Cómo matar tu viejo Django y dejar crecer a sus hijitos

esLibre 2019 - Granada

Tomás Garzón

Hablaremos de...

¿Esto de que va?

Contexto y Motivación

Arquitectura 1.0

Rediseñando la arquitectura (2.0)

Lecciones aprendidas

Camino por recorrer

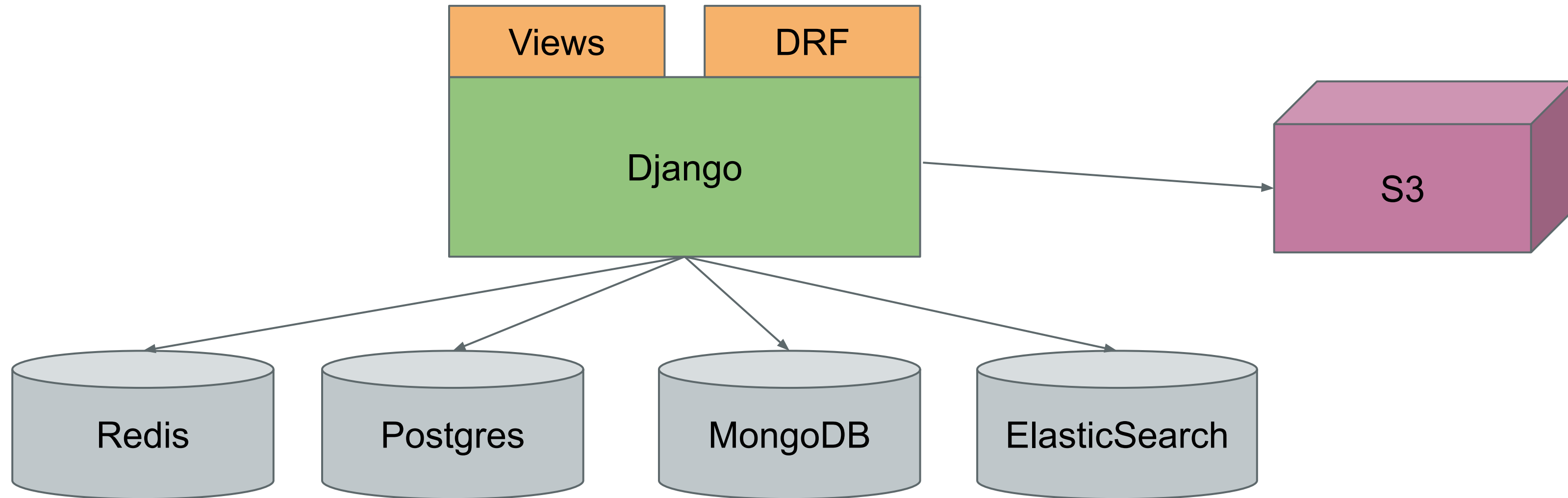
¿Esta charla de qué va?

- Viejo not → algo malo o peyorativo.
  - Código viejo:
    - Cuesta más moverse o adaptarse a cambios.
    - Cuesta más de mantener su calidad
    - Cuesta mantenerlo actualizado.
- Matarlo not → violencia :)
  - Transformación hacia algo algo nuevo desde cero manteniendo su “espíritu”
- Hablaremos cómo transformar una plataforma web monolítica hacía una arquitectura basada en servicios.
  - Basado en nuestra experiencia.
  - Centrándose en la parte backend y en Django-Python como lenguajes y framework de referencia

Motivación

- Entorno de startup: Búsqueda de un modelo de negocio
    - Cambios constantes en funcionalidades.
    - Poco margen para la calidad y la refactorización.
    - Parches y cambios de última hora.
    - Ausencia de procesos de release y validación.
    - Equipos que rotan sin formación/adaptación previa

Arquitectura Monolítica



ARQUITECTURA MONOLITICA

Web browser

FRONTEND

API

django REST framework 3

HTML5 WebSockets

BACKEND

django

STORAGE

S3

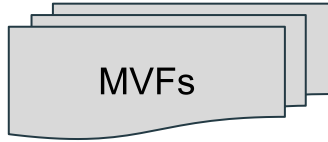
PostgreSQL

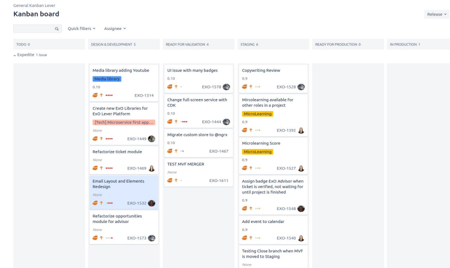
mongoDB

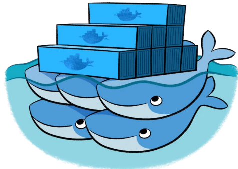
Problemas

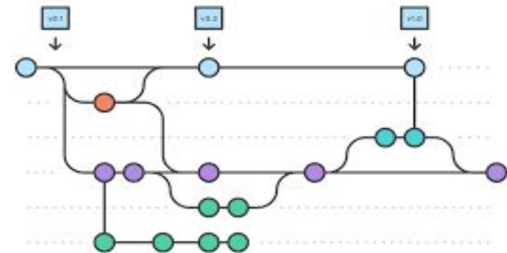
1. Proyecto con más de 100 apps (Propias y externas).
2. Librerías fijadas, no hay una política de actualización.
    a. Hacer el pip install -r requirements.txt era una aventura!
3. Muchas tecnologías que ya realmente no hacían falta
4. Limitación para meter otras librerías por versiones antiguas fijadas.
5. Dificultad para hacer despliegues desde cero o dockerizados.
6. Dependencias entre muchas apps:
    a. Mucho código conectado entre distintas apps, difícil de entender y mantener los cambios.
    b. Muchos tests con dependencias de otras apps, dificil de mantener.
    c. Gran cantidad de tests, casi 1000 que tardaban 25min en pasarse.
    d. Cualquier cambio implicaba revisar mucho código y muchos tests.
    e. En general, mucha **deuda tecnológica**.

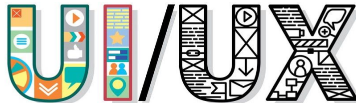
Proceso de Desarrollo (I)



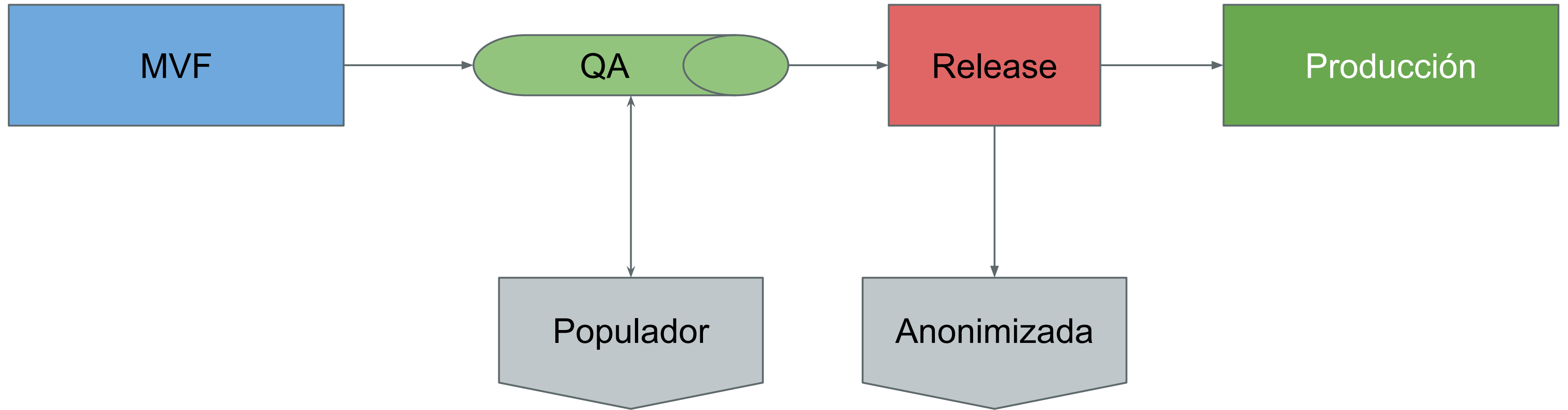








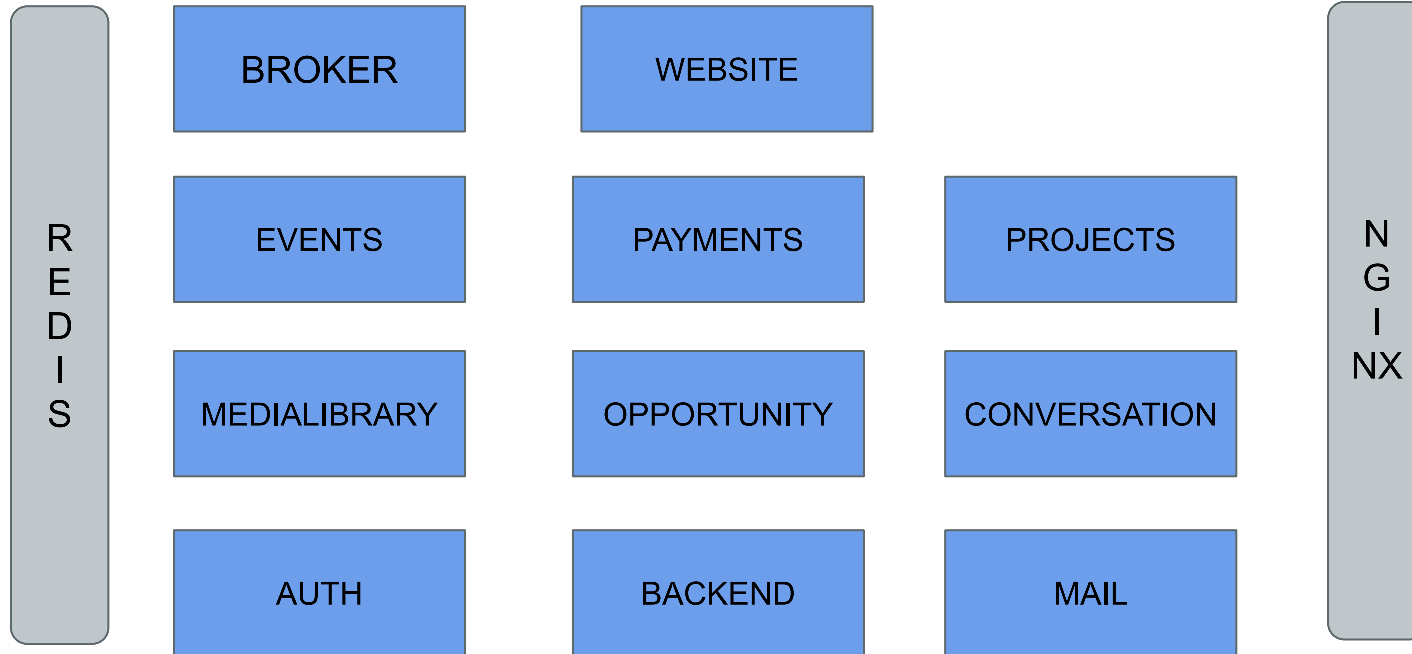
Proceso de Desarrollo (II)



Arquitectura basada en servicios

- Tener servicios más pequeños con un objetivo acotado.
  - Podríamos usar cualquier tecnología, pero usamos Django por afinidad y por requisitos.
  - Desacoplamiento de código y de datos. Cada servicio tiene sus propia base de datos.
  - Cada servicio se testea de manera unitaria
- Los servicios hablan entre sí:
  - Mediante API Rest para pedir o mandar datos.
  - Mediante modelo emisor-subscriptor, en Redis.
  - Mediante una cola de mensajes, tipo RabbitMQ

Arquitectura basada en servicios



Servicios

1. Actualmente tenemos estos servicios:
   a. Broker: Enrutado para las conexiones mediante websockets y comunicación en tiempo real.
   b. Backend: es el anterior backend monolítico, del que se han ido quitando apps y código.
   c. Opportunity: Gestiona las oportunidades y marketplace.
   d. Events: se encarga de gestionar los eventos de la comunidad.
   e. Payments: sirve para que administración u otros servicios puedan generar pagos.
   f. Website: se encarga de generar webs dinámicas basadas en Hugo.
   g. Medialibrary: gestión de recursos por parte de la comunidad, basado en filestack Vimeo
   h. Projects: información relativa a los proyectos que se ejecutan.
   i. Conversations: almacena información relativa a los sistemas de mensajería.
   j. Auth : sirve para autenticar a los usuarios y generar los token JWT con los que opera las APIs.
   k. Mail: servicio donde se gestionan los templates de todos los emails, y se envía mediante sendgrid

Homogeniza servicios

- Cada servicio puede estar desarrollado en un lenguaje o framework distinto.
- Mantener la comunicación mediante API Rest te facilita la comunicación entre servicios y con el frontend.
- Creemos que lo más productivo es tener un cookiecutter y aplicarlo a todos los servicios:
  - Permite encontrar las cosas siempre de la misma forma y en el mismo sitio.
  - Un Dockerfile similar para construir un nuevo servicio sin la necesidad de intervenir devops.
- Nosotros usamos Django para todos los servicios, por que al final necesitamos una base de datos y una API Rest, por lo que hemos optado por este framework.
- Para el broker no usamos Django, y nos basamos en el paquete socketshark.
- Referencias:
  - https://bitbucket.org/exolever/service-cookiecutter-template
  - https://github.com/closeio/socketshark

Cada Servicio

- Tiene que poder construirse como imagen docker (Dockerfile)
- Para que se construya su imagen:
  - Se pueden instalar todas sus librerías.
  - Tiene que pasar los tests unitarios.
  - Tiene que pasar la validación de pep8 y flake8.
  - Tiene que pasar la ejecución del populador de cada servicio.
- Utilizamos pipenv y no fijamos el número de versión, excepto para las releases.
- Usamos una base de datos propia (todos los servicios usan el mismo servidor de base de datos, pero cada uno con su propia base de datos).
- Solemos usar también celery para las tareas asincronas en todos los servicios.

Un solo repo

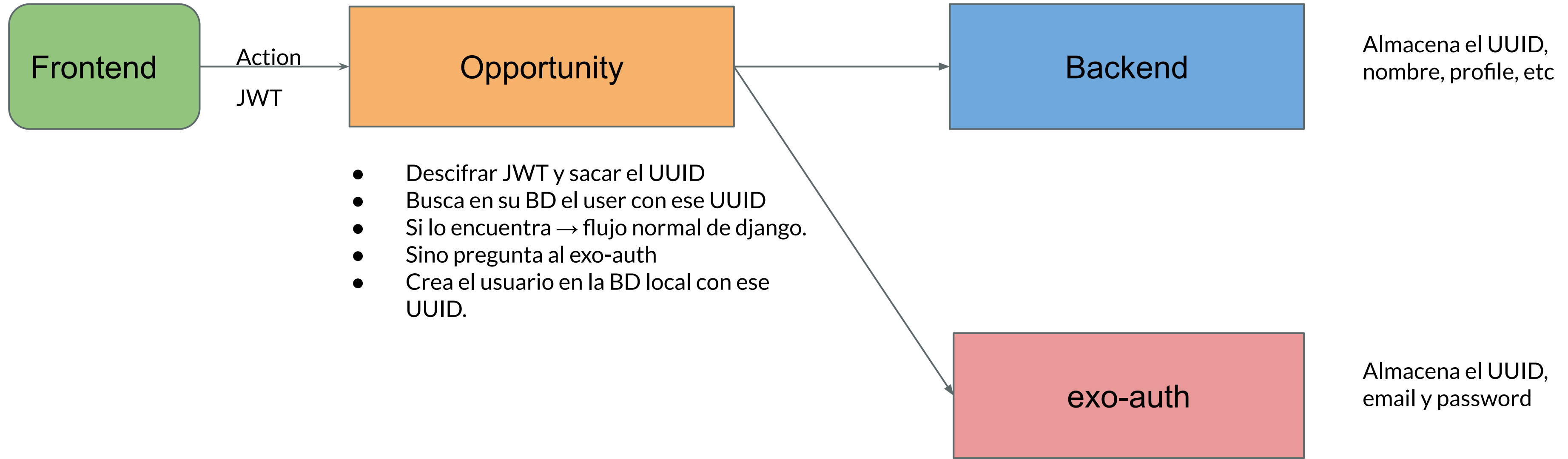
- Se podría pensar en crear tantos repos como servicios, pero nuestra experiencia nos dice que es mejor tener solo uno (si todos usan el mismo lenguaje).
- Ventaja: Cuando se hace una issue que involucra varios servicios, se necesita solo un Pull Request, y se tiene una visión global de la issue.
- Ventaja: Cambios en los servicios ya existentes es mucho más rápido
- Tenemos un pipeline que en cada paso construye las imágenes de cada uno de los servicios.

Populador

- Son ficheros YAML donde se describe la información para la base de datos, con escenarios específicos.
- Arquitectura monolítica: Un proyecto ligado al backend para generar datos reproducibles y escenarios para desarrollo, qa y demos.
- En la arquitectura de servicios esta aproximación no es válida, por que muchas operaciones requieren de otros servicios.
- Solución: cada servicio tiene su propio populador, que se lanza cada vez que se levanta el servicio.
- Importante que durante el proceso de populación no se hagan llamadas fuera del servicio, ya que los otros servicios pueden no estar levantados.
- Librería para añadir la populación a nuestro servicio:
  - https://github.com/exolever/lib-exo-populator

Usuarios y autenticación

- Limitación por Django del modelo User.
- No duplicar la información del user como nombre, email, password, profile, etc.
- Solución: Tener en un solo sitio el User con la información completa y en el resto de servicios usar una versión más simple del User.
- Entre servicios, se identifica al usuario por un UUID, ya que los pk son locales a cada base de datos/servicio.
- Las relaciones en el servicio se hacen como siempre, usando los pk del modelo user,pero cuando quieres obtener información de otro servicio de un usuario, no usas el pk sino el uuid, que es único y compartido por todos.
- Usamos JWT en todas las APIs para almacenar el UUID del usuario.
- Tenemos en todos los servicios compartida la clave de descifrado del JWT.
- Usamos el mecanismo de django para compartir la cookie de la session en redis, para todos los servicios.
- Librería que usamos en los servicios:
  - https://github.com/exolever/django-simple-user


Frontend
Action
JWT
Opportunity
Backend
Almacena el UUID,
nombre, profile, etc
Descifrar JWT y sacar el UUID
Busca en su BD el user con ese UUID
Si lo encuentra → flujo normal de django.
Sino pregunta al exo-auth
Crea el usuario en la BD local con ese UUID.
exo-auth
Almacena el UUID,
email y password

Libera código

- Saca fuera de tu proyecto aquellas partes, apps o módulos:
  - Que no sean el core de tu negocio (algoritmos, know-how, código ad-hoc).
  - Librerías que se pueden reutilizar por otros servicios, y por tanto pueden ser útiles a terceros.
- Ventajas:
  - Ese código está liberado y susceptible de ser mejorado por terceros.
  - Se puede testear y liberar de manera aislada
  - Reduces las líneas de código propias de tu proyecto y los tests de tu servicio.
  - Te ayuda a generalizar y pensar de forma más global el código, para ser reutilizado.

Usa las tecnologías necesarias

- Inicialmente teníamos MongoDB y Elasticsearch → mayor complejidad para los despliegues y para los tests.
- MongoDB: migración a postgresql de los datos y los algoritmos.
- Elasticsearch: buscamos un mecanismo auxiliar de indexado.
- Realmente para el enfoque actual (startup) ya no eran necesarias.

Gestión de releases

- Nosotros tageamos con el mismo número de release todas las imágenes de docker de todos los servicios, aunque no incluyan cambios.
- Para evitar sorpresas, una vez que QA tagea una release, ese pipeline fija el número de versión de todos los paquetes python que se están usando.
- Usamos una app para ejecutar en el servidor de producción aquellos cambios de datos que hace falta hacer (evitamos tener que hacer cambios a mano después de cada release por parte de un desarrollador o devops).
- Librería que usamos
  - https://github.com/exolever/django-changelog → Inspirada en las migrations de Django.

Migración en modo blue-green o silencioso

- O Cómo no matar a tu devops en cada release.
- Lo usamos cuando:
  - Sacamos una cierta funcionalidad desde el monolítico hacia un servicio.
  - Cambiamos de base de datos (sqlite a postgres).
  - Hacemos un cambio muy importante.
- Consiste en sacar la funcionalidad del servicio en varias partes.
- Primero se mete el servicio en el pipeline, se levanta la imagen y se ve que está ok.
- Segundo se hace un script de migración que se ejecute diariamente para migrar los datos del monolítico hacia el servicio. Puede fallar, pero no pasa nada por que nadie lo usa.
- Se pueden dar más iteraciones para ir añadiendo features que faltan.
- Cuando frontend cambia para usar la nueva API, en ese momento el servicio es cuando se está usando realmente.
- Incluimos una issue adicional, para borrar el código antiguo en la siguiente release

Camino por recorrer

- Extender y potenciar el Broker:
  - Enrutador de las APIs hacia los servicios.
  - Balanceador de carga:
    - Poder levantar varias instancias de una imagen.
- Hacer una comunicación de eventos general.
  - Usando rabbitmq, cada servicio puede emitir los eventos y otros servicios suscribirse a aquello que necesiten.
  - Evitaríamos tener que hacer llamadas explícitas para modificar datos en otros servicios.
- Incluir un anonimizador por servicios.

Testing

- Hacer TDD para features, bugs y hotfix.
- Tests automatizados:
    - Unitarios a nivel de app de Django.
    - Unitarios a nivel de servicio.
    - De integración entre servicios.
    - Comparación de imágenes.
- Tests manuales sobre un despliegue en QA
- Los unitarios nunca hacen llamadas externas.
- Test de integración en el broker.

Gracias

Tomás Garzón Hervás
tomas@openexo.com
tomasgarzonhervas@gmail.com