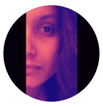
# How we broke up our Monolithic Django Service into microservices


Barkha Agrawal
Dec 6, 2017 · 5 min read

> *Microservices — the in-trend architecture which gives a really practical approach for easing the pain of having to maintain a mammoth-sized monolithic application, but as with everything else in this world, comes with its own unique set of issues to solve.*

Firstly, how do you know if your application is too big to be maintained as a whole? Well, do you encounter the following problems frequently?

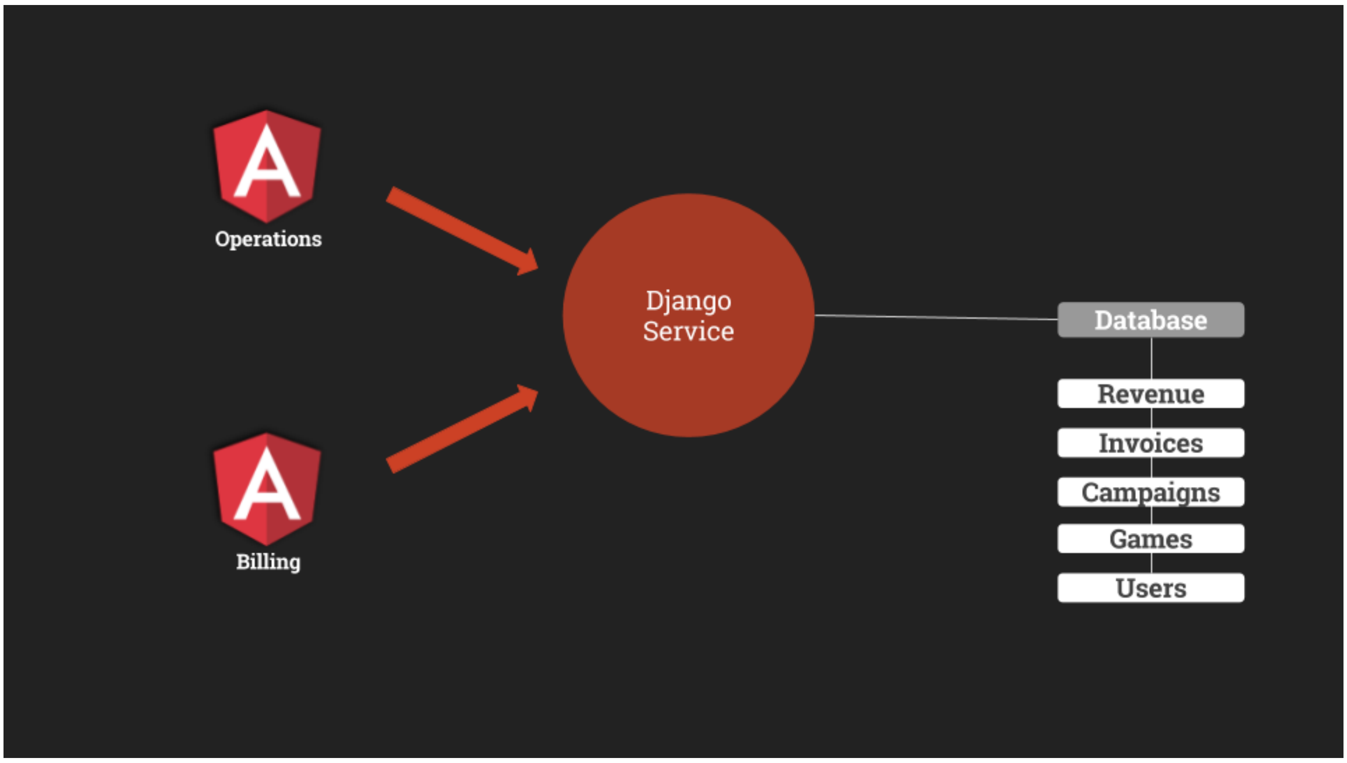
- Is your code update cycle frequent?
- Does the code size enforce a limit on your implementation of a new feature or scaling up an existing one?
- Does the code update often disturb the functionality of other parts of your application?

Well, if your answer is yes to all above, you should think about breaking your application into smaller parts.

Whether microservices is the way you should go about splitting your application is another question entirely. I believe you should go for microservices only when you find that you want to scale up only particular functionalities of your application AND you can clearly outline the boundaries and the dependencies of each part you would want to break your application into. There are a number of ways how Microservice Architecture could do more harm than good in your case. You can go through this blog for answering the how? and really?

It's important to note that if you do decide to take the plunge, be prepared for a lot of work into re-design, setting up infra for new service and the code splitting to generate different services. Testing and monitoring resultant services should definitely not be an afterthought.

GreedyGame serves native advertisements in mobile games, using a number of microservices in the process. We had a single gargantuan Django service catering to the functions of Operations and Billing. There were frequent updates in Operations-related functions than the ones concerned with Billing. As the codebase for both was the same, any changes in a model related to both, had to consider corner cases of both Applications. Any feature push would require a testing of both.



## Challenges when breaking an existing service

> *When you envision and develop an application, you basically think of it as a complete organism. To split it off into parts, at first glance, seems unimaginable.*

**Defining Boundaries of Each Service**

The objective of this exercise is to recognise a factor which would act as a key to decide which service, a certain feature should belong to. The easiest way to figure that out would be looking back to why you decided to split your services, the driving business logic which is making you take this step. For our part, we figured user-base of the application as the factor to separate the models and features between Operations and Billing.

## Separating Resources for resulting Services

> *Any proverbial microservices expert would tell you that sharing resources between two services is going against the complete philosophy to have* independent *deployed smaller services.*

Apart from infrastructure and deployment, there is the matter of other shared resources, such as database. Fortunately for us, as we use a 2NF schema in MySQL, the splitting of tables into two different databases became easier. I imagine it'd be a harder job in case of a NoSQL, Key-value, Graph or other such databases.

## Inter-service Communication

There could be multiple ways you can go about solving this problem. This article provides a good introduction to various solutions to the problem of inter-service communication. The route we took was to write two internal microservices atop our original split services, which would use HTTP calls and will be dedicated to sharing required data with other services.

## Choice of Language and Framework

> *One of greatest advantage of following a microservice architecture is that you can choose to write an individual service in whichever language or framework you think would suit your service, and of course you, the best.*

We, however in this case, chose to stick with Python. Why? Two words. Occam's razor. As both the Django service and the new microservice are part of same project, it'd be easier, for any new developer coming in, to maintain the entire project, without having to have a grasp over python + another language.

As the scope of these microservices extended only to information sharing, requirement was for a lighter framework than Django. Django provides a lot of added features — Django ORM, Admin Interface, Authentication/Authorization and so on — which are not really required for this microservice. We chose to go with Flask as, on top of being a lighter framework with well-supported community, it does not compromise on code modularity.

Here's a sample of what the first draft of app.py looked like:

```python
import os
import traceback
from datetime import datetime

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import desc
from marshmallow import Schema, fields
from flask_mail import Mail, Message

from settings import ProductionConfig, TestingConfig

app = Flask(__name__)

PRODUCTION = 'production'
TESTING = 'test'

FLASK_MODE = os.environ.get('FLASK_MODE', TESTING)
if FLASK_MODE == TESTING:
    app.config.from_object(TestingConfig)
else:
    app.config.from_object(ProductionConfig)

db = SQLAlchemy(app, session_options={"autoflush": True})

class Invoice(db.Model):

    __tablename__ = "invoice"

    id  = db.Column(db.Integer, primary_key=True)
    publisherid = db.Column(db.Integer)
    amount = db.Column(db.Float)
    file_name = db.Column(db.String(300))
    status = db.Column(db.Integer)
    is_published = db.Column(db.Boolean)
    created_at = db.Column(db.DateTime)
    invoice_date = db.Column(db.DateTime)

class InvoiceSchema(Schema):
    id = fields.Integer()
    amount = fields.Float()
    file_name = fields.Str()
    status = fields.Integer()
    created_at = fields.DateTime('%Y-%m-%d %H:%M:%S')
    invoice_date = fields.DateTime('%Y-%m-%d %H:%M:%S')

invoices_schema = InvoiceSchema(many=True)

@app.route('/invoice/<string:pk>', methods=['PUT'])
```

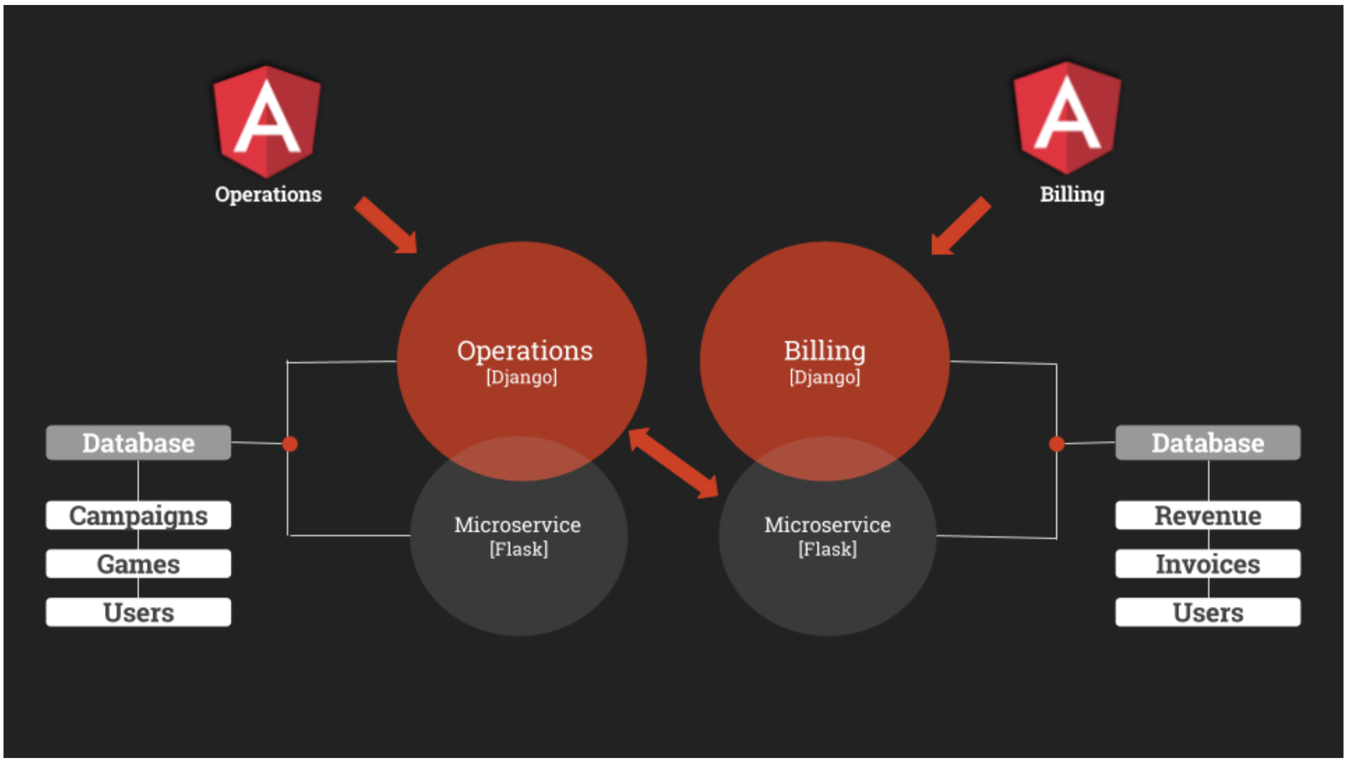
```
@app.route('/invoice/<string:pk>', methods=['PUT'])
def update_invoice_status(pk):
    invoices = Invoice.query.filter_by(publisherid=int(publisher_id), is_published=True).order_
        return invoices_schema.dump(invoices).data
    return jsonify(invoice)

mail = Mail(app)

@app.teardown_request
def teardown_request(exception):
    if exception:
        db.session.rollback()
        if FLASK_MODE == PRODUCTION:
            trace = traceback.format_exc().splitlines()
            msg = Message("ERROR: {}".format(exception), recipients=["concernedparties@greedyga
            msg.html = "<br>".join(trace)
            mail.send(msg)
    db.session.remove()
```

Voila! The microservice was up and running.

## The Final Picture



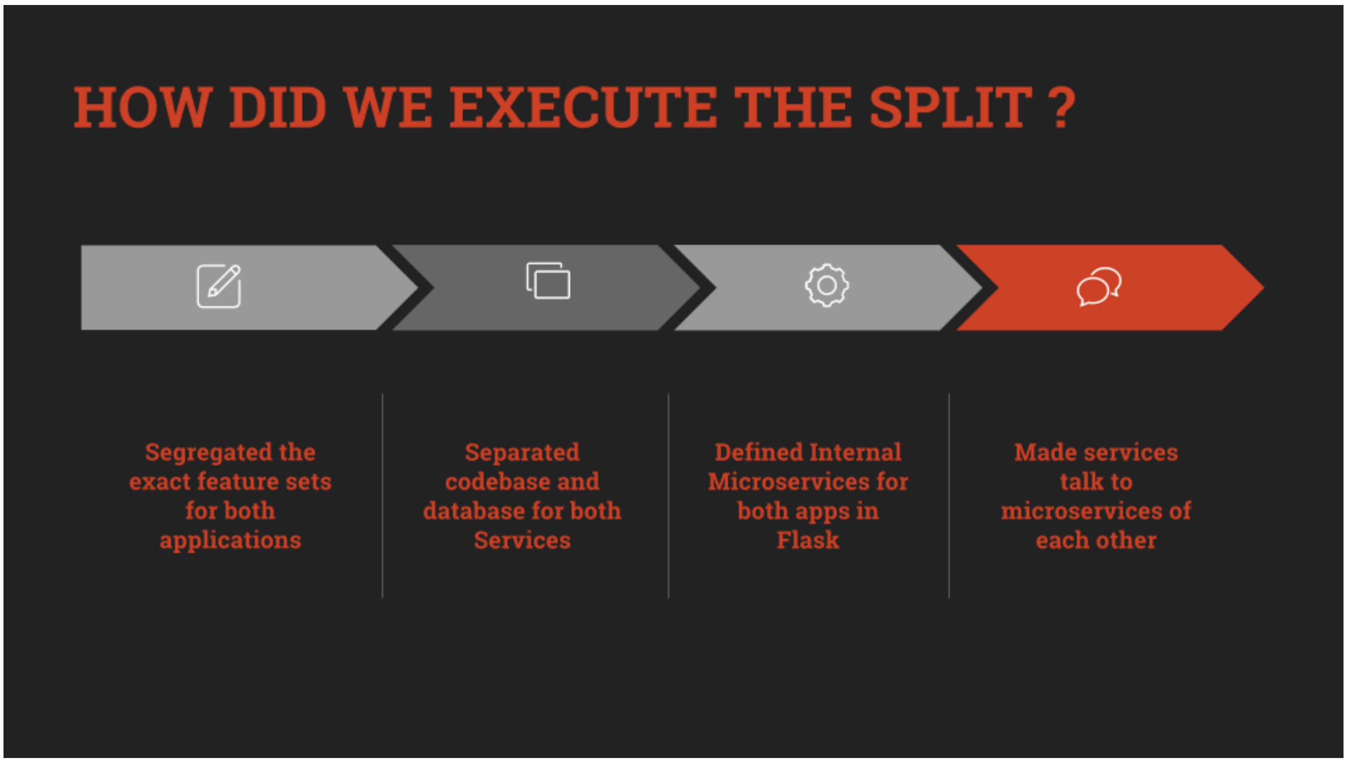
This made the whole architecture much more simpler to maintain and to extend. So, any information that Operations needs from Billing comes from the Billing's Flask microservice and vice-versa. This solved all the aforementioned problems and enabled us to roll out smoother releases for either application.

## Quick Summary

The first and foremost step when deciding to split a large back-end service is to describe the boundaries of each of the resulting smaller services.

Once we knew what feature was required in which service, we simply shifted the code for same to respective service's repository.

The final part was to device a mechanism to share resources between services. We found the solution for it in developing internal microservices in Flask.



HOW DID WE EXECUTE THE SPLIT ?

- Segregated the exact feature sets for both applications
- Separated codebase and database for both Services
- Defined Internal Microservices for both apps in Flask
- Made services talk to microservices of each other

And that's all Folks! Feel free to leave a comment for any doubts or a quick discussion.

Thanks to Arink Verma and Siddharth Gupta.

Microservices Django Backend Flask