SOCOTEC Archi MicroServices

MediaCorpus & Siilk LLC

29 Mars 2021

Guillaume O'Neill

Version 2.0

MDM Agenda

- Goals
- Ressources
- API Tools
- MicroServices the Basis
- Django REST Framework API in depth
- Question on How Socotec will use API
- Django API structure – How to start ?
- Tools to be Discover
- Socotec Specification Guide

MicroServices and Django / Python

- Django RestFrameWork API (DRF)
- Django REST Swagger (Rich User interface for DRF)
- FastAPI (Possible Integration with Django)
- Customized API server(Home Made)
- MicroServices Admin Interface (Django, Customized, ….)

MicroServices the Basis

- **Shared Databases**
  - Supervision, Settings, Logging, Metrics
  - Adding New Microservices Functions
  - Errors Tracking
- **Application Databases**
- **Access Token**
- **Common Public Gateway**
  - URL Structure et Patterns
- **Querying and Data Management**
  - API Router
  - Python Application
- **Admin Tools**
- **Messaging**

DRF the Basis

- Installation
- Permission
- Serializer
- Requests and Responses
- API Hyperlinks
- DRF Views
- Json component
- Status Codes

API and Socotec

- **The Good questions**

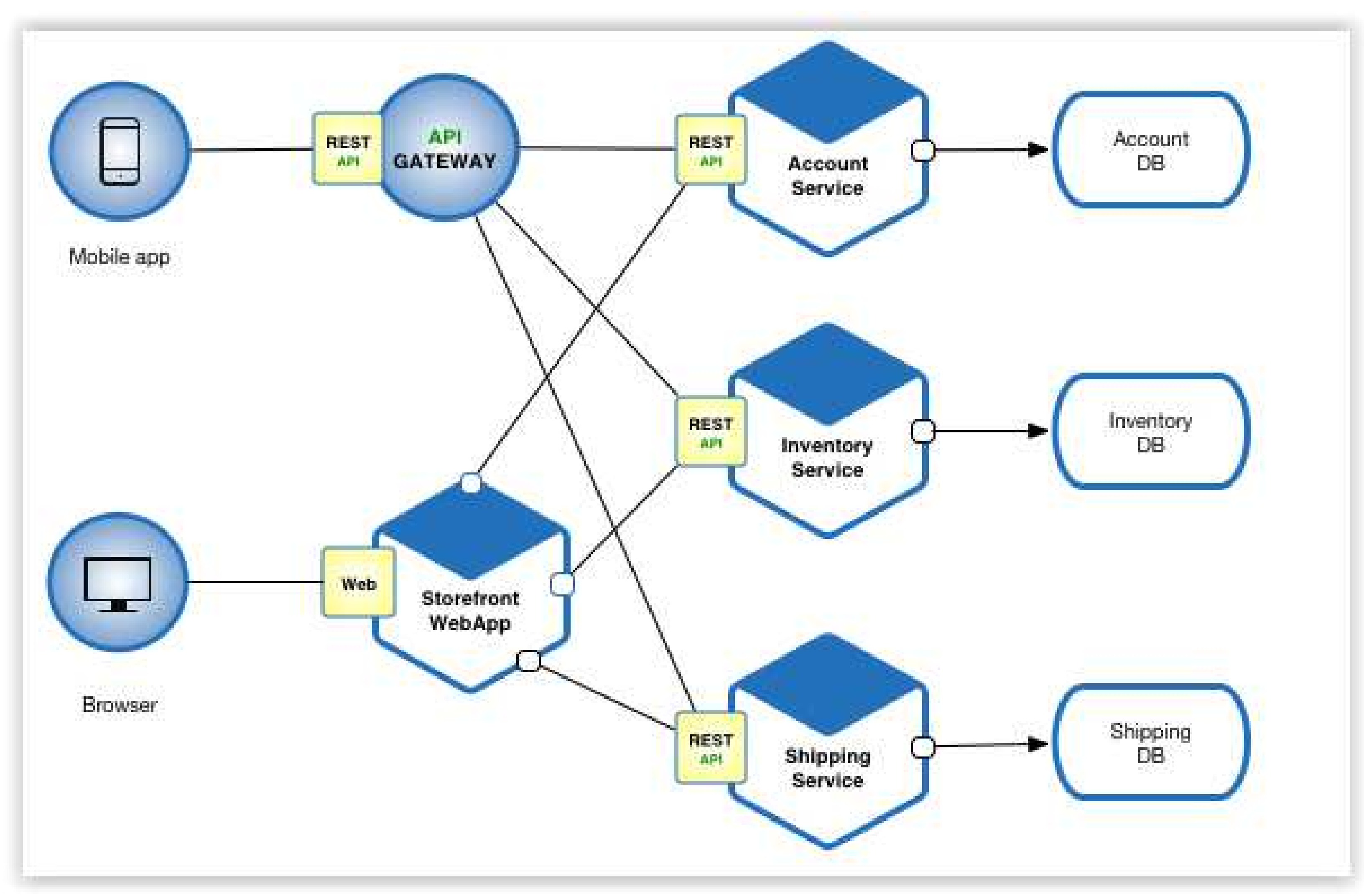
  - Who will use the API / MicroServices (End User) ?
  - Which frontend will access the API (React / Angular, Vue, …)
  - Data Model Architecture (actual and Future)
  - Languages used to Call Socotec MicroServices (Python, PHP, Mobile, Kotlin, …)
  - About Security (Intranet / extranet)
  - Platforms :
    - PC / Laptop / Tablet / Mobiles

Django API Architecture

- **Primary Rules:**
  - Define Communication protocols and Gateway
    - URL, Json, Data transport, serialization
    - Objects transport
    - Return Code
    - Frontend Integration

  - Django API Settings
  - REST API Database Model :
    - Supervision
    - Logging
    - Auth/Token Management
    - Traffic
    - Parameters
    - Errors (Cf Logging)
    - Statistics

Django Architecture



Django API Architecture - 2

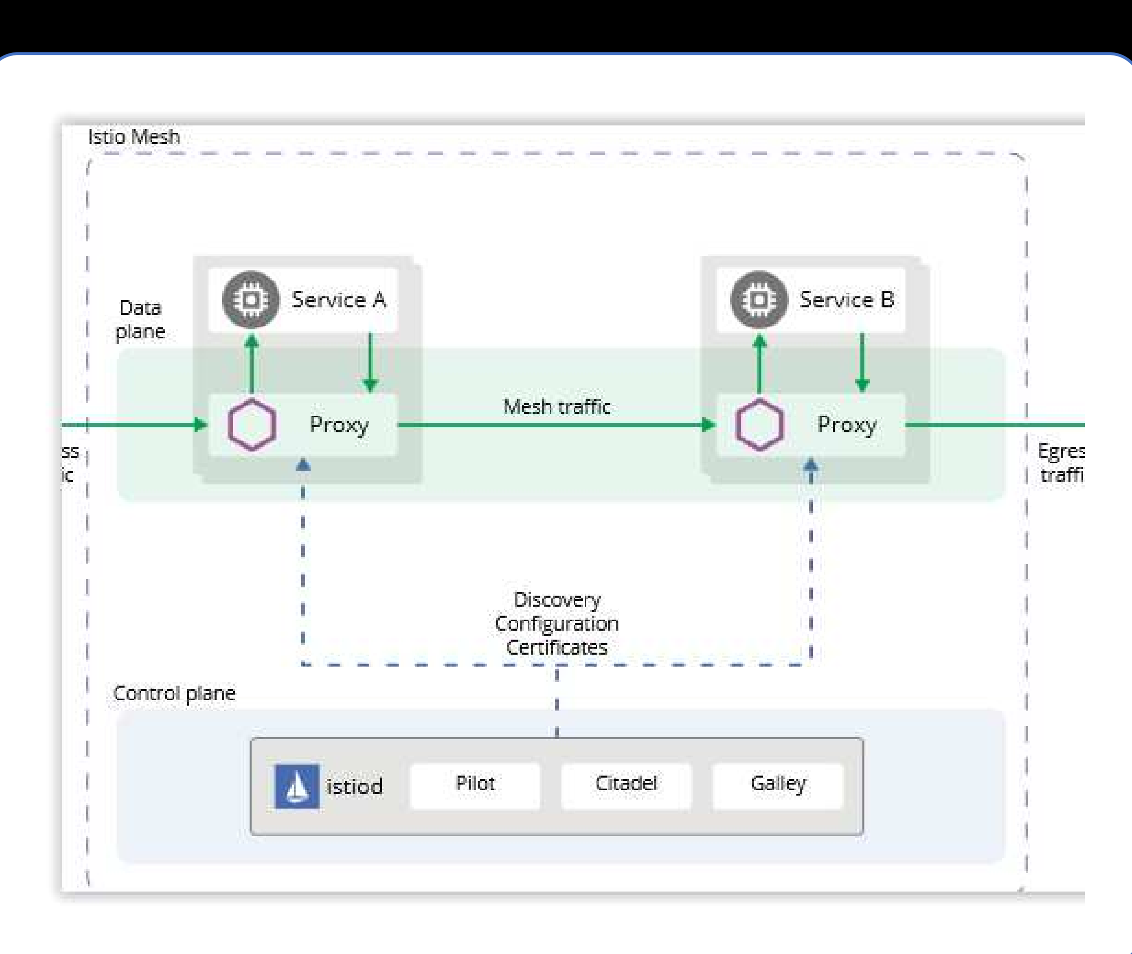
- **Common Interfaces:**
  - One Rest API for each Platform or a Common Gateway ?

Take a Look at Socotec System Tools

- Istio
- Rancher

Istio advantages

- Simply Cloud Managing and Monitoring
- Load Balancing
  - HTTP, TCP and Websocket
- API support
- Metrics and Logs
- Advanced Security



Socotec Specifications Guide
Version 1.0

Specification Agenda

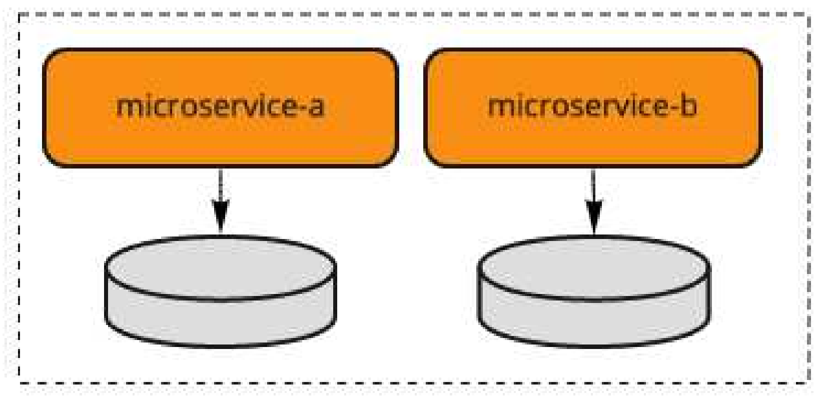
- Storage and Database
- Communication protocol
- API Gateway
- Automation
- Event Sourcing
- CQRS
- Requirements
- API Basis
- Authentication & Token
- Errors management
- Logging
- Scaffolding & Online API Documentation
- External Links
- API Usage

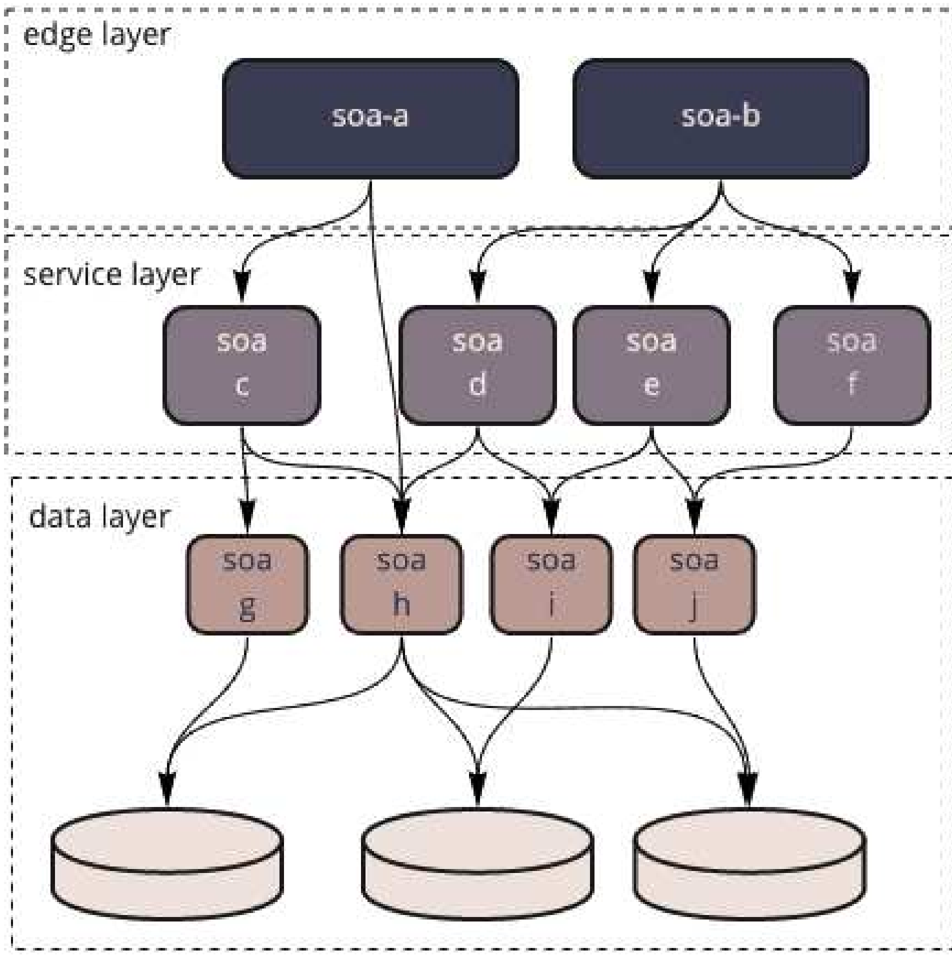
Storage & Databases

DataBase Agenda

- Storage
- Data Supported
- SQL Server
- SQL Engine comparison
- Performances Best practices
- Django
- Database Admin

MicroServices Databases Concept





API Storage

- SQL Server :
  - PostgreSQL 13.2
  - MariaDB 10.5.9
  - MySQL 8.0.23
  - **All Oracle Licenses**

API Supported Data

Input

- Single Data
- Array & Matrix Data
- Json Object

Output

- **Single Data**
- **Array & Matrix Data**
- **Json & XML Object**
- **Binary Object**
- **Graphical Object (MatPlotLib)**
- **PDF Object**
- **Flat Files Object**

DataBase Localization and Deployment

- API Monitoring and Administration :
  - Shared
- API Application services :
  - Dedicated Databases (one per Application Group)
  - Partitioned
- SQL Server
  - Dedicated Server and settings
  - Dedicated ECU for RDS (as AWS or Google)
  - 128 Gb of memory
  - SSD Premium
  - Cloud Services **(the must** !)

DB Comparison

	PostgreSQL	MariaDB
Performance	**High performing +30%**	Good with Low Traffic
Data Type Support	Show errors !!! Strict	**Good Compliance**
Features - Extended	Materialized Views, Partial Indexes	**Former Replication, JSON, JSONB**
Size	SAME	SAME
Partitioning	**NONE**	Spider Storage / Galera / Horizontal
Make The Choice !	Business – Popular – Complex	Popular – Simple !
Yes If	Advanced Features and Speed but needs advanced experts to take benefits !	Simple – Fast enough for 80% of Website !

API Microservices – SGBD Best Practices

- One volume for Shared Database (x1)
- One Volume for API services (xNnnn)
- One Database per Application group
  - Django allows easily to manage a Multiple Database configuration
  - DataBase Router capabilities
- One Database for each consistent Data group
- Table and Index Partitioning
- Indexes ?
  - Yes, but on High cardinality
  - Make SQL Explain
  - Pay attention to Joins at Multiple Levels
  - Foreign Key, Many To Many Choices ? With Django it's 2x Tables More !!
  - Sort requests

API Databases ADMIN

- Django DataBase Admin Tool ?
- Customized Databases Application (Socotec) ?
- Replication
- Backup
- POC (Point of Consistency)
- Recovery

Shared Databases Contents

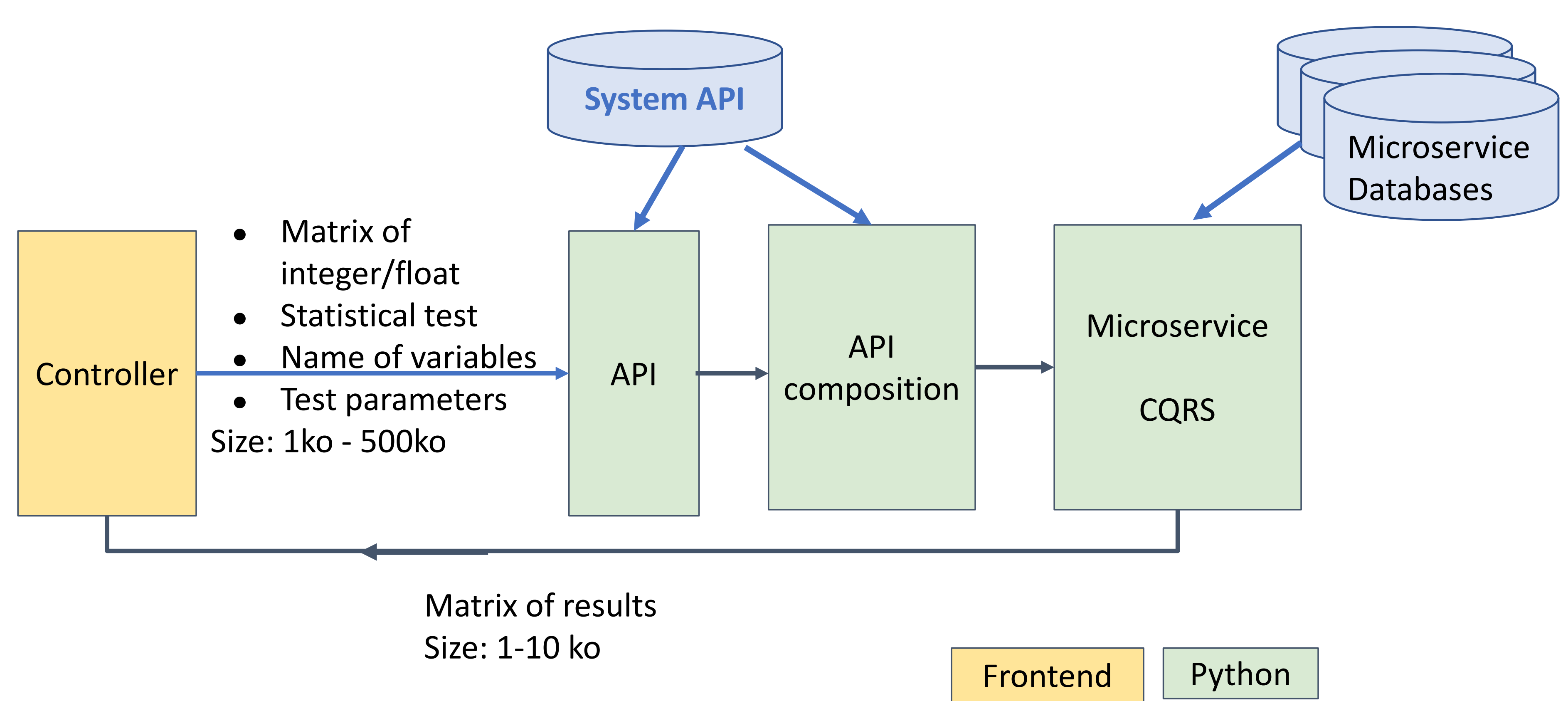
- Shared DataBase (Table's definition) :
  - Logging
  - API Definition
  - URL Definition
  - Token Management
  - API MicroServices Cross References
  - API execution and Completion

API Dedicated Databases Contents

- Dedicated API DataBase (Table's definition) :
  - LUP (Logical Unit Process)
  - API Process Cross Reference
  - API Data Flow (Input / Output) Temporary
  - API Process identification and definition
  - API Documentation & Description
  - API Options, Values & Parameters
  - API Templates

Databases Diagram Flow

MicroServices Tests



Communication Protocol

Protocol Agenda

- API End User
- Data Flow
- Protocol Alternatives
- Security & Permissions
- URL Structure
- Database & Admin
- Nginx http/2
- GRPC and JS Client
- gRPC Logging

API – Who are the end USERs ?

- Program : Python / PHP Application
- Platforms : Laptop / Telephone / Tablet
- OS : Windows , OSX, Android, IOS
- External : Website (Iframe / Widgets) cross URL/cross Company
- Audience : Public / Socotec / Backend / Partner companies

API – Which Data to be sent and received ?

- Data Imbedded into URL
- Temporary API Database
- Django Session
- Media - externalized

Real Protocol Alternatives

- HTTP1 / REST (GET / POST)
- WebSocket
- HTTP2 - GRPC

Protocol's comparison

Nom	Protocol	Avantages	Inconvénient
REST	HTTP/1	- Simple (connu de tous) - Scalable - Outils complet autour	- Lent (de par l'ouverture de connexion à chaque requete) - Uni directionnel
PubSub	Plein (ex: MQTT)	- Gère de gros volume de données - Gère de nombreux cas (retry on error, restart in process …)	- Pas adapté pour les end user -> nécéssite support de deux api différentes - Nécessite un serveur bottelneck
Websocket	HTTP/1.1	- Très simple à utiliser - Peut être utiliser directement par les utilisateurs	- Difficilement Scalable - Nécessite un service bottleneck ou la multiplication de clients à maintenir
GRPC	HTTP/2	- Super rapide - Décentralisé - Supporte la communication asynchrone	- Encore assez recent -> peu d'outils

gRPC

Google RPC

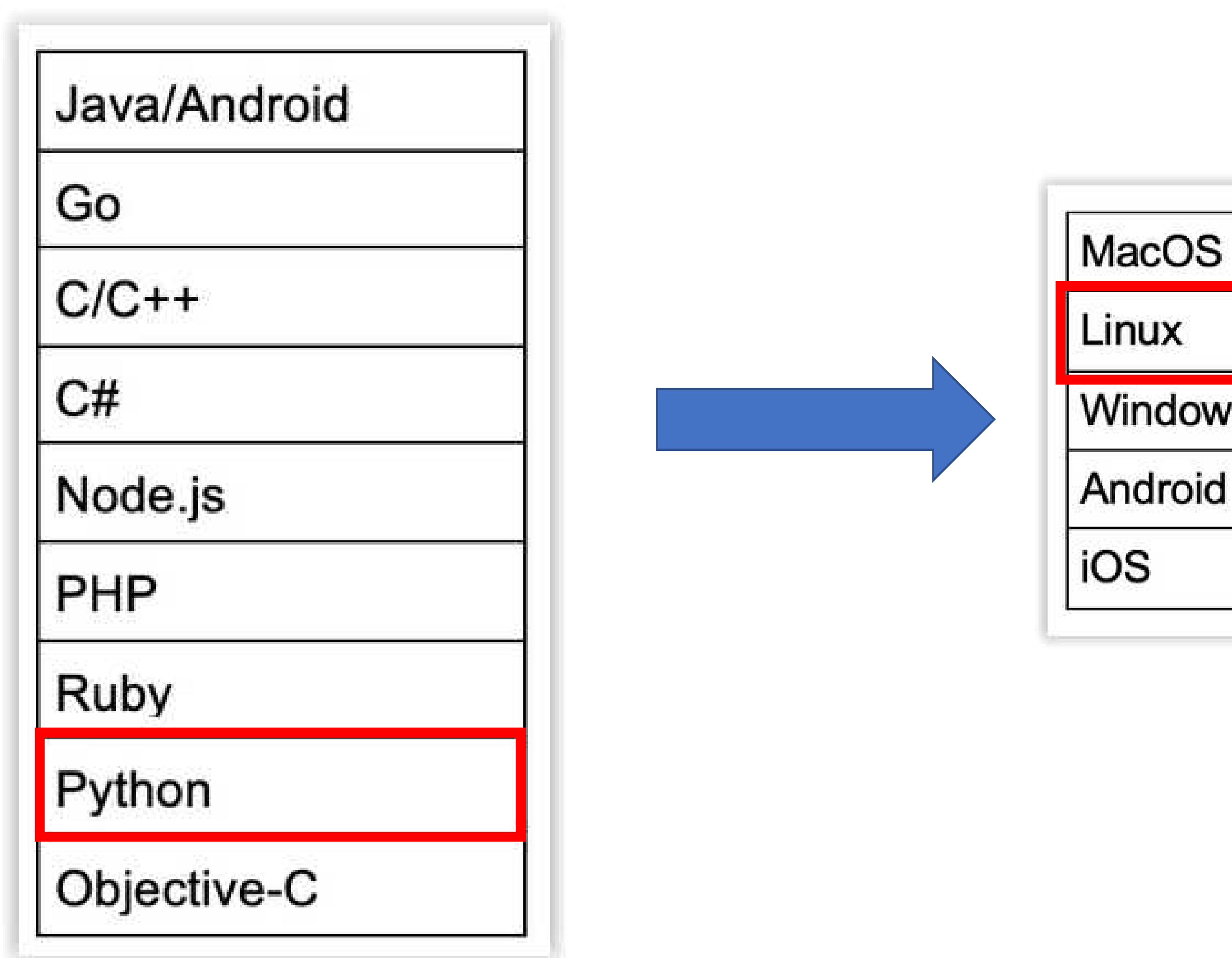
&

Asyncio

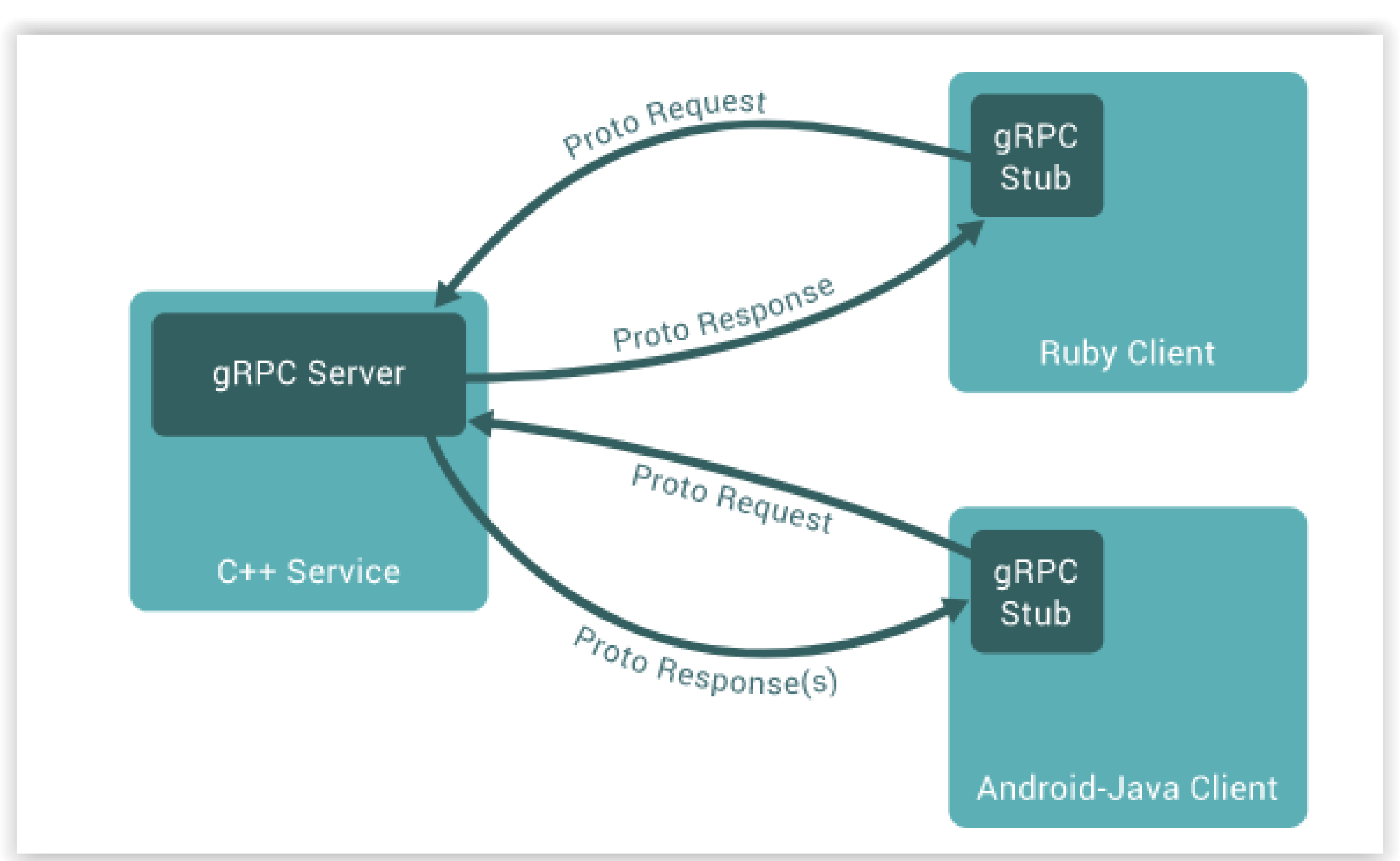
GRPC - Advantages vs Disadvantages

Advantages	Disadvantages
Functional rather than resource-based design	**Lack of consistent error handling**
Reduced network latency	**Lack of developer tooling**
Bi-directional support: HTTP2	**Lack of infrastructure and monitoring support outside of GKE**
Code generation	**Limited insight into common practices**
Documentation	**Lack of edge caching**
Infrastructure support	**Lack of support for additional content types**

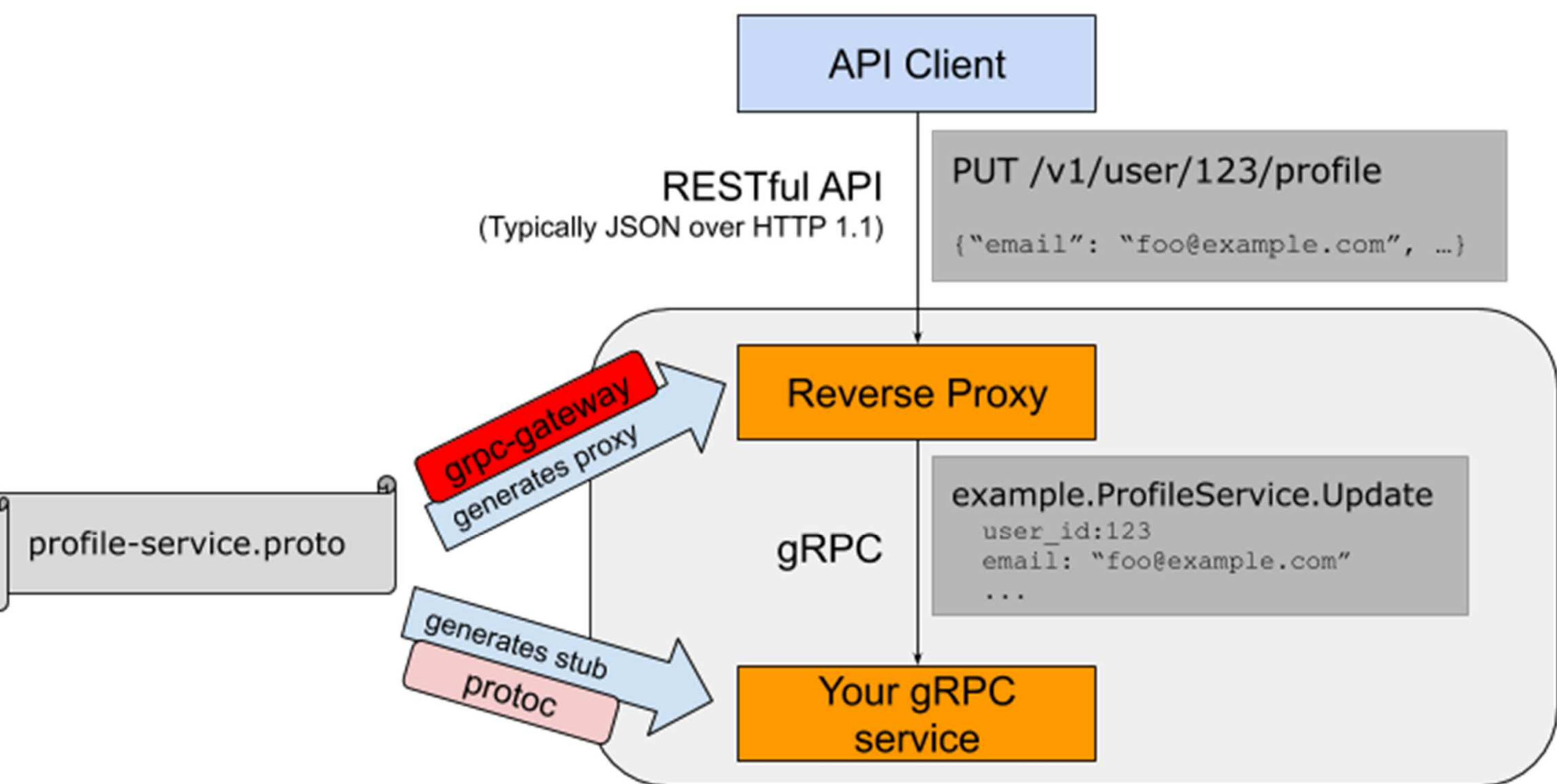
GRPC - Supported Platform



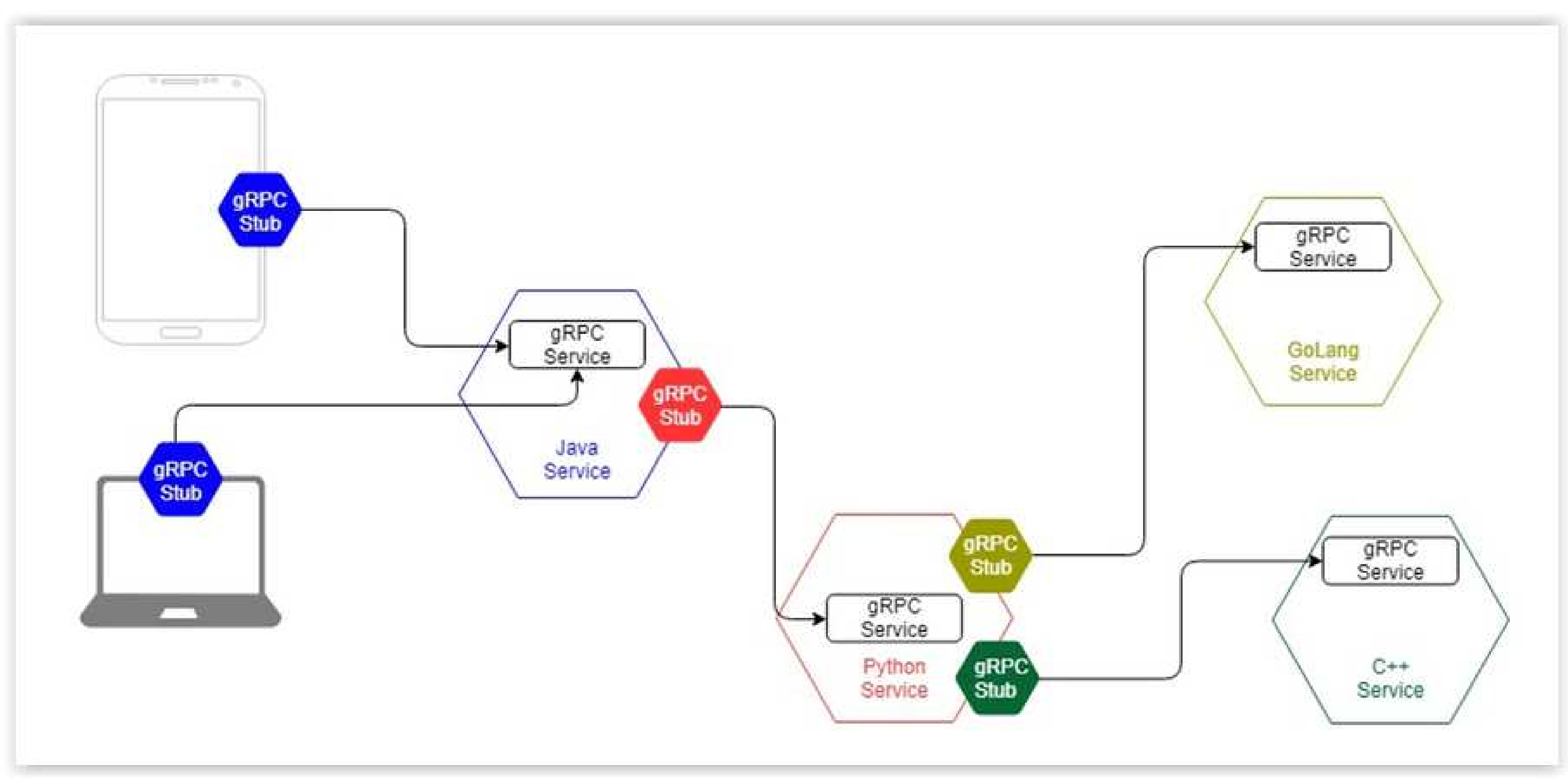
GRPC Flow Diagram



gRPC and API

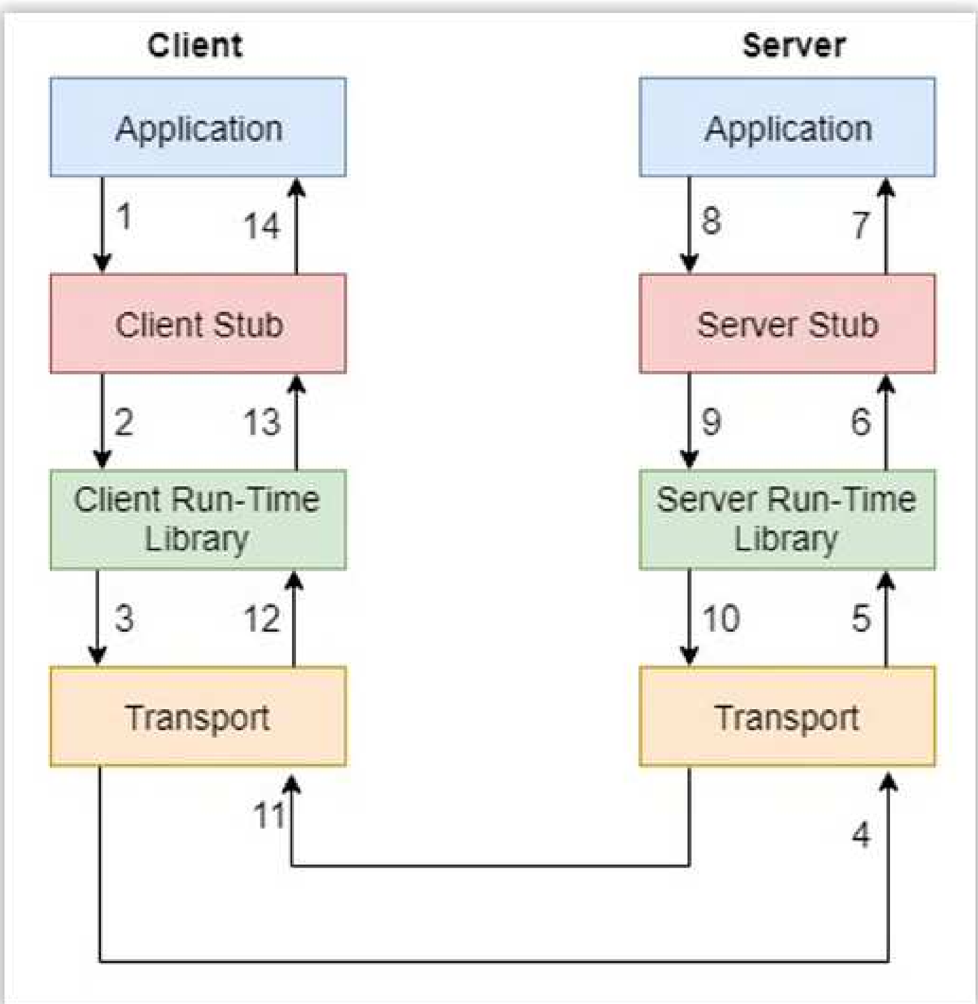


gRPC and Stub



gRPC Diagram Flow
How it works

1. A client application makes a local procedure call to the client stub containing the parameters to be passed on to the server.
2. The client stub serializes the parameters through a process called marshalling.
3. The client stub forwards the request to the local client time library.
4. The local client time library forwards the request to the server stub.
5. The server run-time library receives the request and calls the server stub procedure.
6. The server stub unmarshalls (unpacks) the passed parameters.
7. The server stub calls the actual procedure.
8. The server stub sends back a response to the client-stub in the same fashion.



GRPC main Basis

- gRPC clients and servers can run and talk to each other in a variety of environments
- gRPC is typically an architecture Client-Server.
- gRPC is using **IDL** as Definition language.
- gRPC is working with **Protocol Buffers**
- gRPC is working with **asynchronously**

GRPC Life Cycle

- **Unary RPC**
- **Server Streaming RPC**
- **Client Streaming RPC**
- **Bidirectional Streaming RPC**
- **Deadlines / Timeouts**
- **RPC Termination**
- **Cancelling an RPC**

GRPC Unary

- **Client Call a Stub method, Server is notified with Client's Metadata.**
- **Server will Callback or Wait for the client's request message.**
- **Client's Message received, gRPC will prepare response and sent to Client.**
- **When status OK received by client, the gRPC task is completed.**

GRPC main Basis

- gRPC clients and servers can run and talk to each other in a variety of environments
- gRPC is typically an architecture Client-Server.
- gRPC is using **IDL** as Definition language.
- gRPC is working with **Protocol Buffers**
- gRPC is working with **asynchronously (see gRPC Asyncio)**

GRPC, Python and Django

- **Django gRPC Framework**
- **Django-grpc 1.0.13**
- **django-grpc-secure 0.1.2**
- **django-grpc-framework**
- **django-api-gateway 0.0.7**

Tutorial

- **http://flagzeta.org/blog/using-grpc-with-django/**
-

API URL Structure - remember

- Normalize URL
  - Common URL Part
  - GET or UPDATE
  - API Service Identify

- Dynamic URL
  - Constructed with request
  - Django URLS.PY Patterns

Communication Databases

- A Microservices based on a Django Architecture needs at Minimum to be logged or obtain a uniq Token valid during the API session,

- **Django AUTH**:
  - Login with a Valid User
  - Get Credential attached to the Django account
  - Return a Token valid for a Limited Time
  - Automatic logout

- **Private Socotec Token Database**:
  - Manage our own API User Database
  - Manage our own Token & Life cycle
  - Associate various API authorization to each API user (Read, Update, component, ..)
  - Control Active and revoked Users

gRPC vs HTTP/Rest

- Finally, it’s a wrong comparison start !!
- gRPC manage Microservices itself, not direct interaction with client Browser !!
- API Under Django need to define URL Patterns
- An API architecture needs several mandatory settings :
  - Input Parameters
  - API Structure & Validation
  - Output Structure
  - Permission & Token
  - URL Structure
  - Call Back Return to End User
- gRPC does not manage it, Just MicroServices communication, thanks to ProtoBuf
- Not direct support of HTTP2 by our common Browser

gRPC and Django-Rest-framework

- **The solution could be partially found with the following Addon :**
  - **GRPC Django Framework**
  - **Python Django gRPC microservice framework**
  - **django-grpc**
    - **Easy way to launch gRPC server with access to Django ORM and other handy stuff. gRPC calls are much faster that traditional HTTP requests because communicate over persistent connection and are compressed. Underlying gRPC library is written in C which makes it work faster than any RESTful framework where a lot of time is spent on serialization/deserialization.**
  - **Grpc-web**

Django and gRPC Installation

- Examine various gRPC wrappers available on Django
  - Requisite : Python 3.4 + and Django 2.2 till 3.x
  - https://realpython.com/python-microservices-grpc/ (Tutorial)
  - https://djangogrpcframework.readthedocs.io/en/latest/

  - **Django-grpc** 1.0.13
  - django-json-rpc 0.7.2
  - Django-grpc-framework 0.3
  - Grpc-interceptor

Django gRPC Framework

- **<u>Prerequisites</u> :**
  - **Python 3.6**
  - **Django 2.2**
  - **Django REST Framework 3.10.xx**
  - **gRPC**
  - **gRPCtools**
  - **Proto3 (*Google Cloud API V3*)**

Protocol Buffers - proto3

- Protocol buffers are the flexible, efficient, automated solution to solve exactly this problem.

- With protocol buffers, you write a .proto description of the data structure you wish to store. From that, the protocol buffer compiler creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient binary format. The generated class provides getters and setters for the fields that make up a protocol buffer and takes care of the details of reading and writing the protocol buffer as a unit.

- Importantly, the protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format

Protocol Buffers or proto3

1. **Efficient**: Protocol buffers are verbose and descriptive. But they are smaller, faster, more efficient, and provide high performance

2. **Machine Readable**: Protocol buffers are binary or machine readable and can be used to exchange messages between services and not over browsers

3. **Generators**: With a compiler, Protocol buffers can be easily compiled to source code along with runtime libraries for your choice of programming language. This makes serialization or deserialization easier, with no need for hand parsing

4. **Supports Types**: Unlike JSON, we can specify field types and add validations for the same in the .proto file.

Protocol Buffers - In a Few Steps

1. Define your **protocol format**. Proto file
2. **Compile** your Protocol Buffers definition – protoc command.
3. Define your Protocol **Buffer API** (Based on Django Data Model)
4. **Write** your Message (Python MicroServices Modules & Serializers)
5. **Read** A Message (Python MicroServices Modules)

Protocol Buffers & Python

- Define message formats in a `.proto` file.
- Use the protocol buffer compiler.
- Use the Python protocol buffer API to write and read messages.

Django gRPC Framework Installation

- sudo apt-get install python-dev
- Pip install djangogrpcframework
- Pip install djangorestframework 3.12.2
- Pip install grpcio 1.36.1
- Pip install grpcio-tools 1.36.1
- https://djangogrpcframework.readthedocs.io/en/latest/tutorial/building_services.html#environment-setup
- https://gustavoh.medium.com/building-microservices-in-go-and-python-using-grpc-and-tls-ssl-authentication-cfcee7c2b052

- **Name and application convention for this Tutorial**
  - Django project **Microservices**
  - Application **CareCheck**

Django gRPC Framework Quick Install

- **Startproject** microservices
- **Startapp** carcheck (*for our sample and Tutorial guide*!)
- **Migrate** “Create Django database”
- **Installed** Apps Add the django_grpc_framework to settings.py
- **Define protos** Define gRPC services and messages
- **gRPC code** generate gRPC ode, from microservices Project Folder
- **Serializer** Create a Serializer class
- **Services** Write and connect a GRPC Microservice
- **Handlers** Create a GRPC handler
- **URLS** wire up the handlers (**Django Main Hook URLs**)
- **Settings** Don’t forget Django gRPC Hook settings (**ROOT_HANDLERS_HOOK**)
- **Run Server** Run Microservice, Let’s start the GRPC server **“50051” port**

Learn Protocol Buffer language First !

- https://developers.google.com/protocol-buffers/docs/proto3

Language Guide (proto3)

- Defining A Message Type
- Scalar Value Types
- Default Values
- Enumerations
- Using Other Message Types
- Nested Types
- Updating A Message Type
- Unknown Fields
- Any
- Oneof
- Maps
- Packages
- Defining Services
- JSON Mapping
- Options
- Generating Your Classes

Define Google Protocol Buffer .proto

- python manage.py **generateproto** --model django.contrib.auth.models.User --fields ,username,email,groups --file **user**.proto

- python manage.py **generateproto** --model carcheck.models.Car --fields cost_center,brand,model,version --file **carcheck**.proto

- python manage.py **generateproto** --model carcheck.models.CostCenter --fields code,manager,society_name --file **costcenter**.proto

Execute Google Compile .proto

- python manage.py –m **grpc_toolls.protoc** –proto_path=./ --python_out=./ --grpc_python_out=./ ./**user**.proto

- python manage.py –m **grpc_toolls.protoc** –proto_path=./ --python_out=./ --grpc_python_out=./ ./**costcenter**.proto

- python manage.py –m **grpc_toolls.protoc** –proto_path=./ --python_out=./ --grpc_python_out=./ ./**carcheck**.proto

Generated Socotec .proto

```
syntax = "proto3";

package carcheck;

import "google/protobuf/empty.proto";

service CarController {
    rpc List(CarListRequest) returns (stream Car) {}
    rpc Create(Car) returns (Car) {}
    rpc Retrieve(CarRetrieveRequest) returns (Car) {}
    rpc Update(Car) returns (Car) {}
    rpc Destroy(Car) returns (google.protobuf.Empty) {}
}

message Car {
    string cost_center = 1;
    string brand = 2;
    string model = 3;
    string version = 4;
}

message CarListRequest {
}

message CarRetrieveRequest {
    string uuid = 1;
}
```

```
syntax = "proto3";

package costcenter;

import "google/protobuf/empty.proto";

service CostCenterController {
    rpc List(CostCenterListRequest) returns (stream CostCenter)
{}
    rpc Create(CostCenter) returns (CostCenter) {}
    rpc Retrieve(CostCenterRetrieveRequest) returns (CostCenter)
{}
    rpc Update(CostCenter) returns (CostCenter) {}
    rpc Destroy(CostCenter) returns (google.protobuf.Empty) {}
}

message CostCenter {
    string code = 1;
    string manager = 2;
    string society_name = 3;
}

message CostCenterListRequest {
}

message CostCenterRetrieveRequest {
    string uuid = 1;
}
```

```
syntax = "proto3";

package user;

import "google/protobuf/empty.proto";

service UserController {
    rpc List(UserListRequest) returns (stream User) {}
    rpc Create(User) returns (User) {}
    rpc Retrieve(UserRetrieveRequest) returns (User) {}
    rpc Update(User) returns (User) {}
    rpc Destroy(User) returns (google.protobuf.Empty) {}
}

message User {
    int32 id = 1;
    string username = 2;
    string email = 3;
    repeated int32 groups = 4;
}

message UserListRequest {
}

message UserRetrieveRequest {
    int32 id = 1;
}
```

Compiled Socotec gRPC Python module

Socotec Services	Compiled gRPC Python Module
User	**User**_pb2.py
	User_pb2_grpc
carcheck	**Carcheck**_pb2.py
	Cacheck_pb2_grpc.py
costcenter	**Costcenter**_pb2.py **Costcenter**_pb2_grpc.py

Tests with Django gRPC

Let's start the gRPC server

```
$ python manage.py grpcrunserver --dev
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
March 17, 2021 - 16:16:27
Django version 2.2, using settings 'microservices.settings'
Starting development gRPC server at [::]:50051
Quit the server with CTRL-BREAK.
```

Tests with Django gRPC

Let's call with Django Management Command

```
# ----------------------------------
# --- gRPC Client Part          ----
# ----------------------------------
import grpc
import carecheck_pb2
import carecheck_pb2_grpc

class Command(BaseCommand):

    help = "Test gRPC Client"
    def add_arguments(self, parser):
        help = 'Test gRPC Client'
        parser.add_argument('--mode', action= 'store', dest= 'mode', default = '', help = 'select a Mode gRPC')


    def handle(self, *args, **options):


        with grpc.insecure_channel('localhost:50051') as channel:
            stub = carecheck_pb2_grpc.UserControllerStub(channel)
            stubList = stub.List(carecheck_pb2.UserListRequest())
            for user in stubList:
                print(user, end='')
```

Generate gRPC generic python code

- python -m grpc_tools.protoc --proto_path=./ --python_out=./ --grpc_python_out=./ ./**carecheck**.proto

- **It will generate 2 files (***Theses 2 files should remain intact* **!)**:
  - carecheck_pb2.py
  - carecheck_pb2_grpc.py

Validate each Microservice in "standalone"

- It's possible to evaluate, debug and test each Microservices defined in "services.py" outside the gRPC Server .

- Sample Test code :
  - from django_grpc_framework.test import Channel
  - channel = **Channel**()
  - stub = carecheck_pb2_grpc.UserControllerStub(channel)
  - response = stub.Retrieve(carecheck_pb2.UserRetrieveRequest(id=self.id))
  - response.title

Django gRPC   Python integration

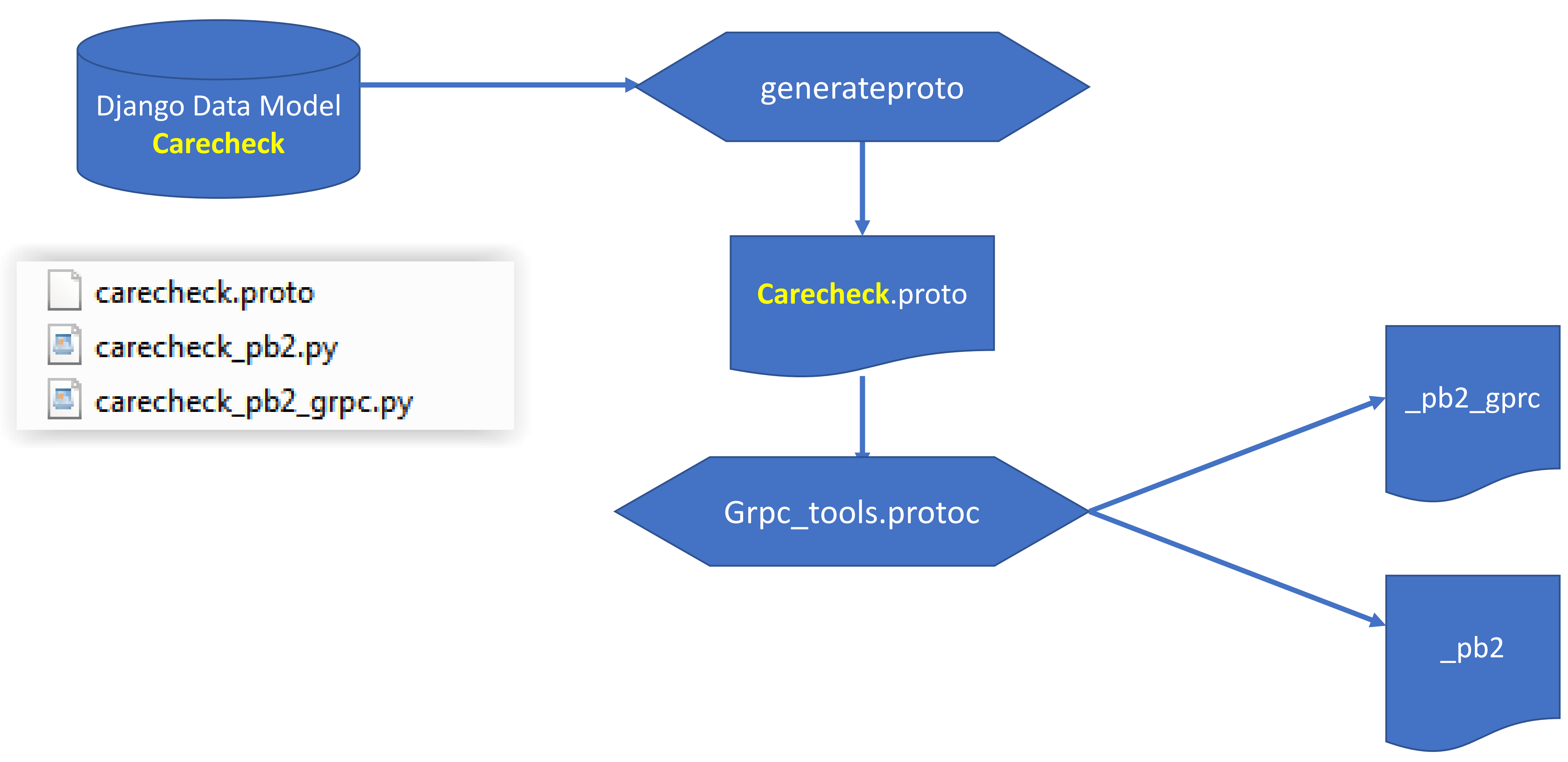
Generated files for service : carcheck

Module	Description	By ?
Carecheck.proto	Proto Buffer Definition	gRPC Framework
Carecheck_pb2.py Carecheck_pb2_grpc.py	gRPC protocol Buffer code gRPC Client & Server Classes	gRPC Framework gRPC Framework

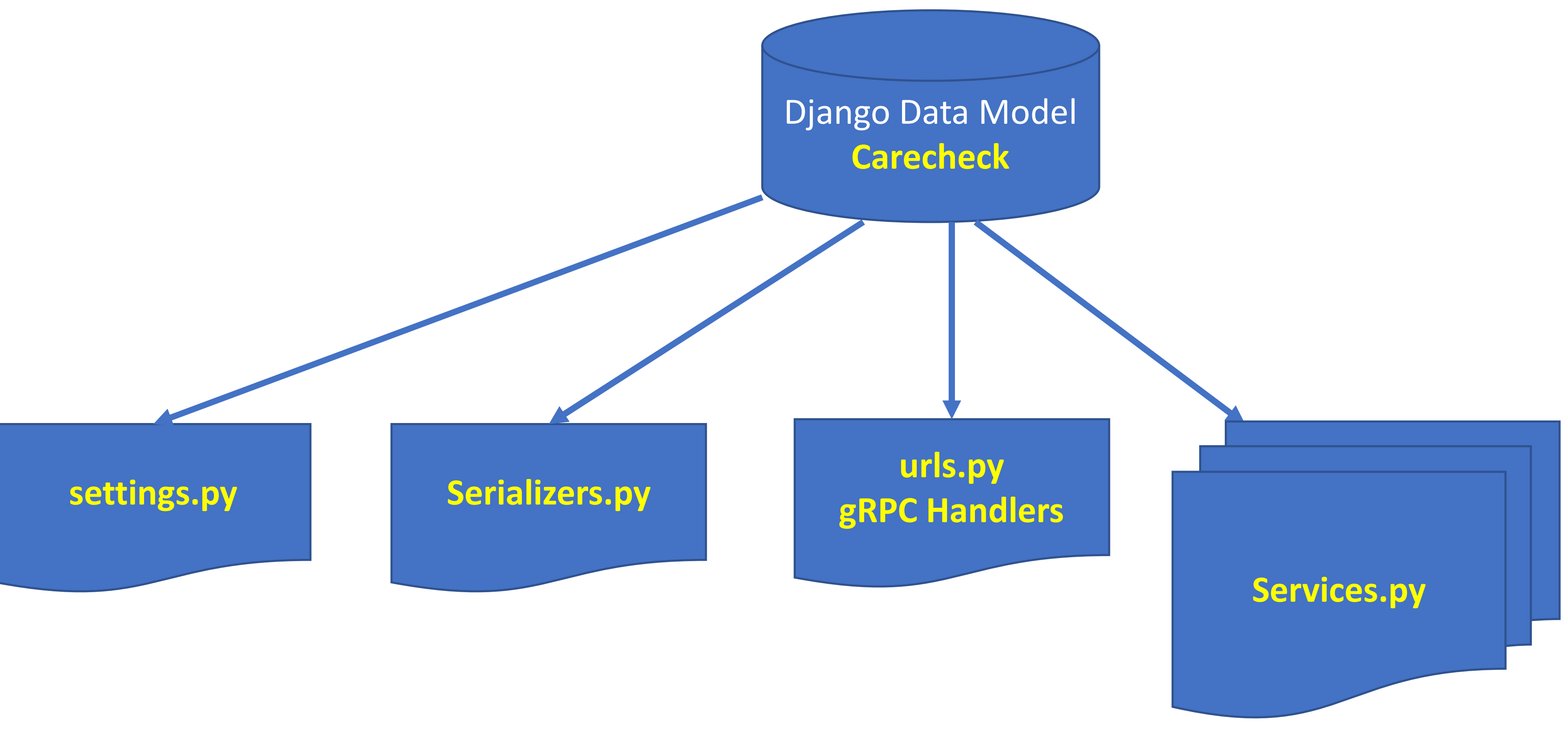
Application files

Module	Description	By ?
Settings.py	Django Settings for gRPC	Developer
Serializers.py Services.py Models.py Urls.py Django Management/command	Django Rest Serializers Microservices definition Django Data Models Django URL and gRPC Register Handler Python gRPC Client Module	Developer Developer Developer Developer Developer

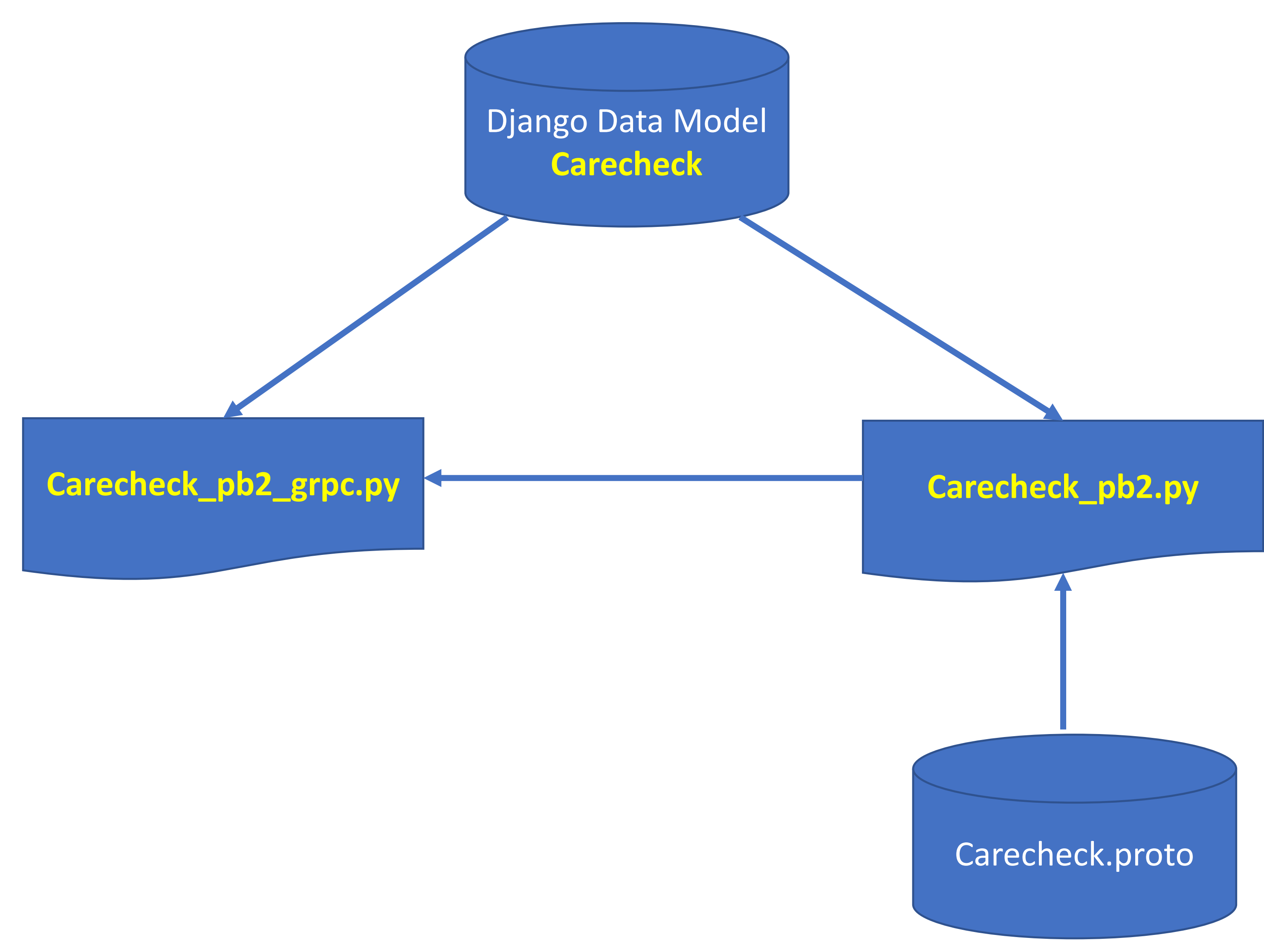
Django gRPC Python Modules compilation



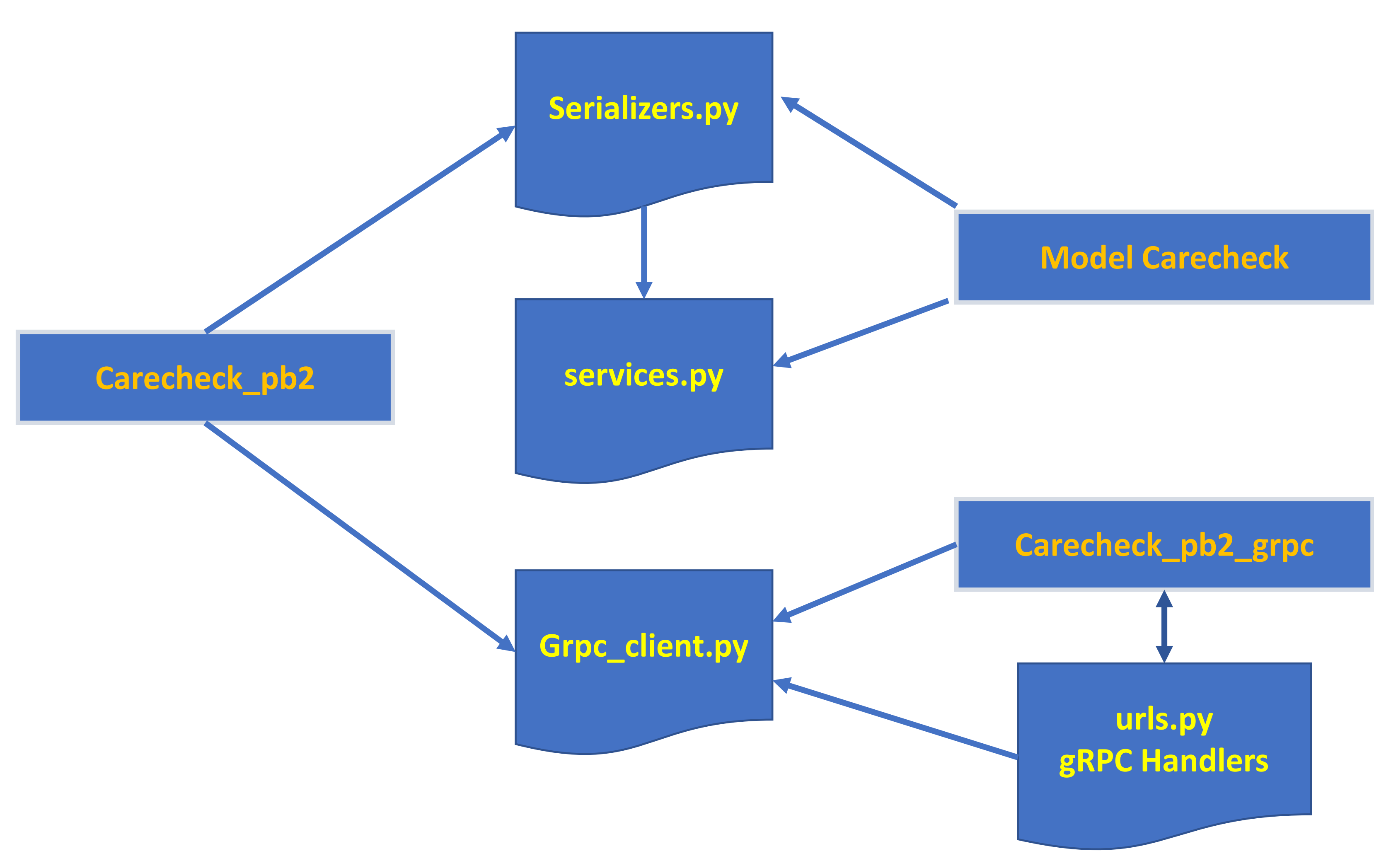
Django gRPC   Python Modules handwriting



Django gRPC pb2 Python Modules (No Edit!)



Django gRPC Python Modules Interactions



Django gRPC - Return Code

Code	Number	Description
OK	0	Not an error; returned on success.
CANCELLED	1	The operation was cancelled, typically by the caller.
UNKNOWN	2	Unknown error. For example, this error may be returned when a `Status` value received from another address space belongs to an error space that is not known in this address space. Also errors raised by APIs that do not return enough error information may be converted to this error.
INVALID_ARGUMENT	3	The client specified an invalid argument. Note that this differs from `FAILED_PRECONDITION`. `INVALID_ARGUMENT` indicates arguments that are problematic regardless of the state of the system (e.g., a malformed file name).
DEADLINE_EXCEEDED	4	The deadline expired before the operation could complete. For operations that change the state of the system, this error may be returned even if the operation has completed successfully. For example, a successful response from a server could have been delayed long
NOT_FOUND	5	Some requested entity (e.g., file or directory) was not found. Note to server developers: if a request is denied for an entire class of users, such as gradual feature rollout or undocumented allowlist, `NOT_FOUND` may be used. If a request is denied for some users within a class of users, such as user-based access control, `PERMISSION_DENIED` must be used.
ALREADY_EXISTS	6	The entity that a client attempted to create (e.g., file or directory) already exists.
PERMISSION_DENIED	7	The caller does not have permission to execute the specified operation. `PERMISSION_DENIED` must not be used for rejections caused by exhausting some resource (use `RESOURCE_EXHAUSTED` instead for those errors). `PERMISSION_DENIED` must not be used if the caller can not be identified (use `UNAUTHENTICATED` instead for those errors). This error code does not imply the request is valid or the requested entity exists or satisfies other pre-conditions.
RESOURCE_EXHAUSTED	8	Some resource has been exhausted, perhaps a per-user quota, or perhaps the entire file system is out of space.
FAILED_PRECONDITION	9	The operation was rejected because the system is not in a state required for the operation's execution. For example, the directory to be deleted is non-empty, an rmdir operation is applied to a non-directory, etc. Service implementors can use the following guidelines to decide between `FAILED_PRECONDITION`, `ABORTED`, and `UNAVAILABLE`: (a) Use `UNAVAILABLE` if the client can retry just the failing call. (b) Use `ABORTED` if the client should retry at a higher level (e.g., when a client-specified test-and-set fails, indicating the client should restart a read-modify-write sequence). (c) Use `FAILED_PRECONDITION` if the client should not retry until the system state has been explicitly fixed. E.g., if an "rmdir" fails because the directory is non-empty, `FAILED_PRECONDITION` should be returned since the client should not retry unless the files are deleted from the directory.

Django gRPC - Return Code

Code	Number	Description
ABORTED	10	The operation was aborted, typically due to a concurrency issue such as a sequencer check failure or transaction abort. See the guidelines above for deciding between `FAILED_PRECONDITION`, `ABORTED`, and `UNAVAILABLE`.
OUT_OF_RANGE	11	The operation was attempted past the valid range. E.g., seeking or reading past end-of-file. Unlike `INVALID_ARGUMENT`, this error indicates a problem that may be fixed if the system state changes. For example, a 32-bit file system will generate `INVALID_ARGUMENT` if asked to read at an offset that is not in the range [0,2^32-1], but it will generate `OUT_OF_RANGE` if asked to read from an offset past the current file size. There is a fair bit of overlap between `FAILED_PRECONDITION` and `OUT_OF_RANGE`. We recommend using `OUT_OF_RANGE` (the more specific error) when it applies so that callers who are iterating through a space can easily look for an `OUT_OF_RANGE` error to detect when they are done.
UNIMPLEMENTED	12	The operation is not implemented or is not supported/enabled in this service.
INTERNAL	13	Internal errors. This means that some invariants expected by the underlying system have been broken. This error code is reserved for serious errors.
UNAVAILABLE	14	The service is currently unavailable. This is most likely a transient condition, which can be corrected by retrying with a backoff. Note that it is not always safe to retry non-idempotent operations.
DATA_LOSS	15	Unrecoverable data loss or corruption.
UNAUTHENTICATED	16	The request does not have valid authentication credentials for the operation.

Django gRPC - Error Cases

Case	Code	Generated at Client or Server
Client Application cancelled the request	CANCELLED	Both
Deadline expires before server returns status	DEADLINE_EXCEEDED	Both
Method not found at server	UNIMPLEMENTED	Server
Server shutting down	UNAVAILABLE	Server
Server side application throws an exception (or does something other than returning a Status code to terminate an RPC)	UNKNOWN	Server
No response received before Deadline expires. This may occur either when the client is unable to send the request to the server or when the server fails to respond in time.	DEADLINE_EXCEEDED	Both
Some data transmitted (e.g., request metadata written to TCP connection) before connection breaks	UNAVAILABLE	Client
Could not decompress, but compression algorithm supported (Client -> Server)	INTERNAL	Server
Could not decompress, but compression algorithm supported (Server -> Client)	INTERNAL	Client
Compression mechanism used by client not supported at server	UNIMPLEMENTED	Server
Server temporarily out of resources (e.g., Flow-control resource limits reached)	RESOURCE_EXHAUSTED	Server
Client does not have enough memory to hold the server response	RESOURCE_EXHAUSTED	Client
Flow-control protocol violation	INTERNAL	Both
Error parsing returned status	UNKNOWN	Client
Incorrect Auth metadata (Credentials failed to get metadata, Incompatible credentials set on channel and call, Invalid host set in `:authority` metadata, etc.)	UNAUTHENTICATED	Both
Request cardinality violation (method requires exactly one request but client sent some other number of requests)	UNIMPLEMENTED	Server
Response cardinality violation (method requires exactly one response but server sent some other number of responses)	UNIMPLEMENTED	Client
Error parsing response proto	INTERNAL	Client
Error parsing request proto	INTERNAL	Server
Sent or received message was larger than configured limit	RESOURCE_EXHAUSTED	Both
Keepalive watchdog times out	UNAVAILABLE	Both

Remarks & Issues

- It seems **generateproto** command does not work properly with Auth User Abstract, to be confirmed !!
  - AUTH_USER_MODEL
- Don't Forget Django **gRPC** settings ! (ROOT_HANDLERS_HOOK)
- By Default, communication gRPC port will be configured at '**50051**"
- A Global port setting should be implemented !
- Pb2 compiled Python module do not be edited !!
- "**Generateproto**" does not support "update" on existing proto !
- "**Generateproto**" take first field as primary field !
- gRPC Python currently don't support server-side global error handle, It's necessary to define a common **Error Handler** !
- It's possible to test in "**standalone**" mode each Microservice

gRPC

Logging

Django Logging Basis

- Loggers
  - **DEBUG, INFO, WARNING, ERROR, CRITICAL**
- Handlers
  - Logging Engine
- Filters
  - Filtering and additional control
- Formatters
  - Rendering Log record (Text, …)

Django Logging Call

- Logger.debug()
- Logger.info()
- Logger.warning()
- Logger.**error**()
- Logger.critical()
- Logger.log()
- Logger.**exception**() ERROR !

Available LOGGING TOOLS

- PAID :
  - Sentry
  - Raven

- Standard Python Open Source:
  - Django base Logging
  - Django-structlog

gRPC LOGGING Settings

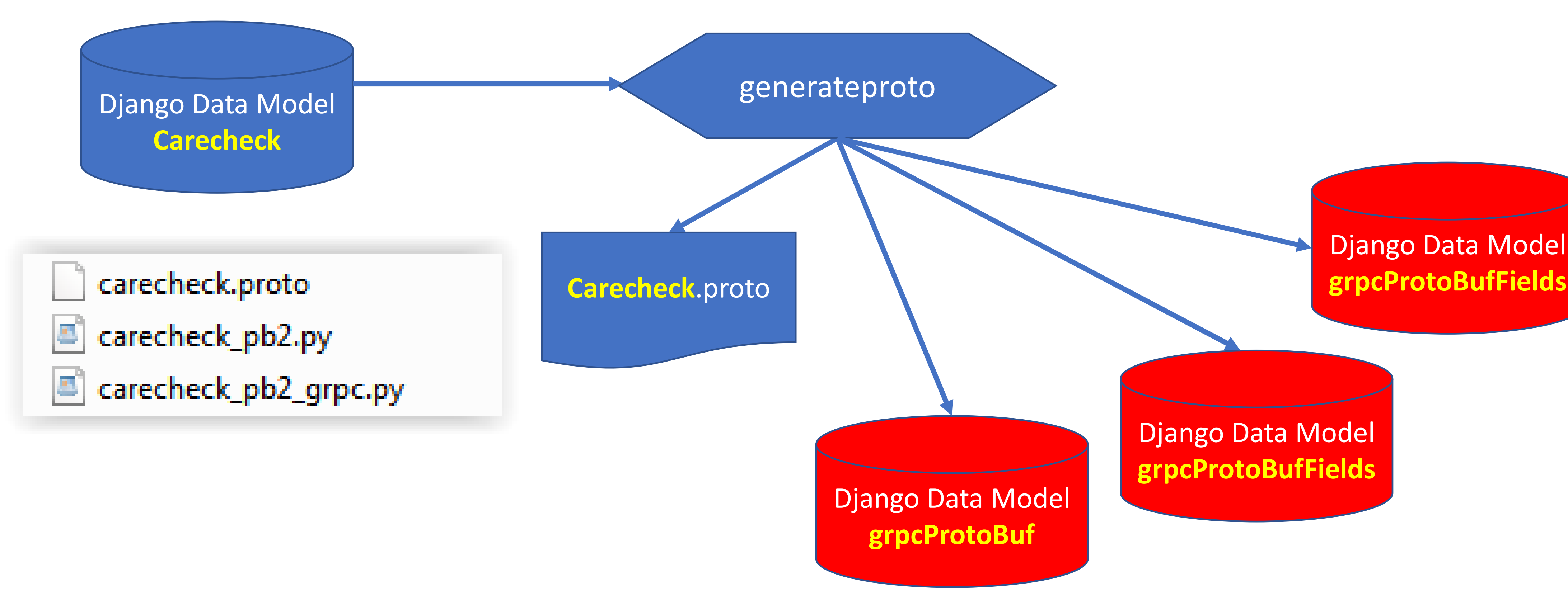
gRPC

Asyncio

gRPC   Client Class

New ProtoBuf generator

Django gRPC New ProtoBuf Generator !



Django gRPC : New gRPC Client Class !

```
# ---------------------------------------------
# ---- gRPC SERVER IS  RUNNING               ---
# ---- CALL MICROSERVICES                    ---
# ---------------------------------------------
grpcHandle = grpcClient(self.service, method=self.operator, channel=self.gRPCPort, value=self.value, debug=True)
if grpcHandle.statusGRPC():
    status = grpcHandle.Microservice()

# -----------------------------------------
# --- PRINT RESULT OF MICROSERVICE HERE ---
# -----------------------------------------
if status:
    print('         ')
    print('--------  RESULT   HERE  ----------')
    if grpcHandle.formatResult() == 'array':
        for user in grpcHandle.resultGRPC():
            print('                      ')
            print('    %s'  % user)
    else:
        print(grpcHandle.resultGRPC())
    print('-----------------------------------')
else:
    print(' --------------------------------------')
    print(' **** N O      R E S U L T S     *****')
    print(' --------------------------------------')
```

Django gRPC : New gRPC Client Admin Tool

Django administration

WELCOME, GON2000FR. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

CARCHECK		
Cars	+ Add	Change
Cost centers	+ Add	Change

DJANGO_GRPC_FRAMEWORK		
GRPC DATABASES	+ Add	Change
GRPC ERRORS CODES	+ Add	Change
GRPC ERRORS HANDLER	+ Add	Change
GRPC LOGGING HANDLER	+ Add	Change
GRPC PROTOBUF	+ Add	Change
GRPC PROTOBUF FIELDS	+ Add	Change
GRPC SERVICES METHODS	+ Add	Change
GRPC SERVICES REFERENCES	+ Add	Change

FCM DJANGO		
FCM devices	+ Add	Change

NOTIFICATION

Recent actions

My actions

- + 908 (INVALID STUD REQUEST) GRPC ERROR CODE
- + 907 (NO GRPC SERVER FOUND) GRPC ERROR CODE
- + 906 (MISSING PB2 GRPC FILE) GRPC ERROR CODE
- + 905 (MISS PROTOBUF FIELDS) GRPC ERROR CODE
- + 904 (NO METHOD FOUND) GRPC ERROR CODE
- + 903 (NO DATABASE DEFINED FOR THIS SERVICE) GRPC ERROR CODE
- + 902 (INVALID SERVICE) GRPC ERROR CODE
- retrieve (carcheck) GRPC SERVICE METHOD
- + retrieve (carcheck) GRPC SERVICE METHOD
- list (carcheck) GRPC SERVICE METHOD

Django gRPC Error Handler : New SQL Tables !

DJANGO_GRPC_FRAMEWORK		
GRPC DATABASES	+ Add	Change
GRPC ERRORS CODES	+ Add	Change
GRPC ERRORS HANDLER	+ Add	Change
GRPC LOGGING HANDLER	+ Add	Change
GRPC PROTOBUF	+ Add	Change
GRPC PROTOBUF FIELDS	+ Add	Change
GRPC SERVICES METHODS	+ Add	Change
GRPC SERVICES REFERENCES	+ Add	Change

Django gRPC Logging Handler : New SQL Table !

Select GRPC LOGGING HANDLER to change

ADD GRPC LOGGING HANDLER +

Action: --------- Go 0 of 8 selected

	SOCOTEC MICROSERVICE	DATABASE MICROSERVICE	METHOD	CREATED
☐	carcheck	Car	Retrieve	March 29, 2021, 1:07 a.m.
☐	carcheck	Car	Retrieve	March 29, 2021, 12:09 a.m.
☐	carcheck	Car	Retrieve	March 29, 2021, 12:09 a.m.
☐	carcheck	Car	Retrieve	March 29, 2021, 12:07 a.m.
☐	carcheck	Car	Retrieve	March 29, 2021, 12:06 a.m.
☐	carcheck	Car	Retrieve	March 29, 2021, 12:03 a.m.
☐	carcheck	Car	List	March 28, 2021, 9:30 p.m.
☐	carcheck	Car	List	March 28, 2021, 8:04 p.m.

8 GRPC LOGGING HANDLER

FILTER

By is active

- All
- Yes
- No

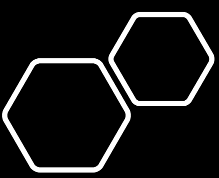
By is delete

- All
- Yes
- No

By Socotec Microservice

- All
- carcheck
- notification
- -

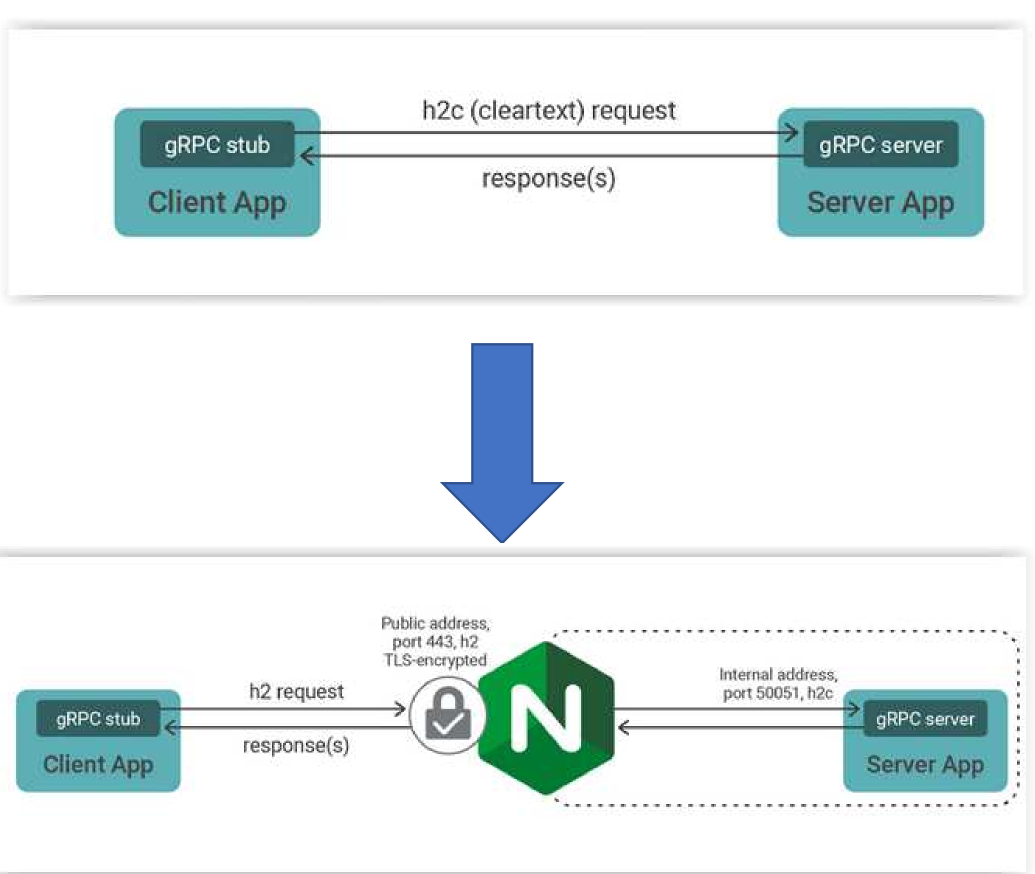
gRPC and NGINX



NGINX and Http/2

- gRPC http/2 support since NGINX 1.13.10

NGINX Proxying gRPC Traffic



NGINX Sample Proxy

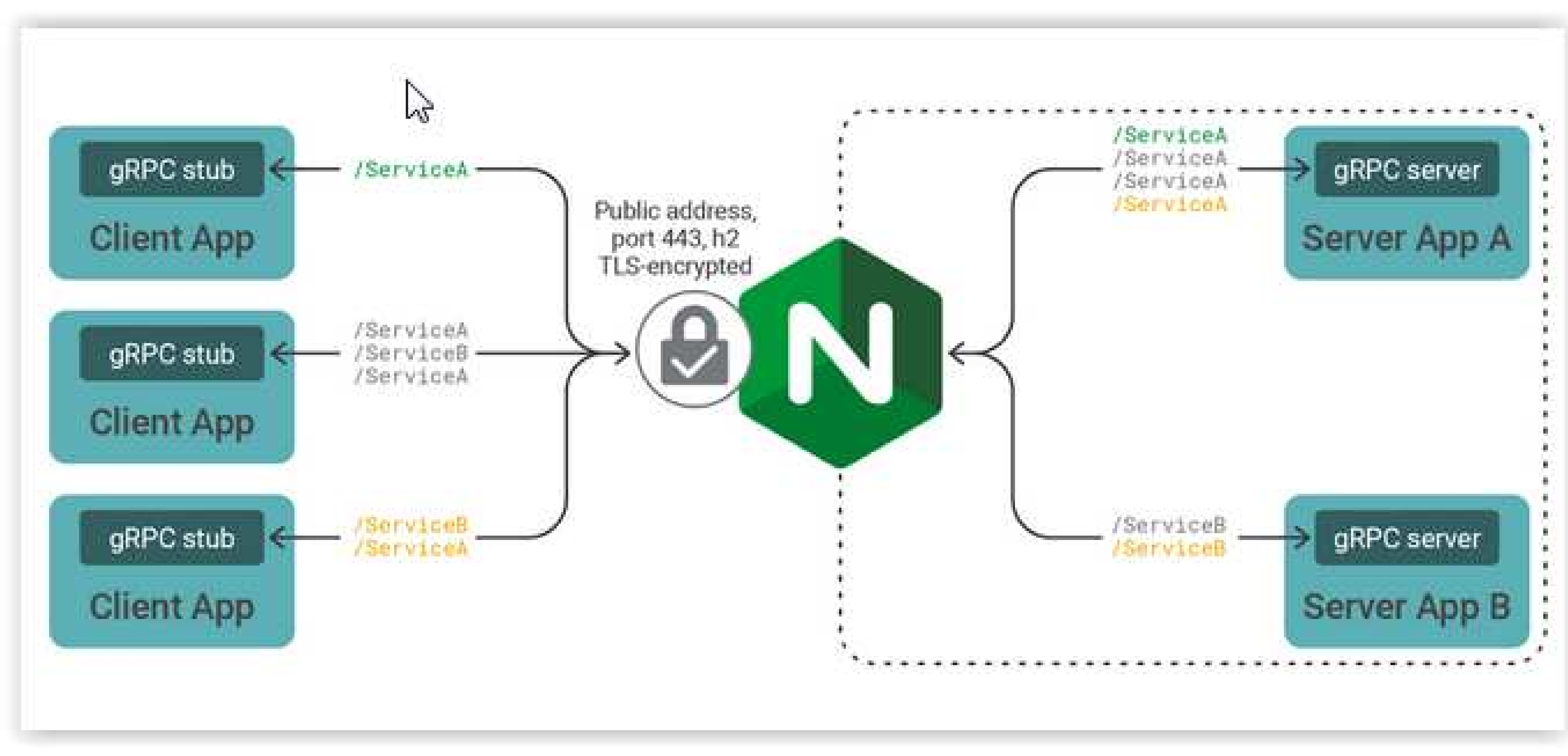
```
http {
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent"';

    server {
        listen 80 http2;

        access_log logs/access.log main;

        location / {
            # Replace localhost:50051 with the address and port of your gRPC
server
            # The 'grpc://' prefix is optional; unencrypted gRPC is the default
            grpc_pass grpc://localhost:50051;
        }
    }
}
```

NGINX gRPC Routing



gRPC   and Asyncio API

Asyncio API - Concept

gRPC AsyncIO API

Overview

gRPC AsyncIO API is the **new version** of gRPC Python whose architecture is tailored to AsyncIO. Underlying, it utilizes the same C-extension, gRPC C-Core, as existing stack, and it replaces all gRPC IO operations with methods provided by the AsyncIO library.

This API is stable. Feel free to open issues on our GitHub repo grpc/grpc for bugs or suggestions.

The design doc can be found here as gRFC.

Caveats

gRPC Async API objects may only be used on the thread on which they were created. AsyncIO doesn't provide thread safety for most of its APIs.

Blocking Code in AsyncIO

Making blocking function calls in coroutines or in the thread running event loop will block the event loop, potentially starving all RPCs in the process. Refer to the Python language documentation on AsyncIO for more details (running-blocking-code).

Asyncio API - Server

Create Server

`grpc.aio.server`(*migration_thread_pool=None, handlers=None, interceptors=None, options=None, maximum_concurrent_rpcs=None, compression=None*) [source]

Creates a Server with which RPCs can be serviced.

Parameters:

- **migration_thread_pool** (*Optional[concurrent.futures._base.Executor]*) – A futures.ThreadPoolExecutor to be used by the Server to execute non-AsyncIO RPC handlers for migration purpose.
- **handlers** (*Optional[Sequence[grpc.GenericRpcHandler]]*) – An optional list of GenericRpcHandlers used for executing RPCs. More handlers may be added by calling add_generic_rpc_handlers any time before the server is started.
- **interceptors** (*Optional[Sequence[Any]]*) – An optional list of ServerInterceptor objects that observe and optionally manipulate the incoming RPCs before handing them over to handlers. The interceptors are given control in the order they are specified. This is an EXPERIMENTAL API.
- **options** (*Optional[Sequence[Tuple[str, Any]]]*) – An optional list of key-value pairs (channel_arguments in gRPC runtime) to configure the channel.
- **maximum_concurrent_rpcs** (*Optional[int]*) – The maximum number of concurrent RPCs this server will service before returning RESOURCE_EXHAUSTED status, or None to indicate no limit.
- **compression** (*Optional[grpc.Compression]*) – An element of grpc.compression, e.g. grpc.compression.Gzip. This compression algorithm will be used for the lifetime of the server unless overridden by set_compression. This is an EXPERIMENTAL option.

Returns: A Server object.

Modules Contents

- Channel Objects
- Server
- Exceptions
  - Errors management
- Interceptors
  - *Server & Client*
- Client

gRPC and JavaScript

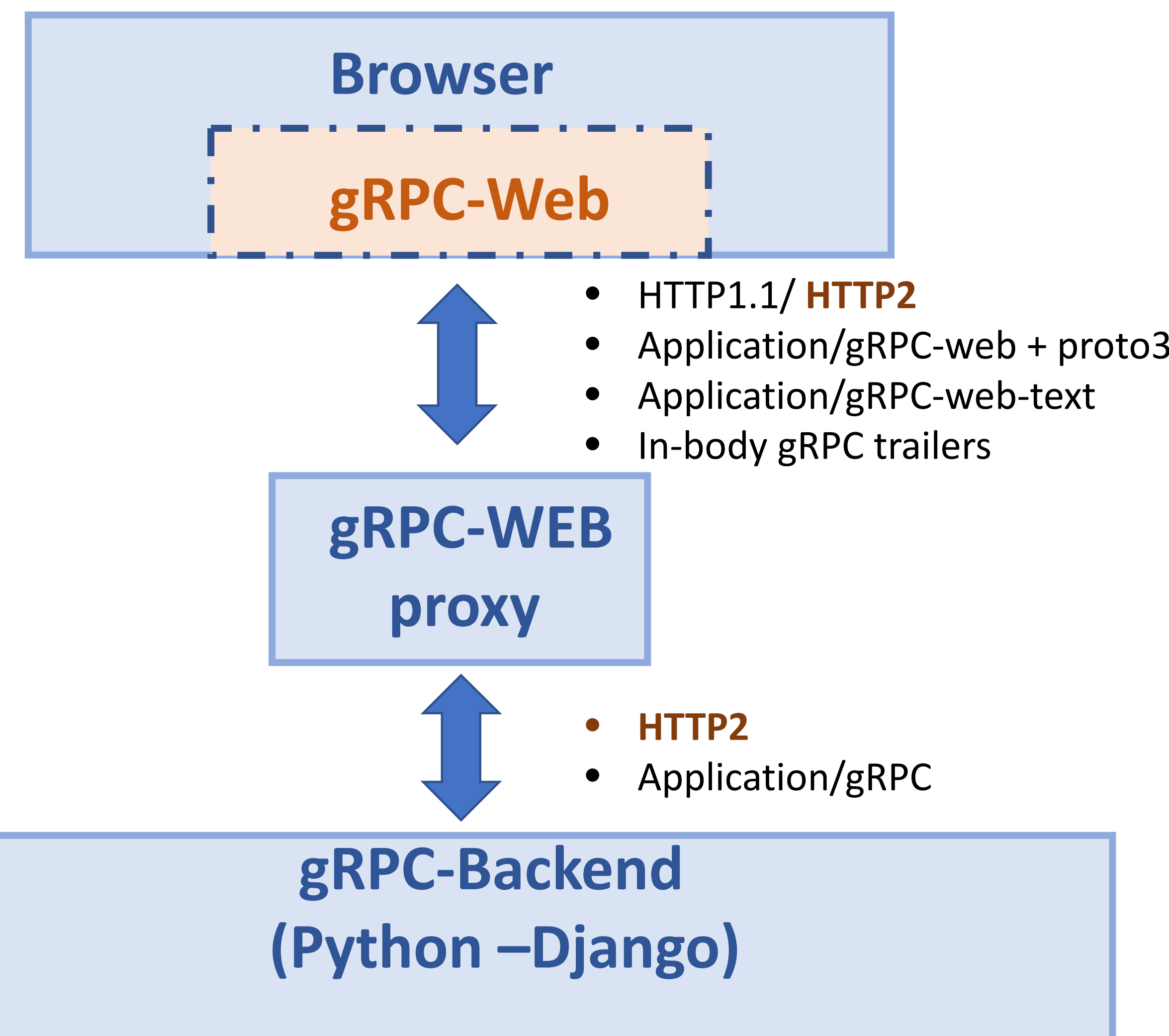
gRPC and Web client

- Installation of gRPC-web - git clone https://github.com/grpc/grpc-web

- Docker-compose pull

- Docker-compose up –d commonjs-client

gRPC and Web client



Django gRPC - External Links – Guide PDF

- https://readthedocs.org/projects/grpc-django/downloads/pdf/latest/
- https://djangogrpcframework.readthedocs.io/_/downloads/en/stable/pdf/
- https://grpc.io/docs/what-is-grpc/core-concepts/

gRPC Internal code Flow

Call Microservice

- _channel.py
  - _channel_managed_call_management
    - **state.channel.integrated_call()** will call the Socotec Microservice itself
    - **_end_unary_response_blocking** end of Response

CQRS and API Integration

-

EventSourcing

CQRS

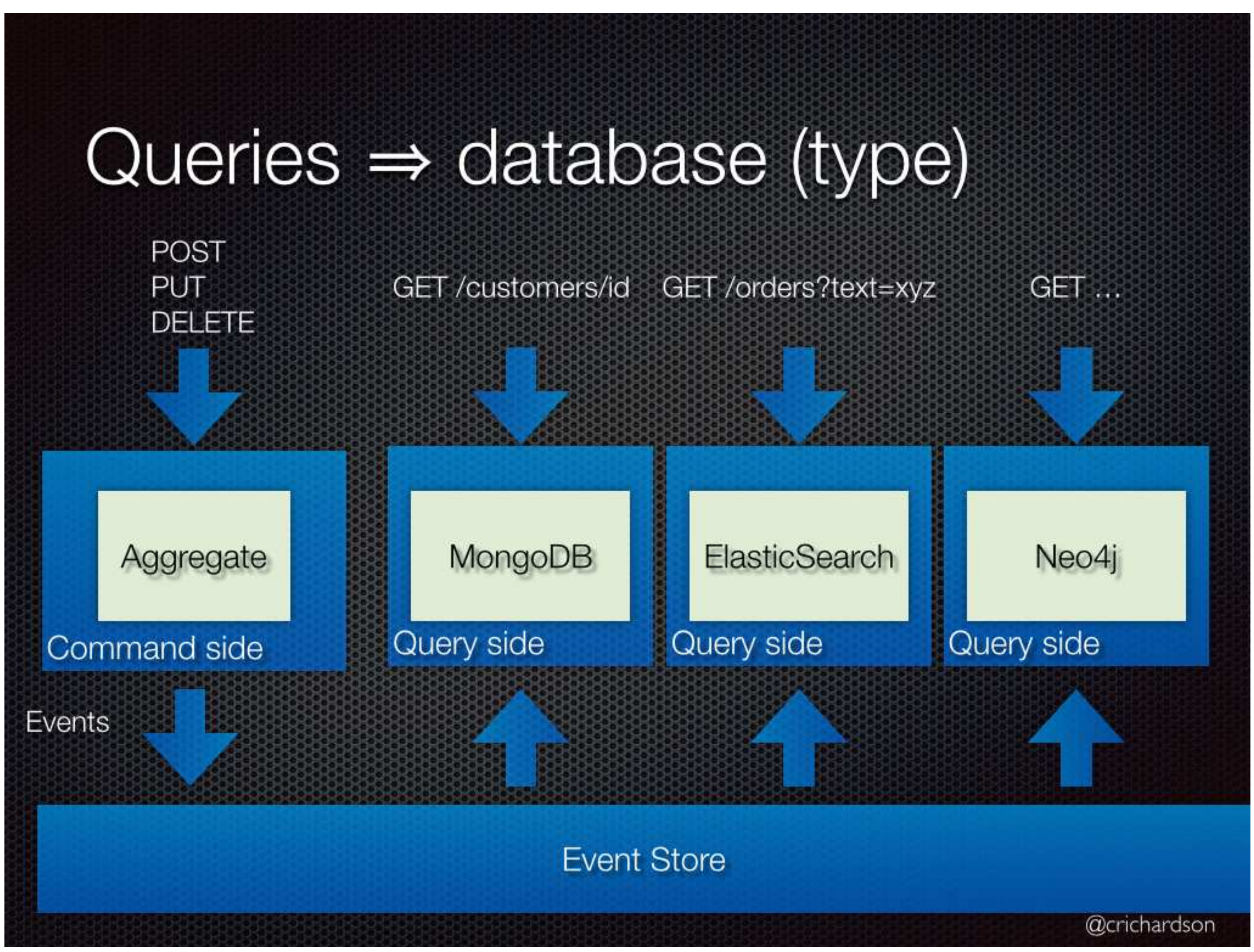
-

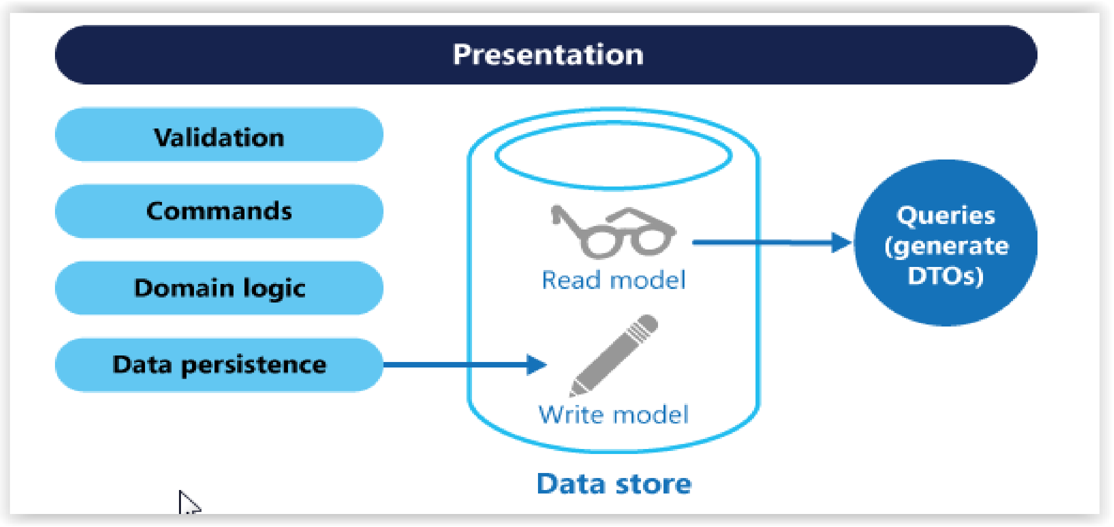
Command Query Responsibility Segregation

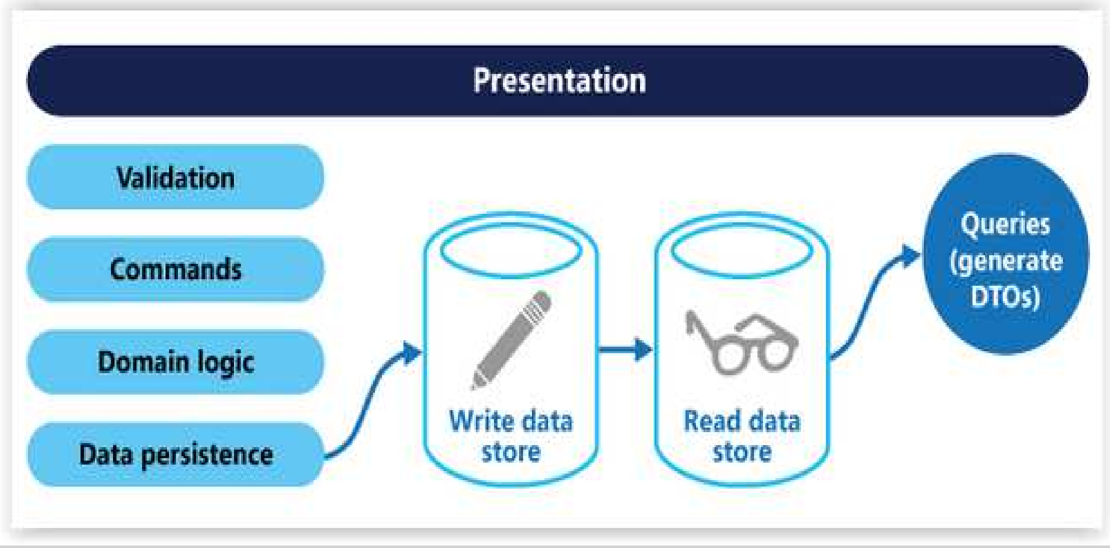
CQRS with Python and Django and EventSourcing

- Basic Resources :
  - Erlang
  - RabbitMQ (on Linux Server)
  - Celery
  - Django-celery
  - **Django-cqrs 1.7.6**
  - **Eventsourcing**
  - **Django-events-sourcing (But no documentation !)**

CQRS the Basis







CQRS - The Solution ?

CQRS the concept

- Centralize all Updates from Various databases
- Update a Global Replica Database in charge to consolidate all Data Up to date !
- Set Update or POC Flag for the Data
- API should access to this Data as main source ?
- Activate Event Sourcing to **keep Data Integrity** for a Data group shared across several databases.
- Use CQRS to create new MicroServices process in **Read only** for complex Request and **queries with various filtering criterias**.

CQRS – What we need, What we have ?

- Individual Socotec Microservices :
  - CRUD / REST API and PostgreSQL DB (Source – Master)
- Webhooks Microservices :
  - Send Outbound HTTP messages
  - Replica DB (*store Data, Historic, Triggers*)
- Aggregate Data service :
  - Read Only Public API, aggregated Data across multiple Socotec Services DB
  - NOSQL DB like MongoDB ?
- Reliable Fast Message Transport :
  - RabbitMQ or Apache Kafka

CQRS and Django

- **django-cqrs 1.7.6**
- **https://django-cqrs.readthedocs.io/en/latest/**

- in Connect we have a rather complex Domain Model. There are many microservices, that are decomposed by subdomain and which follow database-per-service pattern. These microservices have rich and consistent APIs. They are deployed in cloud k8s cluster and scale automatically under load. Many of these services aggregate data from other ones and usually API Composition is totally enough. But, some services are working too slowly with API JOINS, so another pattern needs to be applied.

- The pattern, that solves this issue is called CQRS - Command Query Responsibility Segregation. Core idea behind this pattern is that view databases (replicas) are defined for efficient querying and DB joins. Applications keep their replicas up to data by subscribing to Domain events published by the service that owns the data. Data is eventually consistent and that's okay for non-critical business transactions.

Django-CQRS -- Requirements

• **Django**	>= 1.11
• **Python**	>= 3.6
• **Pika**	1.1.0
• **Kombu**	4.6
• **Ujson**	3.0.0
• Django-model-utils	4.0
• **RabbitMQ AMQP**	

Install RabbitMQ Server & AMQP

- AMQP : **A**dvanced **M**essage **Q**ueuing **P**rotocol
- **See standard Update on APT libraries (***on RabbitMQ website***)**
- **Sudo apt-get update -y**
- Sudo apt-get install erlang
- Sudo apt-get install RabbitMQ-server

Django CQRS Settings

- Add ‘dj_cqrs’ to INSTALLED_APP

- ADD CQRS Base settings :
  - Transport (RabbitMQ) , URL and queue

- Add CQRS Mixin to your DataModel (ReplicaMixin)
  - Add cqrs_revision and cqrs_updated

- Run Django Migration

- Run consumer process :
  - Cqrs_consume

Socotec- CQRS – Django DataBase structure

Table	Action		
auth_group	Parcourir	Structure	Rechercher
auth_group_permissions	Parcourir	Structure	Rechercher
auth_permission	Parcourir	Structure	Rechercher
auth_user	Parcourir	Structure	Rechercher
auth_user_groups	Parcourir	Structure	Rechercher
auth_user_user_permissions	Parcourir	Structure	Rechercher
carcheck_car	Parcourir	Structure	Rechercher
carcheck_costcenter	Parcourir	Structure	Rechercher
django_admin_log	Parcourir	Structure	Rechercher
django_content_type	Parcourir	Structure	Rechercher
django_migrations	Parcourir	Structure	Rechercher
django_session	Parcourir	Structure	Rechercher
user_userprofile	Parcourir	Structure	Rechercher

External Links

External Links - Documentations

- https://realpython.com/python-microservices-grpc/
- https://grpc.io/docs/languages/python/quickstart/
- https://grpc.github.io/grpc/python/grpc.html

- https://pypi.org/project/django-grpc-secure/
- https://pypi.org/project/django-grpc/
- https://pypi.org/project/django-cqrs/
- https://www.django-rest-framework.org/
- https://cloud.google.com/apis/design/proto3

- https://djangogrpcframework.readthedocs.io/en/latest/
- https://github.com/tanmaybaranwal/grpc-django-book-service
- https://github.com/topics/grpc-gateway
- https://github.com/tmc/grpc-websocket-proxy

External Links – Documentations - 2

- https://www.nginx.com/blog/nginx-1-13-10-grpc/
- https://github.com/grpc/grpc-web
- https://grpc.io/docs/languages/python/basics/#example-code-and-setup
- https://awesomeopensource.com/project/fengsp/django-grpc-framework
- https://pypi.org/project/django-grpc-bus/
- https://grpc.github.io/grpc/core/md_doc_statuscodes.html
- https://github.com/grpc/grpc/blob/master/doc/python/server_reflection.md

- https://djangogrpcframework.readthedocs.io/en/latest/tutorial/building_services.html#environment-setup

API Usages

API Usages

- ….. **Car-Check-back** ….
- ….. To be documented ….
- ….. To be documented ….
- ….. To be documented ….