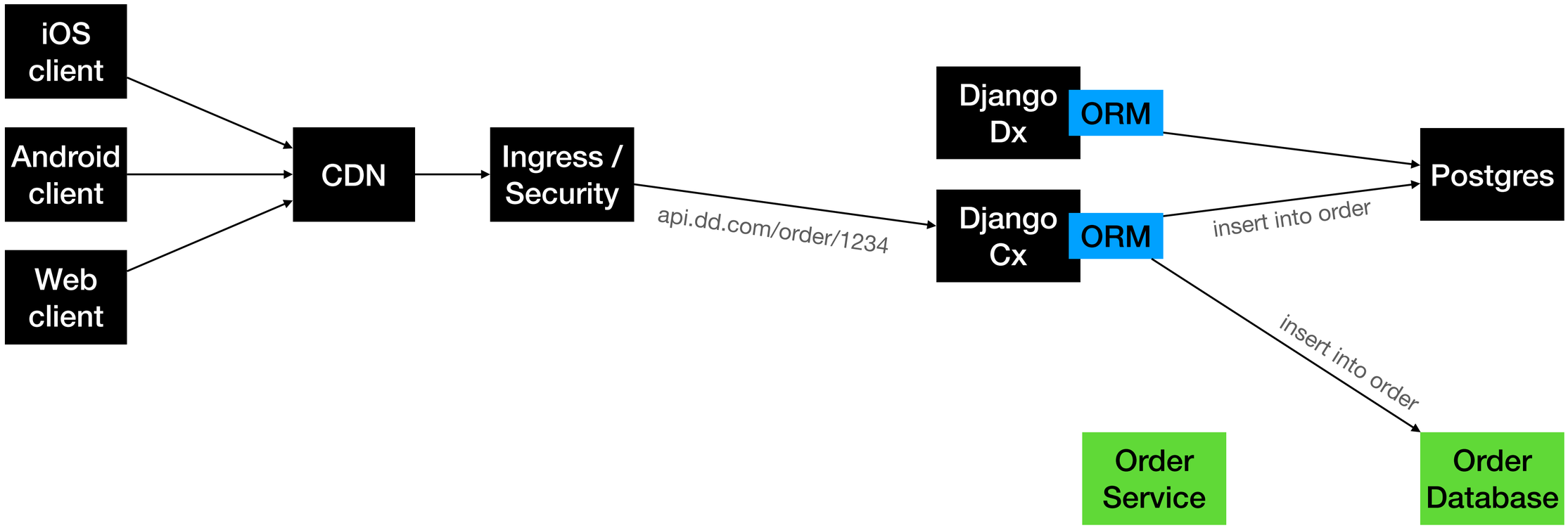
API “Dual Write”



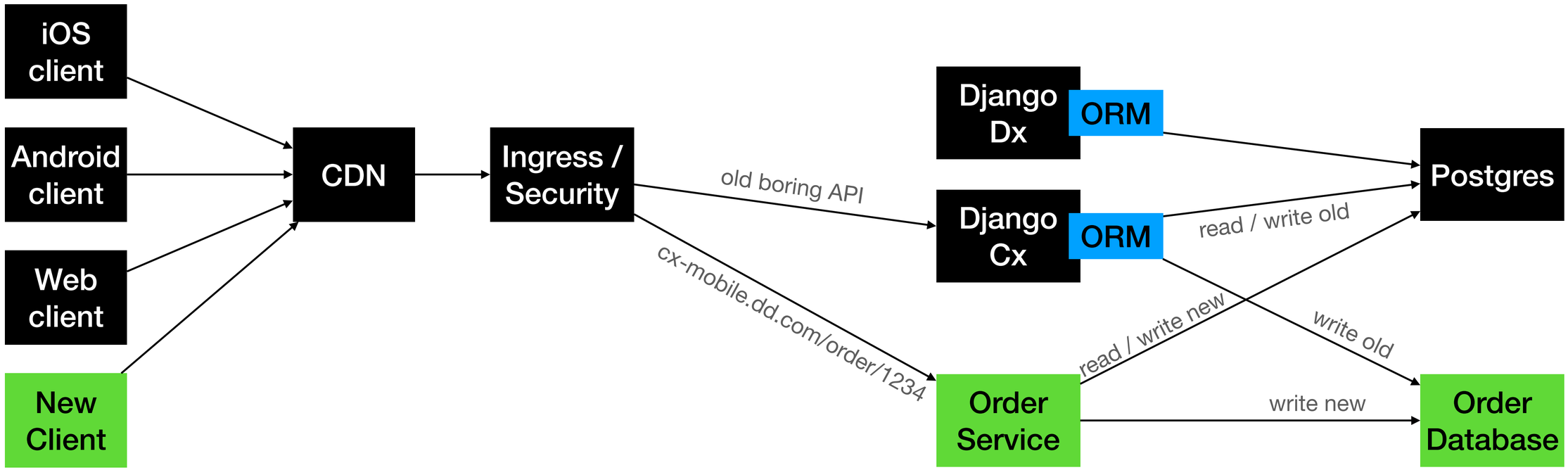
API “Dual Write”



Database “Dual Write”



Surprising Third Option



What Did DD Do?

What Did DD Do?

- Started with DB Dual Write.

What Did DD Do?

- Started with DB Dual Write.
- Took a long time and corrupted some data. Product velocity impacted.

What Did DD Do?

- Started with DB Dual Write.
- Took a long time and corrupted some data. Product velocity impacted.
- Then went with shared DB approach.

What Did DD Do?

- Started with DB Dual Write.
- Took a long time and corrupted some data. Product velocity impacted.
- Then went with shared DB approach.
- This made everyone happier… at first.

What Did DD Do?

- Started with DB Dual Write.
- Took a long time and corrupted some data. Product velocity impacted.
- Then went with shared DB approach.
- This made everyone happier… at first.
- Growth made this challenging, and eventually we were stuck.

What Did DD Do?

- Started with DB Dual Write.
- Took a long time and corrupted some data. Product velocity impacted.
- Then went with shared DB approach.
- This made everyone happier… at first.
- Growth made this challenging, and eventually we were stuck.
- Went back to apply a mix of API and DB Dual Write

What Did DD Do?

- Started with DB Dual Write.
- Took a long time and corrupted some data. Product velocity impacted.
- Then went with shared DB approach.
- This made everyone happier… at first.
- Growth made this challenging, and eventually we were stuck.
- Went back to apply a mix of API and DB Dual Write
- Also, sadly, we had to do some client forced upgrades.

How To Decide Which Technique Is Best?

How To Decide Which Technique Is Best?

- API Dual Write / DB Dual Write

How To Decide Which Technique Is Best?

- API Dual Write / DB Dual Write
  - Backwards compatible. Only 1 external API to support.

How To Decide Which Technique Is Best?

- API Dual Write / DB Dual Write
  - Backwards compatible. Only 1 external API to support.
  - Beware flexible APIs that allow arbitrary “queries”.

How To Decide Which Technique Is Best?

- API Dual Write / DB Dual Write
  - Backwards compatible. Only 1 external API to support.
  - Beware flexible APIs that allow arbitrary “queries”.
  - If external API has diverged from database model, also beware.

How To Decide Which Technique Is Best?

- API Dual Write / DB Dual Write
  - Backwards compatible. Only 1 external API to support.
  - Beware flexible APIs that allow arbitrary “queries”.
  - If external API has diverged from database model, also beware.
  - At the end of a very long upgrade cycle, now you have 2 things that work the same way as your 1 old thing.

How To Decide Which Technique Is Best?

- API Dual Write / DB Dual Write
  - Backwards compatible. Only 1 external API to support.
  - Beware flexible APIs that allow arbitrary “queries”.
  - If external API has diverged from database model, also beware.
  - At the end of a very long upgrade cycle, now you have 2 things that work the same way as your 1 old thing.
  - API vs. DB choice depends on details of the system.

How To Decide Which Technique Is Best?

How To Decide Which Technique Is Best?

- Build new, share DB, wait:

How To Decide Which Technique Is Best?

- Build new, share DB, wait:
  - Allows a different kind of forward progress with the product.

How To Decide Which Technique Is Best?

- Build new, share DB, wait:
  - Allows a different kind of forward progress with the product.
  - Sharing databases is tricky and brittle and hard to walk back.

How To Decide Which Technique Is Best?

- Build new, share DB, wait:
  - Allows a different kind of forward progress with the product.
  - Sharing databases is tricky and brittle and hard to walk back.
  - If you share a database, do you also share a cache?

How To Decide Which Technique Is Best?

- Build new, share DB, wait:
  - Allows a different kind of forward progress with the product.
  - Sharing databases is tricky and brittle and hard to walk back.
  - If you share a database, do you also share a cache?
  - Actual cutover might be tricky, or require downtime.

How To Decide Which Technique Is Best?

- Build new, share DB, wait:
  - Allows a different kind of forward progress with the product.
  - Sharing databases is tricky and brittle and hard to walk back.
  - If you share a database, do you also share a cache?
  - Actual cutover might be tricky, or require downtime.
  - Minimal incentive to “finish”.

How To Decide Which Technique Is Best?

- Build new, share DB, wait:
  - Allows a different kind of forward progress with the product.
  - Sharing databases is tricky and brittle and hard to walk back.
  - If you share a database, do you also share a cache?
  - Actual cutover might be tricky, or require downtime.
  - Minimal incentive to “finish”.
  - Probably don’t do this unless you are really sure it’s what you want.

Ways to Make Migrations Easier

Ways to Make Migrations Easier

- Maybe don’t use an ORM.

Ways to Make Migrations Easier

- Maybe don’t use an ORM.
- Maybe don’t use a SQL database:

Ways to Make Migrations Easier

- Maybe don’t use an ORM.
- Maybe don’t use a SQL database:
  - With complex queries and lots of joins

Ways to Make Migrations Easier

- Maybe don’t use an ORM.
- Maybe don’t use a SQL database:
  - With complex queries and lots of joins
  - With a single primary

Ways to Make Migrations Easier

- Maybe don’t use an ORM.
- Maybe don’t use a SQL database:
  - With complex queries and lots of joins
  - With a single primary
- Think carefully about mixing event-based systems with RPC-based ones.

Ways to Make Migrations Easier

- Maybe don’t use an ORM.
- Maybe don’t use a SQL database:
  - With complex queries and lots of joins
  - With a single primary
- Think carefully about mixing event-based systems with RPC-based ones.
- Try to use “good” abstractions that give you leverage.

Good Abstractions

Good Abstractions

A good abstraction or interface is one that allows either side to change something without requiring coordination or changes to the other side.

Good Abstractions

A good abstraction or interface is one that allows either side to change something without requiring coordination or changes to the other side.

- This gives teams tremendous leverage and safety to make changes.

Good Abstractions

A good abstraction or interface is one that allows either side to change something without requiring coordination or changes to the other side.

- This gives teams tremendous leverage and safety to make changes.
- Beware of claims that something comes “for free”. Over time this is generally never true.

Good Abstractions

A good abstraction or interface is one that allows either side to change something without requiring coordination or changes to the other side.

- This gives teams tremendous leverage and safety to make changes.
- Beware of claims that something comes “for free”. Over time this is generally never true.
- Fun exercise: apply this definition of “good” to interfaces you work with.

Good Abstractions

A good abstraction or interface is one that allows either side to change something without requiring coordination or changes to the other side.

- This gives teams tremendous leverage and safety to make changes.
- Beware of claims that something comes “for free”. Over time this is generally never true.
- Fun exercise: apply this definition of “good” to interfaces you work with.
  - Is GraphQL good?

Good Abstractions

A good abstraction or interface is one that allows either side to change something without requiring coordination or changes to the other side.

- This gives teams tremendous leverage and safety to make changes.
- Beware of claims that something comes “for free”. Over time this is generally never true.
- Fun exercise: apply this definition of “good” to interfaces you work with.
  - Is GraphQL good?
  - How about Kafka?

Why Is This All So Hard?

Why Is This All So Hard?

- Everyone builds their own custom tooling to solve this problem.

Why Is This All So Hard?

- Everyone builds their own custom tooling to solve this problem.
- Hard to justify investing in good abstractions vs. product features.

Why Is This All So Hard?

- Everyone builds their own custom tooling to solve this problem.
- Hard to justify investing in good abstractions vs. product features.
- Cultural bias against premature optimization.

Why Is This All So Hard?

- Everyone builds their own custom tooling to solve this problem.
- Hard to justify investing in good abstractions vs. product features.
- Cultural bias against premature optimization.
- Learned helplessness creeps in.

Why Is It REALLY So Hard?

Why Is It REALLY So Hard?

- We don’t typically reward people for doing good work in this area.

Why Is It REALLY So Hard?

- We don’t typically reward people for doing good work in this area.
- Senior engineers don’t engage because they perceive it as bad for their careers, or maybe they just don’t find it interesting.

Why Is It REALLY So Hard?

- We don’t typically reward people for doing good work in this area.
- Senior engineers don’t engage because they perceive it as bad for their careers, or maybe they just don’t find it interesting.
- Senior engineers should be all about offering leverage from their time.

Why Is It REALLY So Hard?

- We don’t typically reward people for doing good work in this area.
- Senior engineers don’t engage because they perceive it as bad for their careers, or maybe they just don’t find it interesting.
- Senior engineers should be all about offering leverage from their time.
- At this point in our industry, this migration problem should not exist.

Thank You