Formation REST API : Architecture conception et sécurité.

Didier Curvier MALEMBE
Expert Informatique & Système d'information
Formateur IT

Table des matières

I. Introduction

L'architecture logicielle décrit les modèles et les techniques utilisées pour concevoir et créer une application. Dans un contexte où la complexité des systèmes informatiques est sans cesse grandissante, l'architecture logicielle se présente comme une feuille de route qui favorise les meilleures pratiques de développement logiciel.

De tout temps, la recherche sur l'architecture logicielle étudie :

- **les méthodes qui permettent de partitionner un système complexe,**
- **La façon dont les composants logiciels issues de cette partition s'identifie et communiquent,**
- **Comment ces informations sont-elles communiquées ? et ,**
- **La façon dont ces composants peuvent évoluer indépendamment**

Les réponses formulées pour ces questions, distinguent les styles architecturaux qui sont un ensemble de contraintes coordonnées visant la création d'applications.

Notre réflexion dans le cadre de cette formation s'articule autour du style architectural **REST *(Representational State Transfer)***, et sur la façon dont le **REST** impacte le développement d'applications web modernes.

A. Les architectures n-tiers, applications et APIs.

L'architecture N-Tiers est une approche modulaire qui divise une application en plusieurs couches logiques distinctes. Elle est particulièrement utilisée dans les applications web et d'entreprise pour améliorer la scalabilité, la maintenabilité et la répartition des responsabilités.

1. Définition

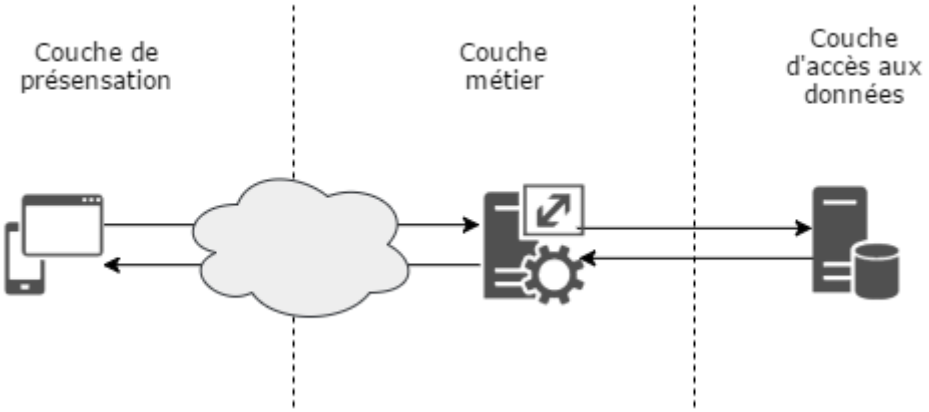
L'architecture **N-Tiers** repose sur la séparation des préoccupations en plusieurs niveaux (ou "tiers"), où chaque tiers a un rôle bien défini. Le **"N"** représente un nombre variable de couches, mais les plus courantes sont :

- **2-Tiers (Client-Serveur)**
- **3-Tiers (Présentation - Métier - Données)**
- **N-Tiers (ajout de services intermédiaires comme cache, API, sécurité, etc.)**

2. Le modèle 3-tiers (Le plus courant)

- **Tier 1** : Couche Présentation (Front-end) Interface utilisateur (site web, application mobile, client lourd, etc.).
  Technologies : HTML/CSS, JavaScript, React, Angular, Vue.js.

- **Tier 2** : Couche Métier (Back-end) Contient la logique métier et les règles de gestion.
  Technologies : Java (Spring Boot), Node.js, .NET, Python (Django/Flask), PHP (Laravel).
- **Tier 3** : Couche Données (Database) Gère la persistance et les requêtes de données.
  Technologies : MySQL, PostgreSQL, MongoDB, Firebase.



3. Avantages & Inconvénients

Modularité : Facile à mettre à jour indépendamment.
Scalabilité : Chaque couche peut être optimisée séparément.
Sécurité : Moins d'accès directs à la base de données.
Réutilisation : Possibilité de connecter différents clients *(Web, Mobile, API)*.

Complexité accrue : Nécessite une bonne gestion des communications entre couches.
Latence potentielle : Plus de couches peuvent ralentir les performances.
Maintenance plus coûteuse : Chaque couche peut nécessiter des ressources différentes.

4. Autres variantes N-tiers

4-Tiers : Ajout d'un proxy, d'un cache (Redis) ou d'un API Gateway.
5-Tiers et plus : Séparation encore plus fine (Microservices, CQRS, etc.).

5. Qu'une API (Application Programming Interface) ?

Une **API (Application Programming Interface)** est un ensemble de règles et de protocoles permettant à des applications logicielles de communiquer entre elles. Elle définit comment des composants logiciels doivent interagir en exposant des **fonctions, des services ou des données** à d'autres applications.

Les API peuvent être utilisées pour **accéder à une base de données, interagir avec un service distant, ou intégrer des fonctionnalités tierces**.

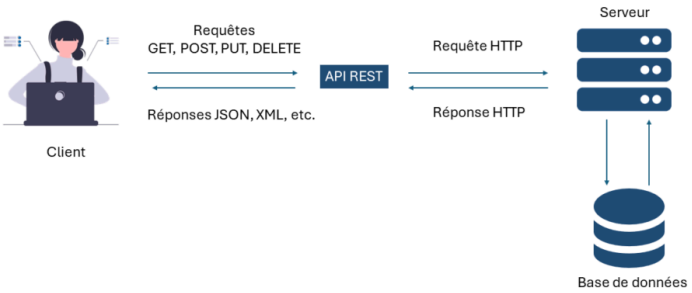
Historiquement, l'échange de données se fait par échanges de fichiers en utilisant un canal de diffusions plus ou moins sécurisé ou fiable *(envoi par mail, ftp/sftp, ...)*. Cela implique des problèmes de sécurité et aussi d'uniformité des échanges.

Les **APIs** offrent un moyen standard pour qu'un fournisseur expose ses services d'une manière uniforme à tous ses consommateurs.

D'un point de vue consommateur, **l'API** permet d'exploiter les données exposées par un fournisseur sans se soucier des technologies utilisées par le fournisseur pour cette exposition.

Les **APIs** sont une solution pour répondre aux défis liés à la sécurité, la réutilisation et le monitoring.

API RESTful : Utilise HTTP et suit les principes REST.
API SOAP : Basée sur XML et plus formelle que REST.
API GraphQL : Permet aux clients de récupérer uniquement les données nécessaires.



Avant d'aborder en profondeur l'architecture **REST**, il est urgent de comprendre les fondamentaux du **protocole HTTP**, qui est à la base de l'approche **REST**.

Comprendre le protocole HTTP en détail.

Rappels TCP , HTTP et persistence

- **TCP (Transmission Control Protocol)** est un protocole de transport garantissant la transmission fiable des données entre deux machines.

- **HTTP (HyperText Transfer Protocol)** est un protocole applicatif basé sur TCP, utilisé pour la communication entre clients (navigateur, API) et serveurs web.

HTTP utilise **TCP** comme protocole de transport sous-jacent, une connexion **TCP** est établie avant l'échange des requêtes **HTTP**.

HTTP Persistant vs Non-Persistant

- **HTTP 1.0** : Par défaut, chaque requête ouvre et ferme une connexion TCP, ce qui est coûteux en termes de performance.
- **HTTP 1.1** (Persistant) : Introduit la persistance avec **Connection: keep-alive**, permettant de réutiliser la connexion TCP pour plusieurs requêtes.

Les PDU(Protocol Data Unit) HTTP : GET, POST, PUT, DELETE, HEAD et TRACE

Les méthodes HTTP

Il existe 11 méthodes HTTP

GET: Récupère une ressource sans modifier l'état du serveur.

POST: Envoie des données au serveur pour créer une nouvelle ressource.

PUT: Met à jour ou crée une ressource spécifique avec un contenu fourni.

DELETE : Supprime une ressource identifiée par une URL.

HEAD: Similaire à GET, mais ne retourne que les en-têtes HTTP, sans le corps.

TRACE: Diagnostique le chemin d'une requête HTTP en renvoyant l'intégralité de la requête reçue par le serveur.

OPTION : Permet de savoir à quelles méthodes va répondre le serveur.

Deux approches sont possibles :

- **OPTIONS * HTTP/1 . 1** : On demande toutes les méthodes autorisées pour l'ensemble des ressources du serveur.
- **OPTIONS /maRessource HTTP/1. 1 :** On demande les méthodes autorisées pour une ressource spécifique

En retour, le serveur doit indiquer avec un code **200** la liste des méthodes autorisées dans l'en-tête **ALLOW**

Les En-tête HTTP

Voici une liste non-exhaustive d'en-tête **HTTP** intéressantes dans le cadre d'implémentation **REST**

Accept-Charset, Accept-Encoding

Ces en-tête permettent de savoir que le serveur et le client peuvent communiquer.

Accept-Charset : Indique le ou les jeux de caractères, supportés par le client et séparés par des (,).

* signifie que n'importe quel caractère est accepté.

```
Accept-Charset : ISO-8859-1,UTF-8

Accept-Charset : *
```

Accept-Encoding : indique les encodages supportés par l'application séparés par des (,). Les valeurs usuelles sont compress et gzip. * signifie que n'importe quel type d'encodage est accepté.

```
Accept-Encoding : compress, gzip

Accept-Encoding : *
```

Accept-Language et Content-Language

L'en-tête **Accept-Language** est envoyé dans une requête, tandis que **Content-Language** est envoyé dans la réponse. Ces deux en-tête désignent la langue utilisée dans le contenu.

```
Accept-Language = fr-fr

Content-Language = fr-fr
```

Age

L'en-tête Age est utilisé dans une réponse, il indique l'âge d'une ressource en seconde. Obligatoire lorsqu'un serveur sert du contenu depuis le cache.

```
Age = 50
```

Allow

L'en-tête est ajouté dans une réponse et indique la (les) méthode (s) autorisée par le serveur.

Si l'application appelante appelle une autre méthode, le serveur doit lui fournir le code 405 *(Method Not Allowed).*

```
Allow : GET, POST, DELETE
```

Accept-Type & Content-Type

Ces en-têtes sont parmi les plus importants lorsqu'une application fournit des services REST dont les réponses peuvent être dans plusieurs formats.

Accept-Type :

L'en-tête Accept-Type se trouve dans la requête, il définit le ou les types de données que l'application peut gérer.

```
Accept : * / *
```

```
Accept : application/json
```

```
Accept : text/*, text/plain, text/html
```

```
Accept : application/xml
```

Si le serveur n'est pas capable de retourner l'un des formats demandés, il devra fournir une réponse de code 406(Not Acceptable).

Content-Type :

L'en-tête Content-Type positionné dans la réponse indique le type de données retourné par l'application

```
Content-Type : application/json
```

```
Content-Type : text/html; charset = UTF-8
```

Authorization

L'en-tête est ajouté à une requête lorsque l'accès à une ressource nécessite une authentification.

```
Authorization : Basic : ezrorijfjhdyyeoel==
```

Codes de statut HTTP (1xx - 5xx)

1xx (Informational) : La requête est en cours de traitement.

2xx (Succès) : La requête a réussi (200 OK, 201 Created).

Quoi que le client ait tenté de faire, cela a réussi jusqu'au point où la réponse a été envoyée. Gardez à l'esprit qu'un statut comme **202 Accepted** ne dit rien sur le résultat réel ; il indique simplement que la requête a été acceptée et est en cours de traitement de manière asynchrone.

3xx (Redirection) : La ressource a été déplacée (301 Moved Permanently, 302 Found).

Il s'agit ici de rediriger l'application appelante vers un autre emplacement pour accéder à la ressource réelle. Les plus connus de ces statuts sont **303 See Other** et **301 Moved Permanently**, couramment utilisés sur le web pour rediriger un navigateur vers une autre URL.

4xx (Erreur client) : Erreur côté client *(400 Bad Request, 401 Unauthorized, 404 Not Found)*.

Avec ces codes de statut, nous indiquons que le client a fait quelque chose d'invalide et qu'il doit corriger la requête avant de la renvoyer.

5xx (Erreur serveur): Erreur côté serveur (500 Internal Server Error, 503 Service Unavailable).

Avec ces codes de statut, nous indiquons qu'une erreur s'est produite au niveau du service. Par exemple, une connexion à la base de données a échoué. En général, une application cliente peut réessayer la requête. Le serveur peut même spécifier quand le client est autorisé à réessayer la commande en utilisant l'en-tête HTTP **Retry-After**.

Voici une liste non exhaustive de codes statut et leur contexte d'utilisation.

- **200** – Tout est **OK** de manière générique
- **201** – Création réussie d'une ressource
- **202** – Requête acceptée mais en cours de traitement asynchrone ***(pour une vidéo : encodage, pour une image : redimensionnement, etc.)***
- **400** – Mauvaise requête **(devrait en principe être réservé à une syntaxe invalide, mais certains l'utilisent aussi pour les erreurs de validation)**
- **401** – Non autorisé **(aucun utilisateur connecté alors qu'il devrait y en avoir un)**
- **403** – L'utilisateur actuel n'est pas autorisé à accéder à cette donnée
- **404** – L'URL ne correspond à aucune route valide, ou la ressource demandée n'existe pas
- **405** – Méthode non autorisée ***(votre framework s'en chargera probablement automatiquement)***
- **410** – La donnée a été supprimée, désactivée, suspendue, etc.

- **415** – Le type de contenu *(Content-Type)* de la requête n'est pas reconnu ou pris en charge par le serveur
- **429** – Limite de débit atteinte (Rate Limited), cela signifie : faites une pause, attendez un peu, puis réessayez
- **500** – Une erreur inattendue est survenue, c'est la faute de l'API
- **503** – L'API est temporairement indisponible, veuillez réessayer plus tard

Les formats de sortie et leurs type MIME *(Multi-Purpose-Internet Mail Extensions)*

Un **type MIME** représente le format de données transmises sur internet.

Un **type MIME** (Multipurpose Internet Mail Extensions) est une **chaîne qui indique le format des données envoyées ou reçues** via Internet.

Il est composé de deux parties **type** et **sous-type**, séparé par un / . **(type/sous-type)**

On distingue :

- **text/Html**
- **text/csv**
- **application/json**
- **application/xml**
- **image/jpeg**

Liens vers les ressources en REST :

Le **REST** est une architecture centrée sur les ressources de l'application. On réalise des opérations sur les ressources *(Lecture - Création - Mise à jour, etc)* à l'aide de verbes, qui sont des méthodes du protocole **HTTP**.

On peut dégager de manière générale la structure de liens suivantes :

Action sur l'ensemble des ressources d'un type

De manière usuelle pour exécuter une action sur l'ensemble des ressources d'un type **T**, il faut appeler une URL de forme **/T .**

GET /books

Un appel **GET** sur un lien **/books** permet de lire l'ensemble des objets de type **Book** disponibles.

Action sur une ressource spécifique

Pour exécuter une action sur une ressource spécifique de type **T**, il faut appeler une URL de forme **/T** en ajoutant en plus une ou plusieurs variables discriminantes **V** pour obtenir une URL de forme **/T/V**.

GET /books/1

L'appel **GET** sur /books/1 retourne le livre d'identifiant 1.

Action sur une ressource liée

Pour exécuter une action sur une ressource **A** d'un type **TA** liée à une ressource **B** d'un type **TB**, qui possède une variable discriminante **VB**.

Il faut appeler une URL de forme **/TB/VB/TA**

GET /categories/1/books

Cet appel retourne tous les livres de la catégorie d'identifiant 1.

Comprendre les principes du REST

En matière de conception architecturale, qu'il s'agisse de génie civil ou de génie informatique, l'une des approches qui permettent de modéliser les systèmes complexes est toujours de partir de **ZÉRO**.

Conformément à ce point de vue, le concepteur part d'une feuille blanche et construit une architecture à partir de composants basiques et familiers jusqu'à ce qu'elle réponde aux besoins du système envisagé.

Le style architectural REST n'est pas en marge de cette approche car en effet, il a été développé par une application incrémentielle d'un ensemble de contraintes.

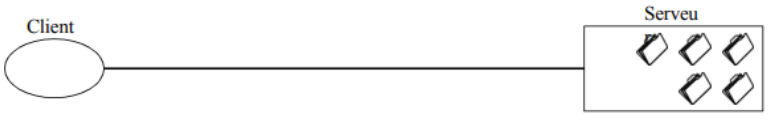
Le point de départ étant le **style Null** (ensemble vide de contraintes) en d'autres termes le style Null décrit un système dans lequel il n'y a pas de frontières entre les composants.

Contrainte 1 : Client-Serveur

Les premières contraintes ajoutées à notre **style Null** de départ sont celles du style architectural **client serveur**. Cette séparation des préoccupations est la première condition d'une architecture REST.

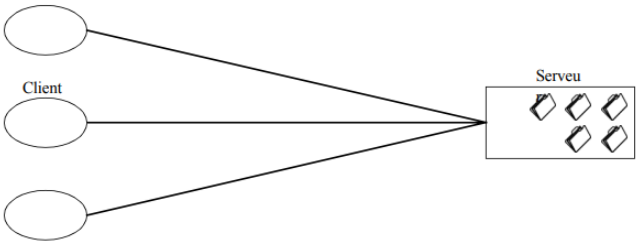
En séparant les préoccupations relatives à l'interface utilisateur des préoccupations relatives au traitement et stockage des données, on améliore la portabilité de l'interface utilisateur sur plusieurs plateformes et l'évolutivité en simplifiant les composants du serveur.

De plus, pour le web, cette claire séparation entre le client et le serveur permet aux composants d'évoluer indépendamment.



Contrainte 2 : Sans État (Stateless)

Nous ajoutons ensuite une contrainte à l'interaction client-serveur : la communication doit être sans état par nature, de sorte que chaque demande du client au serveur doit contenir toutes les informations nécessaires à l'exécution de la tâche.



L'état de la session est donc entièrement conservé sur le client.

Cette contrainte induit les propriétés de **visibilité**, de **fiabilité** et **d'évolutivité** :

La **visibilité** est améliorée parce qu'un système de surveillance n'a pas besoin de regarder au-delà d'une seule donnée de demande pour déterminer la nature complète de la demande.

La **fiabilité** est améliorée parce qu'elle facilite la récupération des défaillances partielles.

L'évolutivité est améliorée parce que le fait de ne pas avoir à stocker l'état entre les demandes permet au composant serveur de libérer rapidement des ressources, et simplifie encore la mise en œuvre parce que le serveur n'a pas à gérer l'utilisation des ressources entre les demandes.

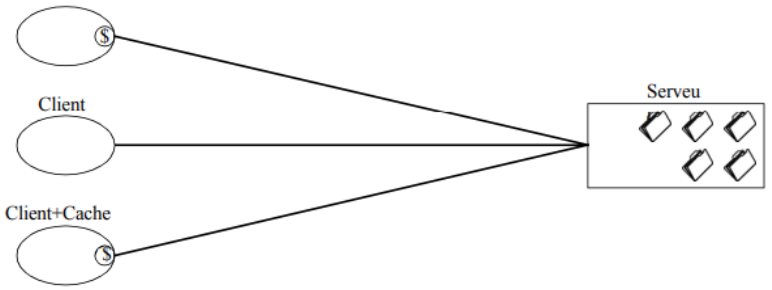
L'inconvénient est qu'elle peut diminuer les performances du réseau en augmentant les données répétitives (frais généraux par interaction) envoyées dans une série de requêtes, puisque ces données ne peuvent pas être laissées sur le serveur dans un contexte partagé.

Contrainte 3 : La mise en Cache

Afin d'améliorer l'efficacité du réseau, nous ajoutons des contraintes de cache pour former le style **client-cache-serveur sans état**.

Les contraintes de cache exigent que les données contenues dans une réponse à une requête soient implicitement ou explicitement étiquetées comme pouvant être mises en cache ou non.

Si une réponse peut être mise en cache, le cache du client a le droit de réutiliser les données de la réponse pour des demandes équivalentes ultérieures.



L'avantage de l'ajout de contraintes de cache est qu'il permet d'éliminer partiellement ou totalement certaines interactions, ce qui améliore l'efficacité, l'évolutivité et la performance perçue par l'utilisateur en réduisant la latence moyenne d'une série d'interactions.

Contrainte 4 : Uniformité de l'interface

La caractéristique centrale qui distingue le style **architectural REST** des autres styles basés sur les réseaux est l'accent mis sur une interface uniforme entre les composants.

Afin d'obtenir une interface uniforme, de multiples contraintes architecturales sont nécessaires pour guider le comportement des composants.

REST est défini par quatre contraintes d'interface : **l'identification des ressources**, **la manipulation des ressources par le biais de représentations**, **les messages auto-descriptifs** et **les hypermédias comme moteur de l'état de l'application**.

Contrainte 5 : Système en plusieurs niveaux

Contrainte 6 : Code à la demande

Le dernier ajout à notre ensemble de contraintes pour **REST** provient du style de code à la demande. **REST** permet d'étendre les fonctionnalités du client en téléchargeant et en exécutant du code sous forme d'applets ou de scripts.

B. Différences essentielles entre API REST et API SOA.

Le **SOA** est une **architecture logicielle** de **haut niveau** qui vise à organiser les fonctionnalités d'un système en **services réutilisables** accessibles à travers un réseau.

- Il s'agit donc d'un **style d'architecture** (pas d'un protocole ou une technologie spécifique).
- SOA repose souvent sur des **standards formels** :

  ➢ **WSDL** : Le **WSDL** ou **Web Services Description Language** est une grammaire **XML** permettant de décrire un service web. **WSDL 1.1** a été proposé en 2001 au **W3C** pour standardisation mais n'a pas été approuvé.
  La version 2.0 a été approuvée le 27 juin 2007 et est désormais une recommandation officielle du **W3C**.

  ➢ **Une description WSDL** est un document **XML** qui commence par la balise **<definitions>** et qui contient les balises suivantes :
  1. **<binding>** : définit le protocole à utiliser pour invoquer le service web. **<port>** : spécifie l'emplacement effectif du service.
  2. **<service>** : décrit un ensemble de points finaux du réseau.

- Les services peuvent être implémentés via différents styles & protocoles :

  ➢ **SOAP** ***(Simple Object Access Protocol - Protocol formel basé sur XML)***
  ➢ **REST** ***(Style architectural basé sur HTTP)***
  ➢ **RPC/gRPC** ***(Modèle d'appel distant)***
  ➢ **JMS, AMQP** ***(Protocole de messagerie)***

REST est un **style architectural** pour les **API web**, introduit par **Roy Fielding**, qui repose sur les **standards HTTP**.

- C'est une manière concrète de concevoir des **API**
- Très utilisé pour les services web légers, souvent en **JSON**
- Utilise les méthodes **HTTP** *(GET, POST, PUT, DELETE)*

Un **style architectural** est un ensemble coordonné de **contraintes architecturales** qui restreignent les rôles/caractéristiques des éléments architecturaux et les relations autorisées entre ces éléments au sein de toute architecture conforme à ce style.

	API REST	SOA(Service-Oriented Architecture)
Définition	Modèle architectural conçu pour les services Web basés sur **HTTP**	Approche architecturale plus large basée sur l'intégration de services autonomes.
Principes fondamentaux	Basé sur les principes du **REST** : ressources, verbes **HTTP**, stateless.	Centré sur des principes comme les services indépendants et les contrats explicites.
Technologies utilisées	Principalement **HTTP, JSON, XML**, ou autres formats légers.	Utilise souvent des protocoles comme **SOAP, WSDL, XML**.
Granularité	Souvent plus fine, expose des ressources spécifiques.	Souvent plus grossière, expose des services complexes.
Facilité d'intégration	Plus simple grâce aux standards du Web.	Peut nécessiter plus d'efforts à cause des outils et protocoles spécifiques.
Performance	Haute performance grâce à sa légèreté et son optimisation **HTTP**.	Peut être plus lent à cause des surcharges liées à **SOAP** et **XML**.
Cas d'usage principal	Applications Web, microservices, systèmes mobiles.	Intégration d'applications complexes, systèmes d'entreprise **(ERP, CRM)**.
Gestion de sécurité	Basée sur **HTTPS, OAuth, JWT**, etc.	Utilise des mécanismes comme **WS-Security** pour des scénarios complexes.

Travaux pratiques 1 : Mise en œuvre d'une API ReST flexible, scalable et résiliente.

On souhaite créer une application qui permet de gérer le catalogue des livres appartenant à des catégories différentes.

Chaque **Livre** est définit par :

- Son **titre** de type String
- Son **auteur** de type String
- Son **description** de type float
- Sa **disponibilité** de type boolean

Une **catégorie** est définie par :

- Son **code** de type Long (Auto Increment)
- Son **nom** de type String

L'application doit permettre :

- **D'ajouter une nouvelle catégorie**
- **Ajouter un livre appartenant à une catégorie**
- **Consulter toutes les catégories**
- **Consulter les livres d'une catégorie**
- **Consulter un livre**
- **Mettre à jour un livre**
- **Supprimer une catégorie**

Les données sont stockées dans une base de données **Postgresql/MySQL**.

L'application est un service Restful basé sur **Spring-Boot**.

II. Les bonnes pratiques

Conventions et bonnes pratiques

Conventions	Description
Conception RESTful (pour les API REST)	Utiliser les **Méthodes HTTP Appropriées** - Ressources nommées de manière claire : *Utiliser des noms de ressources (au pluriel)* clairs et descriptifs dans les URLs. Gestion des Réponses.
Standardisation	Formats de **Réponse Consistents** : Utiliser un format standard (comme JSON) pour toutes les réponses.
Sécurité	**Authentification et Autorisation** : Implémenter des mécanismes robustes comme **OAuth**, **JWT**. Validation et Sanitisation des Données : Toujours valider les données entrantes pour prévenir les attaques telles que l'injection **SQL**.
Documentation	**Documenter l'API** : Fournir une documentation complète et à jour, idéalement avec des exemples d'utilisation. Utiliser des Outils de Documentation : Comme **Swagger** ou **Redoc** pour générer une documentation interactive.
Gestion des Erreurs	**Messages d'Erreur Clairs** : Fournir des messages d'erreur descriptifs avec des codes d'erreur standardisés. **Documentation des Erreurs** : Documenter les types d'erreurs possibles et leur signification.
Performances et Scalabilité	**Pagination** : Pour les grandes collections de données, utiliser la pagination pour limiter la charge du serveur et améliorer la réactivité. **Mise en Cache** : Utiliser des mécanismes de mise en cache pour améliorer les performances.
Versioning	**Versionner l'API** : Prévoir des versions pour l'API pour gérer les changements sans perturber les utilisateurs existants. **Stratégie de Versioning** : Utiliser des URL ou des en-têtes pour gérer différentes versions.
Respect des Limites de Taux	**Limitation des Taux** : Mettre en place des

	limites pour prévenir l'abus et la surcharge des serveurs.
Test et Monitoring	**Tests Automatisés :** Écrire des tests pour chaque aspect de l'API. Surveillance de l'API : Utiliser des outils de monitoring pour surveiller les performances et la santé de l'API.

Stratégie de versionning

Approche 1 : La version dans l'URI

Ajouter un numéro de version dans **l'URI** est une pratique très répandue parmi les **APIs** publiques populaires.

Concrètement, il suffit d'insérer un « v1 » ou un simple chiffre dans l'URL, afin de faciliter le passage à la version suivante :

https://api.example.org/v1/places

Étant donné que cette approche est très répandue dans de nombreuses APIs publiques, c'est souvent **la première méthode que les développeurs adoptent** lorsqu'ils construisent leur propre API. C'est de loin **la plus simple**, et elle fonctionne.

Twitter, par exemple, utilise deux versions : **« /1/ » et « /1.1/ »**, qui étaient toutes deux actives au moment de la rédaction. Cela laisse aux développeurs le temps de mettre à jour leur code.

Approche 2 : Version dans le corps de la requête

Une autre approche consisterait à ajouter la version de l'API directement dans le corps de la requête :

```
POST /places HTTP/1.1
Host: api.example.org
Content-Type: application/json

{
  "version" : "1.0"
}
```

Cela permet de résoudre le problème des URLs qui changent avec le temps, mais cela peut entraîner une expérience utilisateur incohérente. Si le développeur de l'API envoie des données au format `JSON` ou une structure similaire, cela reste efficace. En revanche, si le `Content-Type` est `image/png` ou même `text/csv`, la gestion de la version devient très rapidement compliquée.

Autres approches :

Versioning REST Web Services · Peter Williams

III. La boîte à outil

Nous passons en revue quelques outils nécessaires au développement et/ou au test **d'API**.

1. Mock API

Un **mock API** server est un outil qui imite le comportement d'un véritable serveur d'API en fournissant des réponses réalistes aux requêtes.

Il peut fonctionner en local sur votre machine ou être accessible sur Internet.

Les réponses qu'il fournit peuvent être statiques (toujours les mêmes) ou dynamiques (adaptées à la requête), et simulent les données que l'API réelle retournerait, en respectant le schéma prévu (types de données, objets, tableaux, etc.).

https://mockapi.io/

2. OpenAPI et Swagger

La Spécification **OpenAPI** *(anciennement Spécification Swagger)* est un format de description **d'API** pour les **API REST**. Un fichier **OpenAPI** vous permet de décrire l'intégralité de votre API, notamment :

- Points de terminaison disponibles ***(/users)*** et opérations sur chaque point de terminaison ***(GET /users, POST /users)***
- Paramètres de fonctionnement Entrée et sortie pour chaque opération
- Méthodes d'authentification
- Coordonnées, licence, conditions d'utilisation et autres informations.

Les spécifications de l'API peuvent être écrites en **YAML** ou **JSON**. Le format est facile à apprendre et lisible aussi bien par les humains que par les machines.

Swagger est un ensemble d'outils open source construits autour de la spécification **OpenAPI** qui peuvent vous aider à concevoir, créer, documenter et utiliser des **API REST**.

https://editor.swagger.io/

La spécification définit plusieurs étiquettes qui vont permettre entre autres de :
définir les informations générales sur vos **API** : description, termes d'utilisation, licence, contact, etc. ;

- fournir la liste des services qui seront offerts, avec pour chacun, comment les appeler et la structure de la réponse qui est retournée ;
- définir le chemin pour consommer votre **API** ;
- etc.

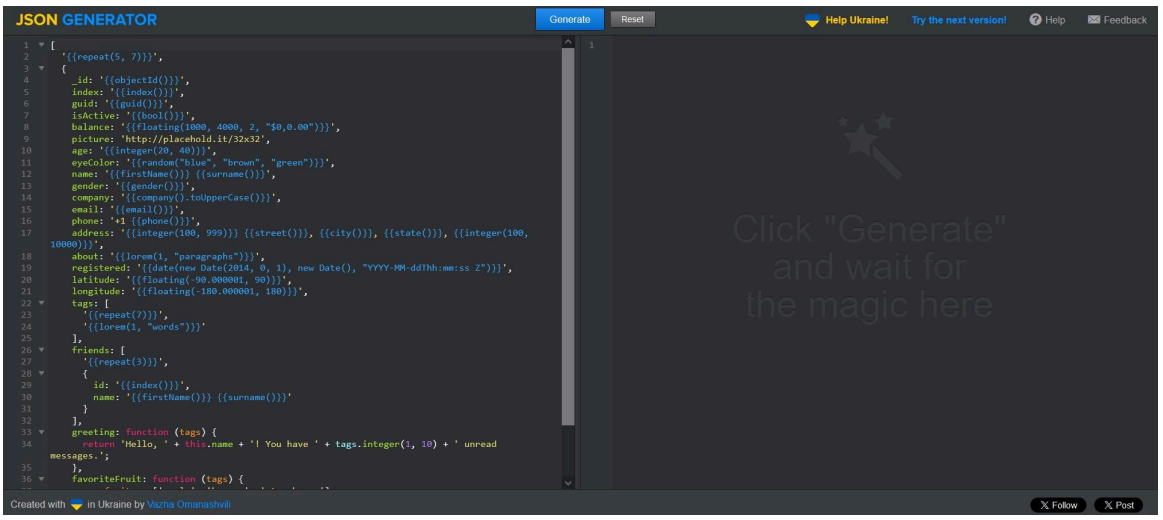
```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) or HTML.
  version: 0.1.9

servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing

paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

3. Postman & Insomnia

   [Découvrir et tester via Postman]

4. JSON generator et JSON server

https://json-generator.com/



Travaux pratiques 2 : Spécification d'une API ReST avec Swagger - Implémentation et test d'une API ReST

Atelier 1

Générer la documentation **Swagger** associée au projet de l'**atelier 1**.

Construisez un projet de test avec **postman**.

- **Définir les objectifs de test**
- **Créer une collection Postman**
- **Configurer l'environnement**
- **Définir les requêtes (requests)**
- **Ajouter des scripts de tests automatisés**
- **Structurer les tests avec des dossiers et des noms clairs**
- **Exécuter les tests manuellement ou automatiquement**
- **Analyser les résultats**
- **Documenter et partager**

Au regard des concepts étudiés précédemment, **l'API** développée dans l'atelier 1 observe t-elle les bonnes pratiques de conception/développement ?

Atelier 2

Voici la documentation d'une API **ReST.**

Créer un projet de test à l'aide de Postman.

Name : **Simple Books API**
Description : Cette API permet de réserver des livres.
url : https://simple-books-api.glitch.me

Endpoints

Status
GET /status : Returns the status of the API.

List of books
GET /books : Returns a list of books.

Optional query parameters:
type*: fiction or non-fiction*
limit*: a number between 1 and 20.*

Get a single book

GET /books/:bookId : Retrieve detailed information about a book.

Submit an order

POST /orders : Allows you to submit a new order. Requires authentication.

The request body needs to be in JSON format and include the following properties:
bookId - Integer - Required
customerName - String - Required

Example
POST /orders/
Authorization: Bearer <YOUR TOKEN>

```
{
"bookId": 1,
"customerName": "John"
}
```

The response body will contain the access token.

Get all orders

GET /orders : Allows you to view all orders. Requires authentication.

Get an order

GET /orders/:orderId : Allows you to view an existing order. Requires authentication.

Update an order

PATCH /orders/:orderId : Update an existing order. Requires authentication.

The request body needs to be in **JSON** format and allows you to update the following properties:

customerName - String

Example
PATCH /orders/PF6MflPDcuhWobZcgmJy5
Authorization: Bearer <YOUR TOKEN>

```
{
"customerName": "John"
}
```

Delete an order

DELETE /orders/:orderId : Delete an existing order. Requires authentication.
The request body needs to be empty.

Example
DELETE /orders/PF6MflPDcuhWobZcgmJy5

Authorization: Bearer <YOUR TOKEN>

API Authentication

To submit or view an order, you need to register your **API client**.

POST /api-clients/
The request body needs to be in **JSON** format and include the following properties:
clientName - String
clientEmail - String

Example
{
"clientName": "Valentin",
"clientEmail": "valentin@example.com"
}
The response body will contain the access token.

Possible errors

Status code 409 - "API client already registered." Try changing the values for **clientEmail** and **clientName** to something else.

IV. Rappels sur la sécurité

Toutes démarches de cybersécurité s'appuie sur les objectifs fondamentaux suivant :

- **Confidentialité/Autorisation** : Empêcher l'accès non autorisé à l'information.
- **Intégrité** : Garantir que les données ne soient pas altérées de façon illégitime.
- **Disponibilité** : S'assurer que les systèmes et données soient accessibles en temps voulu.
- **Authentification** : Vérifier l'identité des utilisateurs ou systèmes.
- **Traçabilité (Auditabilité)** : Conserver des preuves d'accès, de modifications ou de tentatives d'intrusion.

Typologies des menaces en cybersécurité

Farming et Throttling

- Le **farming** dans le contexte des API fait référence à une **utilisation excessive ou abusive** d'une API pour **collecter massivement des données** ou **tirer un avantage démesuré**, souvent via des scripts automatisés ou des bots.

Risques associés :

- **Surchage des serveurs.**
- **Violation des conditions d'utilisation.**
- **Perte de contrôle sur les données diffusées.**
- **Atteinte à la vie privée ou à la sécurité.**

- Le **throttling** est une technique utilisée pour **limiter le nombre de requêtes** qu'un client peut effectuer vers une API dans un laps de temps donné.

Types de throttling :

- ➢ **Rate limiting** : Ex. 100 requêtes par minute.
- ➢ **Quota-based limiting** : Ex. 10 000 requêtes par jour.
- ➢ **Concurrent limiting** : Ex. max 5 connexions simultanées.
- ➢ **Backoff automatique** : ralentissement progressif en cas d'usage excessif.

Exemples de réponses d'erreur liées au throttling :

- **HTTP 429** – Too Many Requests
- Message : "Rate limit exceeded. Retry after 60 seconds."

Présentation de l'OWASP Top 10

L'**OWASP Top 10** est un document de référence conçu pour sensibiliser les développeurs et les professionnels de la sécurité des applications Web aux risques de sécurité les plus critiques. Il est reconnu mondialement et est considéré comme une première étape importante vers un codage plus sécurisé.

La liste de l'**OWASP Top 10** est mise à jour régulièrement pour refléter les nouvelles tendances et menaces dans le domaine de la sécurité des applications Web.

Elle comprend des catégories telles que le contrôle d'accès défectueux, les échecs cryptographiques, et les injections, entre autres.

OWASP top 10

https://owasp.org/www-project-top-ten/

Découvrir le Pentesting

Pentesting *(abréviation de **Penetration Testing**)* est une méthode offensive et contrôlée de cybersécurité qui consiste à simuler une attaque réelle sur un système (application, réseau, API, etc.) dans le but de :

- ➢ **Identifier les failles de sécurité.**
- ➢ **Évaluer leur impact potentiel.**
- ➢ **Préconiser des mesures correctives avant qu'un attaquant réel ne les exploite.**

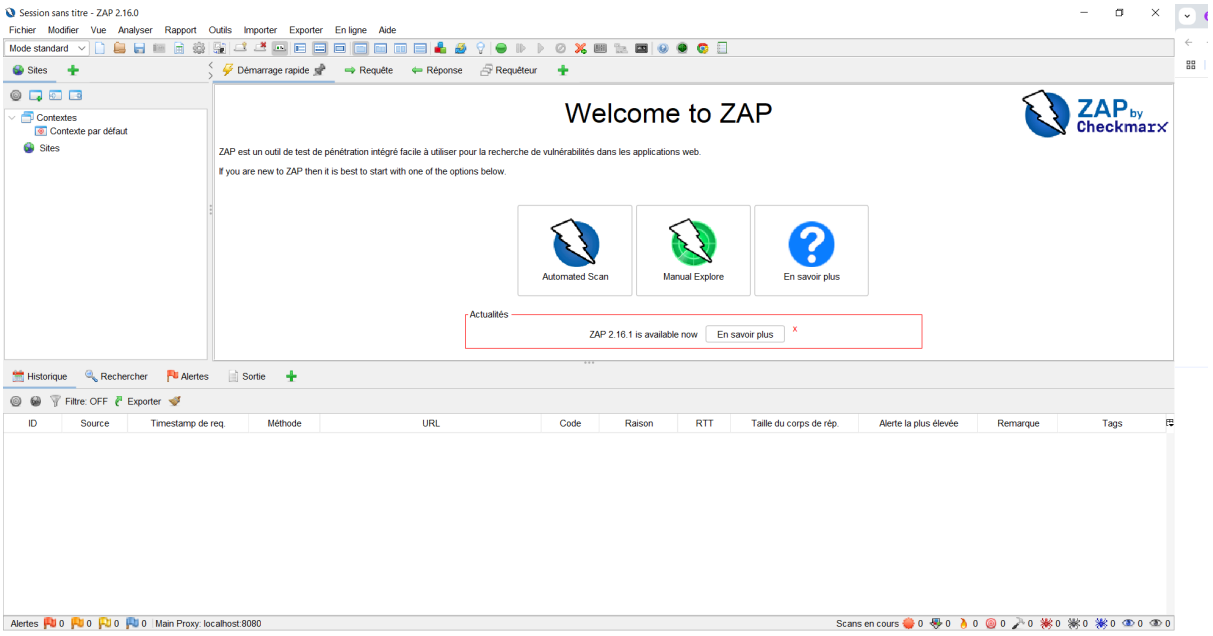
A quoi sert le pentesting ?

Le pentesting a pour objectif de révéler les vulnérabilités potentielles sur les applications. Il permet aussi de s'assurer de la conformité aux normes de sécurité **(OWASP - ISO 27001)**.

Les outils de pentesting

Il existe de nombreux outils de pentesting. Burp Suite - Postman - OWASP ZAP

Nous allons tester l'outil OWASP ZAP https://www.zaproxy.org/



V. Authentification & Autorisation

L'authentification est le processus qui permet de vérifier **l'identité** d'un utilisateur *(ex. : login/mot de passe, token, certificat, etc.)*.

L'autorisation est le processus qui détermine ce que l'utilisateur a le droit de faire une fois authentifié *(ex. : accéder à une ressource, modifier un contenu, etc.)*.

Ex : Un utilisateur peut être authentifié avec succès mais non autorisé à accéder à une ressource spécifique.

L'authentification permet aux API de suivre les utilisateurs, de fournir un contexte utilisateur aux points de terminaison *(« retrouver tous mes articles »)*, de limiter l'accès des utilisateurs à certains endpoints, de filtrer les données, ou même de limiter la fréquence d'utilisation et de désactiver des comptes.

Tout cela est très utile pour de nombreuses API, mais certaines n'auront peut-être jamais besoin de mettre en œuvre une authentification.

Différentes approches d'authentification

Méthode	Description	Avantages	Inconvénients	Cas d'utilisation
Basic Authentication	Authentification avec un nom d'utilisateur et un mot de passe encodé en Base64	Simple, facile à implémenter	Pas sécurisé (besoin de HTTPS), faible flexibilité	API internes ou situations peu sensibles
API Key	Utilisation d'une clé générée pour identifier et authentifier les requêtes	Facile à mettre en œuvre, pas besoin de mot de passe	Moins sécurisé si exposé, pas de gestion fine des permissions	API publiques, accès limité à certaines fonctionnalités
Bearer Token	Utilisation de jetons d'authentification (souvent dans un contexte OAuth 2.0)	Sécurisé, expiration des jetons, simple à utiliser	Risque de vol de jeton, gestion de l'expiration	Applications nécessitant des jetons temporaires
OAuth 2.0	Protocole d'autorisation permettant l'accès aux ressources d'un utilisateur via un jeton d'accès	Accès délégué sécurisé, permet une authentification sans mot de passe	Complexe à implémenter, gestion des jetons	Applications mobiles, intégrations avec services tiers

1. Sécurité de l'authentification.

La sécurité de l'authentification est un élément fondamental de toute application ou **API**, car elle permet de vérifier l'identité des utilisateurs avant de leur accorder l'accès à des ressources protégées.

Une mauvaise gestion de l'authentification peut entraîner des vulnérabilités importantes, telles que l'accès non autorisé aux données ou à des fonctions sensibles. Voici les bonnes pratiques essentielles pour garantir la sécurité de l'authentification.

Une authentification efficiente repose sur les bonnes pratiques suivantes :

- **Forcer le HTTPS pour toutes les communications :**

Pourquoi ?

Le protocole **HTTPS (HyperText Transfer Protocol Secure)** est essentiel pour garantir la confidentialité et l'intégrité des données transmises entre le client et le serveur.

L'utilisation de **HTTPS** chiffre toutes les communications, empêchant les attaquants d'intercepter ou de manipuler les données envoyées (par exemple, les informations d'identification).

- ➢ **Obliger HTTPS pour toutes les requêtes :** Toutes les communications sensibles, y compris les informations d'identification et les données utilisateur, doivent être envoyées sur une connexion HTTPS afin d'éviter toute fuite de données.

- ➢ **Certificats SSL/TLS :** Utilisez un certificat SSL/TLS valide et à jour. Des services comme Let's Encrypt offrent des certificats gratuits et automatisés.

- ➢ **Redirection automatique :** Configurez le serveur pour rediriger automatiquement toutes les requêtes HTTP vers HTTPS.

- **Utiliser des mécanismes de hachage robustes pour stocker les mots de passe (ex. : Argon2, bcrypt).**

Pourquoi ?

Le stockage des mots de passe en clair ou avec un mécanisme de hachage faible est l'une des erreurs de sécurité les plus courantes. Un attaquant ayant accès à la base de données des utilisateurs peut alors obtenir ces mots de passe et compromettre les comptes.

Le hachage des mots de passe avec des algorithmes de hachage robustes protège les mots de passe en les rendant irréversibles.

Bonnes pratiques :

- ➢ **Hachage avec un sel (salt) :** Ajoutez un sel unique pour chaque mot de passe avant de le hacher. Cela empêche les attaques par dictionnaire ou par table arc-en-ciel.
- ➢ **Utilisation d'algorithmes robustes :** Utilisez des algorithmes conçus pour être lents, tels que **bcrypt**, **scrypt** ou **Argon2**, afin de rendre les attaques par force brute plus difficiles.
- ➢ **Taille et coût :** Configurez le coût des algorithmes pour être suffisamment élevé, en fonction de la puissance de calcul disponible, afin de ralentir les attaques par force brute.

- **Implémenter une limitation de tentatives (rate limiting / throttling).**

Pourquoi ?

La limitation du nombre de tentatives est une technique efficace pour protéger vos **API** et systèmes d'authentification contre les attaques par force brute *(comme deviner un mot de passe ou un jeton d'authentification en essayant de multiples combinaisons)*.

Bonnes pratiques :

- ➢ **Limiter le nombre de tentatives par IP :** Après un certain nombre d'échecs, bloquez l'IP de l'utilisateur pendant une période déterminée.
- ➢ **Limiter les tentatives de connexion par compte utilisateur :** Après un certain nombre d'échecs, imposez un délai avant de permettre une nouvelle tentative.
- ➢ **Utiliser des outils de rate limiting** : De nombreux serveurs web et API proposent des outils intégrés pour gérer la limitation des requêtes, tels que **NGINX**, **API Gateway**, ou des bibliothèques comme **express-rate-limit** pour Node.js.

- **Générer et stocker les tokens d'authentification (JWT, OAuth) de manière sécurisée.**

2. Système de logging.

Les **systèmes de logging** permettent d'assurer la traçabilité des événements qui surviennent tout au long du fonctionnement des applications *(actions diverses, détection d'intrusions,audit de sécurité)*.

Un bon système de logging observe les recommandations suivantes :

- Journaliser les **tentatives de connexion**, échecs inclus.
- Ne jamais logguer les **données sensibles** (mots de passe, tokens, numéros de carte...).
- Intégrer les logs dans un système centralisé (ELK, Datadog, etc.).
- Mettre en place des **alertes** sur les comportements suspects.

3. Sécurité côté serveur.

La sécurité côté serveur est tributaire d'un certain nombre de bonnes pratiques. Elle repose sur des principes essentiels :

- **La validation et la sanitisation des données**
- **Application du principe du moindre privilège (attribution de droits stricts)**
- **Mise à jour régulière des dépendances**
- **Gestion stratégique du logging**

4. CORS (Cross-Origin Resource Sharing) et CSRF (Cross-Site Request Forgery).

Le partage des ressources entre origines multiples (**CORS**) est un mécanisme d'intégration des applications. Il consiste à ajouter des **en-têtes HTTP** afin de permettre à un agent utilisateur d'accéder à des ressources d'un serveur situé sur une autre origine que le site courant.

La spécification **CORS** permet aux applications **Web** clientes chargées dans un domaine particulier d'interagir avec les ressources d'un autre domaine. Cela est utile, car les applications complexes font souvent référence à des **API** et à des ressources tierces dans leur code côté client.

CSRF est une attaque dans laquelle un site malveillant force un utilisateur connecté à un autre site à exécuter une action à son insu *(ex : transfert d'argent, changement de mot de passe)*.

5. Canonicalization, Escaping et Sanitization.

Concept	Objectif	Description	Exemple	Risques
Canonicalization	Unifier les représentations équivalentes	Convertir des données sous des formes différentes en une forme standardisée	Normaliser un chemin de fichier (/home/user/file.txt vs /home/user/../user/file.txt)	Accès non autorisé à des fichiers sensibles ou contrôle d'accès incorrect
Escaping	Neutraliser les caractères spéciaux	Remplacer les caractères spéciaux par des versions sûres pour éviter leur interprétation	Transformer <script> en <script> dans du HTML	Exécution de code malveillant (ex : attaques XSS)
Sanitization	Filtrer ou nettoyer les entrées non sûres	Supprimer ou modifier les entrées pour éviter les attaques comme l'injection SQL, XSS, etc.	Filtrer des balises HTML dans un champ de commentaire (<script> devient une chaîne vide)	Injections malveillantes, stockage de données non sûres

6. Gestion des permissions : Role-Based Acces vs. Resource-based access.

Role-Based Access Control (RBAC)

Dans le contrôle d'accès basé sur les rôles **(RBAC),** les permissions sont attribuées en fonction du rôle d'un utilisateur dans l'organisation. Un utilisateur obtient un ou plusieurs rôles, et chaque rôle est associé à des permissions spécifiques.

Caractéristiques de RBAC :

- **Rôles** : Chaque utilisateur est associé à un ou plusieurs rôles (ex : administrateur, utilisateur, modérateur, etc.).
- **Permissions associées aux rôles** : Les rôles définissent les actions qu'un utilisateur peut exécuter. Par exemple, un rôle d'administrateur peut avoir des permissions pour créer, modifier et supprimer des ressources, tandis qu'un utilisateur standard peut seulement lire des données.
- **Hiérarchie des rôles** : Les rôles peuvent être hiérarchiques, ce qui permet à un rôle plus élevé d'hériter des permissions d'un rôle inférieur.
- **Facilité de gestion** : Si un utilisateur change de rôle, il suffit de mettre à jour son rôle pour modifier ses permissions.

Exemple :

- **Rôle :** Administrateur → Permissions : Créer, Modifier, Supprimer
- **Rôle :** Utilisateur → Permissions : Lire

Resource-Based Access Control (RBAC)

Objectif principal :
Le contrôle d'accès basé sur les ressources (Resource-Based Access Control) est centré sur les ressources elles-mêmes. Plutôt que d'attribuer des permissions en fonction des rôles des utilisateurs, les permissions sont spécifiées pour chaque ressource individuelle.

Caractéristiques de Resource-Based Access Control :

- **Ressources** : Chaque ressource (fichier, donnée, document, API, etc.) a un ensemble de permissions définies qui peuvent être attribuées aux utilisateurs.

- **Permissions liées aux ressources** : Les utilisateurs ont un accès explicite aux ressources en fonction des permissions définies pour chaque ressource.

- **Granularité fine** : Chaque ressource peut avoir un contrôle d'accès indépendant, ce qui permet un contrôle d'accès très détaillé au niveau de chaque élément ou ressource.

- **Peut être combiné avec RBAC** : Le contrôle basé sur les ressources peut être utilisé en combinaison avec RBAC, en ajoutant une couche supplémentaire de gestion des permissions.

Exemple :

7. **Ressource :** Document A → Permissions : Lecture, Écriture (Accordées à l'utilisateur 1)

8. **Ressource :** Document B → Permissions : Lecture (Accordées à l'utilisateur 2)

VI. Le middleware JWT (Json Web Token).

1. Rappels sur la cryptographie

La **cryptographie** est la science qui étudie les techniques de **chiffrement** et de **protection des informations**, afin d'assurer la **confidentialité**, l'**intégrité**, l'**authenticité** et la **non-répudiation** des données, que ce soit lors de leur transmission ou de leur stockage.

En d'autres termes, la cryptographie permet de transformer des données lisibles (texte en clair) en données illisibles (texte chiffré), sauf pour les personnes autorisées possédant les clés nécessaires pour les déchiffrer.

On distingue deux approches de cryptographie :

- **Cryptographie symétrique** (nécessitant une seule clé ou chiffrer et déchiffrer les données).
  Exemples: ***AES (Advanced Encryption Standard)****.*
- **Cryptographie asymétriques** (Utilise une **clé publique** pour chiffrer, une **clé privée** pour déchiffrer.)
  Exemples : ***RSA****,* ***ECC****.*

A la notion de cryptographie, on associe souvent la notion de **Hachage.**

La notion de Hachage désigne une fonction qui transforme des données en une empreinte unique *(ex : SHA-256)*. Le hachage est irréversible c'est-à-dire qu'on ne peut retrouver l'entrée d'origine à partir du hash.

L'utilisation conjuguée d'une fonction de hachage et de la cryptographie asymétrique permet de garantir que la données provient bien de l'expéditeur et qu'elle n'a pas été modifiée.

De nos jours, l'un des outils de cryptographie les plus répandus pour l'authentification des API est le **JWT (JSON Web Token)**.

2. Les grands principes de JWT

Les **JSON Web Tokens (JWT)** reposent sur plusieurs **grands principes fondamentaux** qui permettent d'assurer une transmission sécurisée et structurée des données entre deux parties, notamment en contexte d'authentification et d'autorisation.

Structure en 3 parties :

Un JWT est composé de **trois parties** encodées en base64 et séparées par des points :

[header].[payload].[signature]

- **Header** : contient le type de token (JWT) et l'algorithme de signature *(ex. HS256, RS256)*.
- **Payload** : contient les données (ou *claims*) que l'on souhaite transmettre (ex. user_id, role, exp).
- **Signature** : permet de vérifier l'intégrité du token et son authenticité, en fonction du header et du payload signés avec une clé secrète ou une clé privée.

Le JWT est souvent utilisé pour transmettre des **informations d'identité** ou de **droits d'accès** entre un client (navigateur, app mobile) et un serveur. Il peut être utilisé dans le header d'une requête HTTP (Authorization: Bearer <token>).

Un JWT est **auto-contenu** : il contient toutes les informations nécessaires, ce qui évite de stocker une session côté serveur. Cela rend l'application **scalable** (utile dans des architectures sans état comme RESTful APIs).

3. Risques et vulnérabilités intrinsèques

Risque	Problème	Conséquence	Bonne pratique
Absence de chiffrement	Les données dans le JWT (notamment le payload) sont encodées en base64, mais pas chiffrées.	Toute personne ayant accès au token peut lire son contenu, y compris les informations sensibles si elles y sont stockées.	Ne jamais inclure de données confidentielles dans le payload.
Falsification de signature (alg="none")	Certains serveurs mal configurés acceptent des tokens avec **alg: none**, ce qui désactive la vérification de signature.	Un attaquant peut fabriquer un faux token et l'envoyer comme valide.	Toujours désactiver le support de **alg=none** et forcer un algorithme signé comme HS256 ou RS256.
Mauvaise gestion des clés	Utilisation d'une clé secrète faible ou exposition de la clé (dans le code ou dépôt public).	Permet à un attaquant de signer ses propres tokens.	Utiliser des clés suffisamment longues et stockées dans un coffre sécurisé (ex. : Vault,

			AWS Secrets Manager).
Absence ou mauvaise gestion de l'expiration (exp)	Si le champ exp est omis ou mal géré, le token peut rester valide indéfiniment.	Si un token est volé, il peut être utilisé pour toujours.	Toujours inclure une date d'expiration courte, avec renouvellement automatique via un refresh token.

Travaux pratiques **(Corrigé cf github)** :

- Sécurisation de l'API

Récapitulatif RoadMap sécurité des API

VII. Les tests d'API

Les tests d'API (Application Programming Interface) sont une partie essentielle du développement de logiciels modernes. Ils visent à s'assurer que les interfaces entre différentes applications logicielles fonctionnent correctement.

Postman est l'un des nombreux outils qui permettent de tester les API. Les tests réalisés sur Postman visent à s'assurer du bon fonctionnement, de la sécurité et de la performance des services web. Ces tests peuvent se répartir comme suit :

Test fonctionnels

Ces tests vérifient que l'API fonctionne comme attendu.

- Vérification des codes de statut HTTP (200, 404, 500, etc.)
- Validation du contenu de la réponse (structure JSON, champs spécifiques)
- Tests sur les en-têtes de réponse
- Contrôle des valeurs retournées (valeurs exactes, types, formats)

Test de performance (basiques)

Postman permet de tester si la réponse est rapide (ex : moins de 1 seconde).

Test de sécurité

Tests de validation de schéma (schema validation) :

Vérifie que le corps de la réponse respecte un schéma JSON prédéfini (structure attendue) .

Collections & Variables dans Postman :

Organiser les requêtes dans les collections :

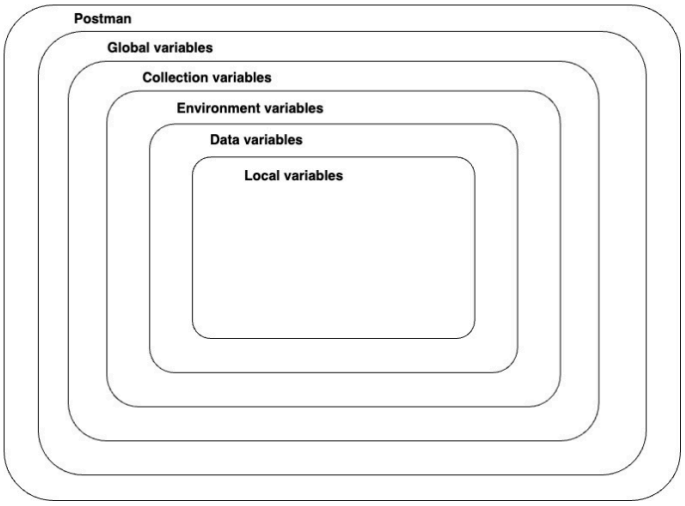
Dans Postman, une collection est un ensemble portable de requêtes pouvant être utilisées et automatisées. Les collections permettent l'enregistrement d' informations importantes de chaque requête API (le type d'autorisation - les headers - le body - les scripts et variables - la documentation).

Création d'une collection (Postman DOC)

Stocker les valeurs dans les variables :

Les variables permettent de stocker et de réutiliser les valeurs dans Postman.Les variables vous aident à travailler efficacement, à collaborer avec vos collègues et à mettre en place des workflows dynamiques.

La portée des variables sur Postman :



[comprendre les variables (Postman DOC)]

Génération de données random:

Postman permet de générer des données aléatoires **(random)** grâce à sa bibliothèque intégrée ***faker.js*** (via pm.variables.replaceIn() ou directement via les fonctions **{{$variable}}** dans les champs de requête).

Quelques exemples :

Variable Postman	Description
{{$randomFirstName}}	Prénom aléatoire
{{$randomLastName}}	Nom de famille aléatoire
{{$randomEmail}}	Email aléatoire
{{$randomInt}}	Nombre entier aléatoire
{{$randomPhoneNumber}}	Numéro de téléphone aléatoire
{{$randomUUID}}	UUID aléatoire
{{$randomAddress}}	Adresse aléatoire

Ci-dessous un exemple de corps de requête en JSON :

```
{
  "name": "{{$randomFirstName}} {{$randomLastName}}",
  "email":  "{{$randomEmail}}",
  "age": {{$randomInt}},
  "phone": "{{$randomPhoneNumber}}"
}
```

[Données aléatoirement générées]

VIII. API Management

Une solution d'**API Management**, est un ensemble d'outils et de services utilisés pour créer, gérer, sécuriser, et analyser les **API (Application Programming Interfaces)**.

Ces solutions sont conçues pour aider les organisations à gérer efficacement leurs interfaces de programmation, surtout lorsque le nombre **d'API** augmente dans le cadre de la transformation numérique et de l'intégration de divers systèmes et services.

Les fonctionnalités des API Managers

Portail pour Développeurs : Offre une interface où les développeurs peuvent s'inscrire, découvrir, et utiliser les API disponibles.

Sécurité et Authentification : Intègre des mécanismes de sécurité comme **OAuth, JWT**, pour contrôler l'accès et protéger les **API**.

Gestion du Trafic : Permet de gérer le trafic des API, incluant le throttling (limitation) et la mise en cache pour optimiser les performances.

Analyse et Reporting : Fournit des outils pour surveiller l'utilisation des API, analyser les performances, et identifier les problèmes.

Gestion du Cycle de Vie des API : Supporte le développement et la maintenance des API à travers leur cycle de vie complet.

Transformation et orchestration : Permet la transformation de requêtes et réponses, ainsi que l'orchestration des appels API.

Gravitee API Manager

Gravitee est une plateforme open source qui offre des fonctionnalités **d'API Management.**

Elle permet aux organisations de gérer efficacement leurs **APIs** en fournissant un ensemble d'outils et de fonctionnalités.

Prérequis pour la mise en place de l'APIm Gravitee avec Docker

- **Docker**
- **Un éditeur de code**

https://docs.gravitee.io/apim/3.x/apim_installation_guide_docker_compose_quickstart.html#installing_apim

Première connexion :

- login : admin
- MDP : admin

Travaux pratiques **(Corrigé cf github)** :

- Utiliser Gravitee pour créer une AP