



Web services

Sommaire

1. Introduction & SOAP Web Services
2. Web Services RESTful
3. REST avancé, sécurité et documentation
4. gRPC et GraphQL
5. Les microservices



Introduction & SOAP Web Services

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Qu'est-ce qu'un web service ?

- Un **web service** est une application accessible via un **réseau (souvent Internet)**, permettant à différentes applications – écrites dans des langages ou exécutées sur des plateformes différentes – de **communiquer et d'échanger des données**.

Objectif principal :

- Rendre des fonctionnalités d'un système disponibles à d'autres systèmes via des interfaces standardisées.

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Qu'est-ce qu'un web service ?

-Exemple simple :

Un service météo accessible via Internet :

- Entrée : nom d'une ville → `{"city": "Paris"}`
- Sortie : température → `{"temperature": 19}`

Propriétés clés :

- Interopérabilité
- Communication via des protocoles standard (HTTP, HTTPS, TCP, etc.)
- Utilisation de formats d'échange universels (XML, JSON, Protobuf, etc.)

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ **Architecture SOA (Service-Oriented Architecture)**

- L'**architecture orientée services (SOA)** repose sur la notion de **services indépendants et réutilisables** qui interagissent via des interfaces bien définies.

Principes de SOA :

- **Faible couplage** : les services ne dépendent pas fortement les uns des autres.
- **Interopérabilité** : les services peuvent communiquer malgré des technologies différentes.
- **Réutilisabilité** : un même service peut être consommé par plusieurs applications.
- **Découverte dynamique** : possibilité de publier et de découvrir les services via un registre (ex. UDDI).

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ **Architecture SOA (Service-Oriented Architecture)**

- L'**architecture orientée services (SOA)** repose sur la notion de **services indépendants et réutilisables** qui interagissent via des interfaces bien définies.

Principes de SOA :

- **Faible couplage** : les services ne dépendent pas fortement les uns des autres.
- **Interopérabilité** : les services peuvent communiquer malgré des technologies différentes.
- **Réutilisabilité** : un même service peut être consommé par plusieurs applications.
- **Découverte dynamique** : possibilité de publier et de découvrir les services via un registre (ex. UDDI).

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Architecture SOA (Service-Oriented Architecture)

Exemple :

Une architecture d'entreprise peut avoir :

- un service Facturation,
 - un service Livraison,
 - un service Stock,
- tous accessibles via un bus d'intégration (ESB).

Évolution :

SOA a inspiré les architectures **microservices**, plus légères et découplées.

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Protocoles utilisés

Les web services utilisent différents **protocoles de transport** selon le contexte et les besoins :

Protocole	Type	Usage principal	Caractéristiques
HTTP	Application	REST, GraphQL, SOAP	Basé sur requêtes/réponses (GET, POST, PUT, DELETE)
HTTPS	Application	REST, SOAP, gRPC	Version sécurisée de HTTP via SSL/TLS
TCP	Transport	gRPC, WebSockets	Connexion persistante, fiable, orientée flux
UDP	Transport	Streaming, IoT	Rapide, sans garantie de livraison

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Protocoles utilisés

Exemple :

- REST → HTTP/HTTPS
- gRPC → HTTP/2 (donc basé sur TCP)
- SOAP → HTTP ou SMTP (selon le contexte)

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Formats d'échange de données

Format	Type	Description	Exemple
XML	Texte structuré	Format verbeux, typé, extensible	<person><name>Ali</name></person>
JSON	Texte léger	Format clé-valeur, lisible et efficace	{"name": "Ali"}
Protobuf	Binaire	Format compact et rapide utilisé par gRPC	Encodage binaire, non lisible humainement

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Formats d'échange de données

Format	Type	Description	Exemple
XML	Texte structuré	Format verbeux, typé, extensible	<person><name>Ali</name></person>
JSON	Texte léger	Format clé-valeur, lisible et efficace	{"name": "Ali"}
Protobuf	Binaire	Format compact et rapide utilisé par gRPC	Encodage binaire, non lisible humainement

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Formats d'échange de données

Comparaison :

Critère	XML	JSON	Protobuf
Lisibilité	★★★	★★★★★	★
Poids des messages	⚖️ Lourd	⚖️ Léger	⚖️ Très léger
Performance	⚡ Moyenne	⚡ Rapide	⚡ ⚡ Très rapide
Typage fort	Oui	Non	Oui
Standard courant	SOAP	REST	gRPC

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Différence entre SOAP, REST, GraphQL et gRPC

Critère	SOAP	REST	GraphQL	gRPC
Protocole	HTTP, SMTP	HTTP	HTTP	HTTP/2
Format	XML	JSON	JSON	Protobuf
Type	Contrat rigide (WSDL)	Flexible	Requêtes dynamiques	Contrat rigide (.proto)
Performance	Moyenne	Bonne	Très bonne	Excellente

Introduction & SOAP Web Services

1. Concepts fondamentaux

◆ Différence entre SOAP, REST, GraphQL et gRPC

Critère	SOAP	REST	GraphQL	gRPC
Interopérabilité	Très élevée	Très élevée	Bonne	Moyenne (nécessite support Protobuf)
Streaming	Non	Non	Oui (via WebSockets)	Oui (natif bidirectionnel)
Usage typique	Services bancaires, gouvernementaux	API web classiques	API front modernes	Microservices internes, performance

Introduction & SOAP Web Services

1. Concepts fondamentaux

SOAP : ancien, structuré, robuste → pour systèmes critiques

REST : universel, simple, lisible → pour API web

GraphQL: flexible, orienté client → pour front-end moderne

gRPC : rapide, binaire, typé → pour communication interservices



Introduction à gRPC

Introduction à gRPC

Qu'est-ce que gRPC ?

- **gRPC (Google Remote Procedure Call)** est un **framework RPC** (Remote Procedure Call) développé par **Google**.
- Il permet à deux programmes de **communiquer entre eux via le réseau** en appelant des **méthodes distantes** comme si elles étaient locales.

Introduction à gRPC

Qu'est-ce que gRPC ?

Objectif :

Plutôt que d'envoyer des requêtes HTTP avec des JSON (comme dans REST),

👉 tu appelles directement une **fonction distante** avec des **objets typés** (Protobuf).

Exemple :

```
// REST  
GET /api/users/42 → {"id":42, "name":"Ali"}  
  
// gRPC  
UserService.getUser(UserRequest{id:42}) → UserResponse{id:42, name:"Ali"}
```

► Pour le développeur, ça ressemble à un appel de méthode **locale**, mais la communication passe par le réseau.

Introduction à gRPC

Qu'est-ce que gRPC ?

Composants principaux :

Élément	Rôle
Service	Définit les méthodes RPC disponibles
Message	Définit les structures de données échangées
.proto	Fichier de définition du contrat (service + messages)
Stub	Code client généré automatiquement à partir du .proto
Skeleton (Impl)	Code serveur généré automatiquement
Channel	Connexion réseau gérée par gRPC
StreamObserver	Interface Java utilisée pour gérer les flux de données (streaming)

Introduction à gRPC

Comment ça marche en interne ?

1 Étape 1 : Le contrat (.proto)

C'est la base du système : un **fichier de description** partagé entre le client et le serveur.

Il définit :

- les **messages échangés**,
- les **services** (méthodes RPC),
- et les **options de génération de code**.

Introduction à gRPC

Comment ça marche en interne ?

1 Étape 1 : Le contrat (.proto)

Exemple :

```
syntax = "proto3";
package user;

option java_multiple_files = true;
option java_package = "com.example.user";
option java_outer_classname = "UserProto";

service UserService {
    rpc GetUser (UserRequest) returns (UserResponse);
}
message UserRequest {
    int32 id = 1;
}
message UserResponse {
    int32 id = 1;
    string name = 2;
    string email = 3;
}
```

Introduction à gRPC

Comment ça marche en interne ?

2 Étape 2 : Compilation Protobuf

Le compilateur **protoc** transforme le `.proto` en classes pour différents langages :

```
protoc --java_out=. --grpc-java_out=. user.proto
```

→ Génère :

```
UserServiceGrpc.java  
UserRequest.java  
UserResponse.java
```

Introduction à gRPC

Comment ça marche en interne ?

3 Étape 3 : Implémentation du serveur

Côté serveur, tu **implémentes les méthodes RPC** définies dans ton fichier `.proto`.

```
@GrpcService
public class UserServiceImpl extends UserServiceGrpc.UserServiceImplBase {

    @Override
    public void getUser(UserRequest request, StreamObserver<UserResponse> responseObserver) {
        UserResponse response = UserResponse.newBuilder()
            .setId(request.getId())
            .setName("Faïza Guène")
            .setEmail("faiza.guene@example.com")
            .build();
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}
```

Introduction à gRPC

Comment ça marche en interne ?

Étape 4 – Implémentation client

```
ManagedChannel channel = ManagedChannelBuilder
    .forAddress("localhost", 6565)
    .usePlaintext()
    .build();

UserServiceGrpc.UserServiceBlockingStub stub =
    UserServiceGrpc.newBlockingStub(channel);

UserRequest req = UserRequest.newBuilder().setId(1).build();
UserResponse res = stub.getUser(req);

System.out.println(res.getName());
channel.shutdown();
```

Introduction à gRPC

Comment ça marche en interne ?

Étape 5 – Transmission (interne)

1. Le client sérialise `UserRequest` en **Protobuf binaire**.
2. Le message est transmis via **HTTP/2** (connexion persistante).
3. Le serveur déserialise, exécute la méthode, renvoie la réponse.
4. Le client reçoit et déserialise en `UserResponse`.

Introduction à gRPC

HTTP/2 et performance

gRPC exploite **HTTP/2** pour :

- le **multiplexage** : plusieurs flux sur une seule connexion TCP,
- la **compression des entêtes**,
- le **streaming bidirectionnel**,
- la **sécurité TLS native**.

👉 Résultat :

gRPC est **5 à 10 fois plus rapide** que REST (grâce à Protobuf + HTTP/2).

Introduction à gRPC

Types d'appels RPC

Type	Description	Exemple d'usage
Unary	1 requête → 1 réponse	Authentification
Server streaming	1 requête → plusieurs réponses	Envoi d'un flux de données
Client streaming	plusieurs requêtes → 1 réponse	Upload de fichiers
Bidirectional streaming	plusieurs requêtes ➡ plusieurs réponses	Chat, streaming temps réel

Introduction à gRPC

Syntaxe complète du fichier **.proto**

◆ En-tête

```
syntax = "proto3"; // Version du langage Protobuf
package bank; // Namespace logique
```

◆ Options de compilation (annotations)

```
option java_multiple_files = true;
option java_package = "com.example.bank";
option java_outer_classname = "BankProto";
```

“ Ces options indiquent au compilateur comment générer les classes. ”

Introduction à gRPC

Syntaxe complète du fichier **.proto**

◆ Messages

```
message Account {  
    int32 id = 1;  
    double balance = 2;  
    string owner = 3;  
    repeated string tags = 4; // Liste  
}
```

Élément	Signification
message	Déclare une structure de données
int32, double, string	Types primitifs
repeated	Collection (liste)
= 1	Numéro unique du champ (pour la sérialisation binaire)

Introduction à gRPC

Syntaxe complète du fichier `.proto`

◆ Types supportés

Type	Java	Description
int32, int64	int, long	Entier
float, double	float, double	Décimal
bool	boolean	Booléen
string	String	Texte
bytes	ByteString	Binaire
map<K, V>	Map<K,V>	Dictionnaire
repeated	List	Liste d'éléments

Introduction à gRPC

Syntaxe complète du fichier `.proto`

◆ Champs imbriqués et messages internes

```
message BankAccount {  
    int32 id = 1;  
    double balance = 2;  
  
    message Transaction {  
        int32 id = 1;  
        double amount = 2;  
    }  
  
    repeated Transaction history = 3;  
}
```

Introduction à gRPC

Syntaxe complète du fichier **.proto**

◆ Numérotation des champs

- Chaque champ a un identifiant **unique et permanent**.
- Les numéros **1-15** sont plus rapides à encoder (1 octet).
- **⚠ Ne jamais réutiliser un ancien numéro supprimé.**

```
reserved 3, 5;  
reserved "old_field";
```

Introduction à gRPC

Syntaxe complète du fichier **.proto**

◆ Imports et modularisation

```
import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";
```

Exemple :

```
message LogEntry {
    string message = 1;
    google.protobuf.Timestamp created_at = 2;
}
```

Introduction à gRPC

Syntaxe complète du fichier `.proto`

◆ Imports et modularisation

```
import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";
```

Exemple :

```
message LogEntry {
    string message = 1;
    google.protobuf.Timestamp created_at = 2;
}
```

Introduction à gRPC

Syntaxe complète du fichier **.proto**

◆ Services RPC

```
service BankService {  
    rpc GetAccountBalance (BalanceCheckRequest) returns (AccountBalance);  
    rpc Withdraw (WithdrawRequest) returns (stream Money);  
    rpc Deposit (stream DepositRequest) returns (AccountBalance);  
    rpc Transfer (stream TransferRequest) returns (stream TransferResponse);  
}
```

Mot-clé	Description
rpc	Méthode distante
stream	Active le streaming
returns	Type de réponse
service	Conteneur de RPCs

Introduction à gRPC

Syntaxe complète du fichier `.proto`

◆ Bonnes pratiques de nommage

-  Nom des messages : **PascalCase** (`UserRequest`, `BankAccount`)
-  Nom des champs : **snake_case** (`account_number`, `created_at`)
-  Nom des services : **PascalCase** (`BankService`)
-  Nom des méthodes : **CamelCase** (`GetBalance`, `WithdrawMoney`)

Introduction à gRPC

Protobuf – Types avancés et bien connus

Google fournit des **types standards** réutilisables :

Type	Description
google.protobuf.Empty	Message vide (équivalent void)
google.protobuf.Timestamp	Représente une date/heure
google.protobuf.Duration	Durée
google.protobuf.Any	Type générique (polymorphisme)

Exemple :

```
import "google/protobuf/timestamp.proto";

message Transaction {
    double amount = 1;
    google.protobuf.Timestamp date = 2;
}
```

Introduction à gRPC

Les Stubs (clients)

Après compilation du `.proto`, le code généré contient 3 types de stubs :

Type	Description
BlockingStub	Appel synchrone
FutureStub	Asynchrone (basé sur Future)
AsyncStub	Basé sur callback StreamObserver

Exemple :

```
BankServiceGrpc.BankServiceBlockingStub stub = BankServiceGrpc.newBlockingStub(channel);
AccountBalance res = stub.getAccountBalance(req);
```

Introduction à gRPC

Sécurité dans gRPC

Niveau	Mécanisme
Transport	TLS / SSL intégré à HTTP/2
Authentification	Intercepteurs (<code>ServerInterceptor</code> , <code>ClientInterceptor</code>)
Autorisation	Jetons JWT ou métadonnées (<code>Metadata</code>)

Exemple :

```
Metadata md = new Metadata();
md.put(Metadata.Key.of("authorization", Metadata.ASCII_STRING_MARSHALLER), "Bearer <token>");
```

Introduction à gRPC

Écosystème gRPC

Outil	Rôle
protoc	Compilateur Protobuf
grpcurl	Tester les services depuis le terminal
BloomRPC / Insomnia	Interface graphique pour gRPC
grpc-gateway	Convertit une API gRPC en REST
Spring Boot gRPC Starter	Intégration facile avec <code>@GrpcService</code> , <code>@GrpcClient</code>

Introduction à gRPC

Avantages majeurs

Avantage	Détail
 Performance	Encodage binaire rapide + HTTP/2
 Typage fort	Contrat partagé via <code>.proto</code>
 Interopérabilité	Multi-langages (Java, Go, Python...)
 Streaming	Bidirectionnel natif
 Évolutif	Ajout de champs sans casser l'existant

Introduction à gRPC

Limites

Limite	Description
 Pas accessible via navigateur	Pas de support HTTP/1.1
 Moins intuitif que REST	Format binaire moins lisible
 Versioning manuel	Gestion des <code>.proto</code> requise
 Outils de test limités	grpcurl ou BloomRPC nécessaires

Introduction à gRPC

Cas d'usage typiques

Cas	Pourquoi gRPC ?
Microservices internes	Performance + typage
IoT / streaming temps réel	Support du flux bidirectionnel
Systèmes distribués	Stabilité et scalabilité
Backend multi-langages	Java ↔ Python ↔ Go
Migration REST → gRPC	Gain de bande passante



Introduction à GraphQL

Introduction à GraphQL

Pourquoi GraphQL ?

GraphQL a été inventé pour répondre aux limites du modèle REST, en particulier dans des environnements où le frontend a besoin de données **flexibles, précises, rapides et optimisées** (ex : applications mobiles, plateformes complexes, microservices).

Introduction à GraphQL

Pourquoi GraphQL ?

GraphQL apporte :

- **Un seul endpoint (`/graphql`)**
- **Des requêtes flexibles (tu demandes exactement ce que tu veux)**
- **Pas d'excès de données (pas d'overfetching)**
- **Pas de manque de données (pas d'underfetching)**
- **Moins d'allers-retours HTTP**
- **Une documentation automatique**
- **Un typage fort via le schema**
- **La possibilité de récupérer plusieurs ressources dans un seul call**

Introduction à GraphQL

Les 6 grandes limites de REST

1 REST → Overfetching

➤ Problème

Quand tu appelles une API REST, tu reçois généralement **plus de données que nécessaire**.

2 REST → Underfetching

➤ Problème

Une route REST renvoie souvent **trop peu** de données.

Tu dois en appeler plusieurs pour composer une page.

Introduction à GraphQL

Les 6 grandes limites de REST

3 REST → Trop d'endpoints (explosion du nombre de routes)

Dans un gros produit, tu vas avoir :

```
/users  
/users/:id  
/users/:id/posts  
/users/:id/posts/:id  
/articles  
/articles/:id/comments  
...
```

➤ Problème :

- La documentation devient difficile à maintenir
- Il y a des incohérences entre services
- Les versions s'accumulent (`/v1`, `/v2`, `/v3...`)

GraphQL → **un seul endpoint** : `/graphql`.

Tout est décrit dans le **schema**, pas dans l'URL.

Introduction à GraphQL

Les 6 grandes limites de REST

4 REST → Pas flexible pour le client

Dans REST, le serveur impose la structure de la réponse.

Dans GraphQL, **le client choisit la structure.**

REST :

```
{  
  "id": 1,  
  "title": "Hello",  
  "content": "..."  
}
```

Impossible de demander :

- juste `title`
- ou `title + author + 3 derniers commentaires`

GraphQL te permet d'adapter la réponse **à ton écran.**

Introduction à GraphQL

Les 6 grandes limites de REST

5 REST → Versioning lourd

REST évolue souvent comme ça :

- /api/v1/users
- /api/v2/users
- /api/v3/users

Car :

- on ajoute des champs
- on modifie des structures
- on déprécie des routes

GraphQL, lui, **ne nécessite pas de versioning** :

- On ajoute des champs
- On déprécie des champs avec `@deprecated`
- Les clients restent compatibles

Introduction à GraphQL

Les 6 grandes limites de REST

6 REST → Difficile d'agréger plusieurs microservices

Dans une architecture microservices, pour construire une page, il faut souvent :

- appeler le service User
- appeler le service Orders
- appeler le service Products
- assembler tout

Côté frontend, ou via une API Gateway.

GraphQL permet :

- ✓ D'agréger plusieurs sources de données
- ✓ De faire le *data stitching*
- ✓ Ou de faire de la **fédération GraphQL**
- ✓ De simplifier l'exposition de données hétérogènes

Le client ne voit qu'un **schema unique**.

Introduction à GraphQL

Le rôle de Facebook dans la création de GraphQL

Étape	Rôle de Facebook
2012	Invention de GraphQL en interne
2013-2015	Utilisation en production dans toutes les apps mobiles
2015	Open-source officiel lors de React Conf
2015-2019	Mainteneur principal de l'écosystème
2019	Transfert à la GraphQL Foundation
Aujourd'hui	Contributeur majeur + utilisation massive

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

GraphQL repose sur **3 éléments qui travaillent ensemble** :

1. **Schema** → ce qui est *possible* de demander
2. **Query** → ce que le *client* demande
3. **Resolver** → comment *obtenir* chaque donnée

GraphQL ne fonctionne que grâce à cette collaboration.

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

1 Schema GraphQL – *le contrat ou le plan du serveur*

Le **schema** décrit tout ce que l'API GraphQL peut faire.

Il contient :

✓ Les types (model)

Exemple :

```
type User {  
  id: ID!  
  name: String!  
  email: String!  
}
```

✓ Les opérations disponibles

Deux types principaux :

- **Query** (lecture)
- **Mutation** (écriture)
- **Subscription** (temps réel – optionnel)

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

1 Schema GraphQL – *le contrat ou le plan du serveur*

Exemple :

```
type Query {  
  users: [User!]!  
  user(id: ID!): User  
}  
  
type Mutation {  
  createUser(name: String!, email: String!): User!  
}
```

✓ Les Input types (pour les mutations complexes)

```
input NewUserInput {  
  name: String!  
  email: String!  
}
```

À retenir

Le **schema**, c'est la description de toutes les questions que le client peut poser, et de toutes les réponses que le serveur peut donner.

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

2 Query GraphQL – *la question posée par le client*

Une **requête GraphQL**, c'est le client qui dit :

- 👉 exactement quelles données il veut
- 👉 exactement quels champs il veut

Exemple :

```
query {  
  user(id: 1) {  
    name  
    email  
  }  
}
```

Ici, le client demande :

- Le user #1
- Avec uniquement : `name` et `email`

Pas plus. Pas moins.

Le client contrôle totalement la **forme de la réponse**.

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

3 Resolvers – *les fonctions qui fournissent la donnée*

Les **resolvers** sont des fonctions côté backend qui disent au serveur :

👉 *Comment obtenir la valeur de chaque champ demandé dans la Query ?*

Chaque champ GraphQL **peut** avoir son resolver.

Exemple en pseudo-code Node.js :

```
const resolvers = {
  Query: {
    user: (_, args) => db.users.findById(args.id)
  },
  User: {
    email: (parent) => parent.email,
    name: (parent) => parent.name
  }
};
```

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

En Spring Boot ce serait :

```
@QueryMapping
public User getUser(@Argument Long id) {
    return userService.findById(id);
}

@SchemaMapping
public List<Post> posts(User user) {
    return postService.findByUserId(user.getId());
}
```

Résolveurs = *Logique métier*

Les resolvers :

- vont chercher les données (BD, APIs, microservices)
- vérifient les permissions
- transforment les données si besoin
- agrègent plusieurs sources

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

Comment les 3 travaillent ensemble ?

Voici le flow complet :

1. Le client envoie une Query :

```
query {  
  user(id: 42) {  
    name  
    email  
  }  
}
```

2. Le serveur GraphQL lit le Schema :

- Existe-t-il un type `User` ?
- Existe-t-il un champ `user(id)` dans Query ?
- `name` et `email` existent-ils dans le type User ?

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

Comment les 3 travaillent ensemble ?

Voici le flow complet :

3. Le moteur appelle les Resolvers :

- `Query.user` → pour récupérer le user 42
- `User.name` → pour récupérer le nom
- `User.email` → pour récupérer l'email

4. Le serveur renvoie la réponse

```
{  
  "data": {  
    "user": {  
      "name": "Mohamed",  
      "email": "m@m.com"  
    }  
  }  
}
```

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver

Le triptyque résumé en 1 phrase

Le Schema dit ce qui est possible

La Query dit ce qui est demandé

Le Resolver dit comment récupérer les données

C'est comme un restaurant :

Élément GraphQL	Métaphore restaurant
Schema	La carte du restaurant
Query	La commande du client
Resolver	Le chef qui prépare le plat

Introduction à GraphQL

Le triptyque de GraphQL : Schema – Query – Resolver**

Le triptyque résumé en 1 phrase

Le Schema dit ce qui est possible

La Query dit ce qui est demandé

Le Resolver dit comment récupérer les données

C'est comme un restaurant :

Élément GraphQL	Métaphore restaurant
Schema	La carte du restaurant
Query	La commande du client
Resolver	Le chef qui prépare le plat



Le langage GraphQL

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

Le Schéma GraphQL (Schema Definition Language – SDL)

Le **schéma GraphQL** est écrit en **SDL (Schema Definition Language)** et décrit :

- les **types** exposés par l'API
- les **opérations** possibles (Query, Mutation, Subscription)
- les **valeurs scalaires**
- les **structures d'entrée**
- les **énumérations**
- les **relations entre objets**

C'est **le contrat** entre le client et le serveur.

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

1 **type** → Définition d'un objet

type sert à définir un **objet** avec des champs.

Exemple :

```
type User {  
  id: ID!  
  name: String!  
  email: String!  
  age: Int  
  active: Boolean!  
}
```

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

1 **type** → Définition d'un objet

Explications :

- `User` est un objet du schéma.
- Chaque champ a :
 - un nom : `id`, `name`, `email`...
 - un type : `ID`, `String`, `Int`, `Boolean`
 - `!` signifie *non nul*.

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

1 **type** → Définition d'un objet

Types des champs :

Nom du type	Description
String	du texte
Int	entier
Float	nombre décimal
Boolean	true/false
ID	identifiant unique (souvent string)

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

2 **Query** → Entrée principale pour *lire* des données

Chaque API GraphQL doit définir un type **Query**.

Exemple :

```
type Query {  
  users: [User!]!  
  user(id: ID!): User  
}
```

Signification :

- **users** → retourne une liste d'utilisateurs
- **user(id: ID!)** → retourne un user qui correspond à l'ID

Règle :

👉 Toutes les requêtes GraphQL commencent obligatoirement dans **Query**.

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

3 Mutation → Modifier, créer, supprimer des données

Mutation permet de définir des opérations d'écriture :

- création
- mise à jour
- suppression

Exemple :

```
type Mutation {  
    createUser(input: NewUserInput!): User!  
    deleteUser(id: ID!): Boolean!  
}
```

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

3 Mutation → Modifier, créer, supprimer des données

Interprétation :

- `createUser` crée un utilisateur et renvoie le `User` créé.
- `deleteUser` renvoie un `Boolean` (true si suppression réussie).

Règle :

Les opérations de mutation ressemblent à des fonctions.

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

4 **Scalar** → Types primifs (ou scalaires)

Un **scalar** est une valeur primitive, non composée :

GraphQL propose **5 scalaires natifs** :

Scalar	Description
Int	entier
Float	réel
String	texte
Boolean	booléen
ID	identifiant unique (souvent string opaque)

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

4 **Scalar** → Types primitifs (ou scalaires)

Tu peux créer des scalars personnalisés :

```
scalar DateTime
```

Et ensuite l'utiliser dans un type :

```
type Post {  
  id: ID!  
  title: String!  
  publishedAt: DateTime!  
}
```

Le resolver définira comment parser/formater ce scalar.

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

5 **Enum** → Ensemble de valeurs possibles

Enum permet d'imposer une liste de valeurs autorisées.

Exemple :

```
enum Role {  
    ADMIN  
    USER  
    GUEST  
}
```

Usage :

```
type User {  
    id: ID!  
    name: String!  
    role: Role!  
}
```

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

5 **Enum** → Ensemble de valeurs possibles

Avantages :

- Pas d'erreur de typo
- Documentation automatique
- Typage strict
- Parfait pour les statuts, types, rôles, catégories...

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

6 **Input** → Structure d'entrée pour les mutations

Les mutations ont souvent besoin de champs complexes (objets).

On utilise donc `input`.

Exemple :

```
input NewUserInput {  
  name: String!  
  email: String!  
  age: Int  
}
```

Utilisation dans une mutation :

```
type Mutation {  
  createUser(input: NewUserInput!): User!  
}
```

Introduction à GraphQL

Le schéma : `type`, `Query`, `Mutation`, `Scalar`, `Enum`, `Input`

6 `Input` → Structure d'entrée pour les mutations

Pourquoi utiliser `input` ?

- Sépare clairement les données d'entrée des types de sortie
- Évite les ambiguïtés
- Permet de valider les champs structurés

Règle :

Un type `input` ne peut contenir que des scalars, `enum`, ou d'autres `input`, mais **pas des types "object"**.

Introduction à GraphQL

Le schéma : **type**, **Query**, **Mutation**, **Scalar**, **Enum**, **Input**

Bonus : Relations dans les types

GraphQL gère naturellement les relations :

```
type User {  
  id: ID!  
  name: String!  
  posts: [Post!]!  
}  
  
type Post {  
  id: ID!  
  title: String!  
  author: User!  
}
```

Grâce à ça, GraphQL peut faire des "jointures" dans une seule requête.

