AISI · ARCHITECTURE ET INTÉGRATION DES SI · QCM

Vendredi 27 novembre, 9h05. Quatre thèmes, une seule logique.

L'examen est un **QCM**. Le cours entier raconte une seule histoire : on arrête de brancher les applications n'importe comment (urbanisation), on les fait parler par un **contrat** (API REST), on code ce contrat en Python (**FastAPI**), puis on l'emballe pour qu'il tourne partout (**Docker**).

- 1. Urbanisation des SI & architectures
- 2. Principes des API REST
- 3. FastAPI sur GenevaLogistics
- 4. Conteneurisation Docker

Source des 4 thèmes : mail de rappel de vos enseignants (lun. 23 nov.). Contenu tiré des 3 supports de cours et des 2 TP du Classroom AISI.

VOTRE PLAN JUSQU'À VENDREDI

Une heure par soir suffit pour viser 4,5 à 5,5

Lundi 23

Urbanisation & architectures

- Pages 3 à 6
- Retenir : 4 visions, Zone › Quartier › Îlot, 3 règles d'or, 1/2/3-Tiers

Mardi 24

API REST

- Pages 7 à 9
- Retenir : verbes, idempotence, codes 200/201/204/401/404/422

Mercredi 25

FastAPI + GenevaLogistics

- Pages 10 à 12
- Le plus « rentable » : le TP avait déjà son propre QCM de 13 questions

Jeudi 26

Docker + QCM blanc

- Pages 13 et 14
- Puis le QCM d'entraînement (fichier `qcm-entrainement-AISI.html`) jusqu'à 80 %

Ce qui tombera très probablement : les codes HTTP, « quel fichier fait quoi » dans GenevaLogistics, la différence VM / conteneur, Dockerfile › image › conteneur, et les 3 règles d'or de l'urbanisation. C'est mon estimation à partir du QCM du TP, pas une information de vos enseignants.

Brancher chaque application à chaque autre finit en « plat de spaghettis »

Le piège du point-à-point

Nombre de liaisons possibles = **n × (n − 1) / 2**.

5 applications → 10 liaisons. 10 applications → 45. Ces liens cachés (accès direct aux tables, triggers SQL, fichiers partagés) sont des « contrats implicites » qui cassent en cascade.

Le TCO (coût total de possession)

Comme un iceberg :

20 à 30 % visibles (achat, licences, développement).
70 à 80 % cachés (maintenance, ressaisies, contournements Excel, dette technique).

La métaphore de la ville

On ne rase jamais une ville pour la reconstruire. L'urbaniste fait cohabiter les vieux bâtiments (le **legacy** = anciens systèmes encore vitaux) avec du neuf (Web, mobile, cloud), sans couper les services.

Les 4 visions du SI (du haut vers le bas)

VISION	QUESTION	EXEMPLE
1. Métier	Pourquoi ? Quoi ?	Les processus : « de la commande client à l'encaissement »
2. Fonctionnelle	Quelles fonctions ?	Zones, quartiers, îlots — **sans aucune technologie**
3. Applicative	Quels logiciels ?	ERP, CRM, API REST, SGBD
4. Infrastructure	Où ?	Serveurs, machines virtuelles, réseau, conteneurs

Le plan d'occupation des sols : des boîtes dans des boîtes, et 3 règles d'or

POS fonctionnel = ce que FAIT l'entreprise

ZONE — grande finalité (Opérationnelle, Décisionnelle, Référentiels communs, Échanges externes)

QUARTIER — domaine métier homogène (Gestion commerciale, Logistique & stocks…)

Îlot : Prise de commande | Îlot : Gestion des devis

Îlot = la plus petite unité, autonome (ex. Émission des factures).

Piège QCM : un POS ne contient **jamais** de technologie (pas de serveur, pas d'Excel, pas de SQL). Sinon c'est une vue applicative ou technique.

RÈGLE D'OR 1

Couplage faible, forte cohésion

Chaque bloc est autonome à l'intérieur et dépend le moins possible du fonctionnement interne des autres.

RÈGLE D'OR 2

Source unique (référentiels)

Pas de doublons : un seul référentiel Clients, un seul catalogue Articles (« Single Source of Truth », données maîtres), consulté par tous.

RÈGLE D'OR 3

Échanges par contrat stable

Les applications se parlent par des contrats versionnés (API REST / OpenAPI, événements), **jamais** en attaquant la base du voisin. But : tout changement reste local, observable et réversible.

Toute application a 3 couches ; les architectures diffèrent par leur répartition

[P] Présentation (l'écran, la saisie) · **[M] Métier** (règles, calculs) · **[D] Données** (stockage durable)

TYPE	OÙ SONT P, M, D ?	PLUS	MOINS
1-Tier monolithe, silo	Tout au même endroit (exécutable local, fichier Excel/Access, mainframe)	Simple, aucune latence réseau	Silo de données, pas de travail à plusieurs, pas d'API, ressaisies
2-Tiers client / serveur	Client lourd (P + un peu de M) ⇄ serveur SGBD (D + M en procédures stockées)	Données partagées sur le réseau local	Mise à jour sur N postes, connexion SQL directe risquée, dépendance au SGBD (« vendor lock-in »)
3-Tiers API-first	Client léger Web/mobile (P) › serveur d'application / API REST (M) › SGBD privé (D)	Sécurité par identité (OAuth), contrat OpenAPI, rien à installer	Il faut bien concevoir les API et gérer les appels réseau

Excel / Access = 1-Tier

Feuilles et formulaires = [P], formules et macros VBA = [M], données dans le fichier .xlsx/.accdb = [D]. Ces fichiers créés par les métiers sans l'informatique, c'est le **Shadow IT**.

Migrer un 2-Tiers vers un 3-Tiers, en 3 étapes

1. Mettre une API (passerelle) devant la base : login individuel (OAuth/token), quotas, contrat OpenAPI.
2. Brancher les nouveaux canaux (portail web) sur cette API uniquement.
3. Sortir petit à petit la logique des procédures SQL vers le serveur d'application.

Monolithe modulaire, SOA ou microservices : on choisit le plus simple qui suffit

CRITÈRE	MONOLITHE 3-TIERS MODULAIRE	SOA AVEC ESB	MICROSERVICES
Taille des blocs	Un bloc découpé en modules métier + façade API REST	Gros services d'entreprise	Petits services, un par sous-domaine
Communication	Appels internes + API REST (API Gateway)	Bus central « intelligent » (**ESB**) qui route et traduit	Tuyaux légers : REST ou événements asynchrones
Données	Une base unique	Bases souvent partagées	**Une base privée par service**
Déploiement	Simple, en un bloc	Coordonné, gouvernance centrale	Indépendant par conteneur (CI/CD avancé)
Pour qui ?	PME, 2 à 8 développeurs	Grandes entreprises, nombreux progiciels	Cloud à très forte charge, grandes équipes

Les deux dangers à connaître

ESB : si on y met de la logique métier, il devient un goulet d'étranglement et un **SPOF** (point unique de défaillance : s'il tombe, tout tombe).
Microservices : la « taxe microservices » — synchroniser les données, tracer le réseau, déployer 10 services demande une grosse équipe.

Règle d'arbitrage (à savoir mot pour mot) : privilégier toujours l'architecture la plus simple qui respecte le niveau de service attendu (SLO), la trajectoire du legacy et la capacité réelle de l'équipe IT.

Une API REST est un contrat : on demande, on ne touche pas à la base du voisin

Définition

Interface qui permet à des applications de communiquer et de manipuler des **ressources** via HTTP, ses méthodes et ses codes, en général en **JSON**.
REST = REpresentational State Transfer, un **style d'architecture** défini par **Roy Fielding**.

Pourquoi une API dans le SI ?

Interopérabilité (ERP, CRM, mobile se parlent)
Abstraction (la base est cachée)
Réutilisabilité (un backend, plusieurs clients)
Sécurité (un point de contrôle unique)

Les contraintes REST

Séparation client–serveur · **sans état** (stateless) · cache possible · interface uniforme · organisation en couches.

« Sans état » (stateless), la notion la plus testée

Chaque requête contient **tout** ce qu'il faut (jeton + identifiant). Le serveur ne garde aucune session entre deux appels. Avantages : résistance aux pannes, ajout facile de serveurs (scalabilité horizontale), mise en cache.

STYLE	FORMAT	USAGE
REST	JSON / XML	Le standard du Web, orienté ressources (CRUD)
GraphQL	JSON, un seul point d'entrée	Le client choisit les champs (évite d'en recevoir trop ou trop peu)
gRPC	Binaire (Protocol Buffers, HTTP/2)	Microservices, très haute performance

Piège QCM : REST est un **style**, FastAPI est un **framework Python** qui sert à le coder, OpenAPI est le **format qui documente** le contrat. REST seul ne garantit ni la sécurité ni la fiabilité : il faut les concevoir.

Des noms dans l'adresse, l'action dans le verbe

✓ `GET /articles/123` · `POST /clients`

✗ `/get-article` · `/create-user`

OPÉRATION (CRUD)	VERBE	ADRESSE (URI, AU PLURIEL)	CORPS JSON	IDEMPOTENT ?	CODE SI RÉUSSI
Lire la liste / un élément	**GET**	`/clients` · `/clients/42`	Aucun	OUI	200
Créer	**POST**	`/clients`	Les données	NON	201 Created
Remplacer entièrement	**PUT**	`/clients/42`	Objet complet	OUI	200
Modifier partiellement	**PATCH**	`/clients/42`	Seulement les champs changés	PAS GARANTI	200
Supprimer	**DELETE**	`/clients/42`	Aucun	OUI	204 No Content

Idempotent = rejouer la même requête donne le même résultat. Seul **POST** crée une nouvelle ressource à chaque appel.

Anatomie d'un échange

1. Verbe + URI (l'action, la cible) · 2. En-têtes (format `Content-Type: application/json`, authentification) · 3. Corps JSON + code de statut.

2xx tout va bien, 4xx le client s'est trompé, 5xx le serveur a planté

2XX · SUCCÈS

200 OK — lecture ou modification réussie

201 Created — ressource créée (POST)

204 No Content — supprimé, **aucun corps** renvoyé

4XX · ERREUR DU CLIENT

400 Bad Request — requête mal formée

401 Unauthorized — pas authentifié / mauvaise clé

403 Forbidden — authentifié mais **pas le droit**

404 Not Found — identifiant introuvable

422 Unprocessable — données invalides (FastAPI/Pydantic)

5XX · ERREUR SERVEUR

500 Internal Server Error — bug inattendu côté serveur

401 ou 403 ?
401 = « je ne sais pas qui vous êtes ».
403 = « je sais qui vous êtes, mais c'est interdit ».

FastAPI = un moteur web (Starlette) + un contrôleur de données (Pydantic)

```python
from fastapi import FastAPI, Depends
app = FastAPI()

@app.get("/articles/{article_id}")   # décorateur = verbe + chemin
def lire_article(article_id: int,      # paramètre de CHEMIN (dans l'URL)
                 devise: str = "CHF"):  # paramètre de REQUÊTE ?devise=
    return {"id": article_id}            # dict → JSON automatiquement

class Article(BaseModel):                # modèle Pydantic = contrat
    nom: str
    prix: float                           # "abc" → 422 automatique

@app.delete("/items/{item_id}")
def delete_item(item_id: int,
    api_key = Depends(verify_api_key)):  # exécuté AVANT la fonction
    ...
```

Starlette · ASGI

Le moteur web : routage, requêtes asynchrones (`async/await`). D'où la rapidité.

Pydantic · validation

Utilise les annotations de type Python. Donnée invalide → **422** sans exécuter une seule ligne de votre fonction.

Depends() · injection de dépendances

Partager une logique (clé d'API, connexion base, pagination) sans la recopier. Évaluée avant la fonction.

Règle path / query

Tout argument qui n'est pas entre `{ }` dans le chemin devient un paramètre de requête (`?search=…`).

`uvicorn main:app --reload` lance le serveur (redémarre à chaque sauvegarde) · `/docs` Swagger UI générée toute seule · `/openapi.json` le contrat

Un fichier = une responsabilité. Savoir « qui fait quoi » rapporte des points

FICHIER	RÔLE	CE QU'IL CONTIENT (À RECONNAÎTRE DANS UNE QUESTION)
`app/main.py`	Point d'entrée, assemblage	`FastAPI()`, middleware **CORS**, `app.include_router(inventory.router)`, route racine `/` (healthcheck)
`app/schemas.py`	Validation (Pydantic v2)	`StockItemBase`, `StockItemCreate`, `StockItemUpdate`, `StockItemResponse`. Règles : **SKU 3 à 20 caractères**, **prix > 0** (`gt=0`), **quantité ≥ 0**
`app/dependencies.py`	Sécurité	`verify_api_key()` lit l'en-tête **X-API-Key** (`Header(...)`), compare à la clé du TP, lève **401** si faux
`app/database.py`	Persistance (Repository)	`InMemoryStockRepository` (mémoire, protégé par `threading.Lock()`), `_seed_data()` = 5 articles de départ, calcule **is_available = quantity > 0**
`app/routers/inventory.py`	Contrôleur REST	`APIRouter(prefix='/inventory', tags=['Stock'])` : GET paginé, GET /{id}, POST, PATCH /{id}, DELETE /{id}
`tests/test_inventory.py`	Tests	**13 tests** automatisés (Pytest + TestClient)

Le « préfixage »

Grâce à `prefix='/inventory'`, les routes du routeur s'écrivent `"/"` et `"/{item_id}"` ; FastAPI ajoute `/inventory` devant.

Qui est protégé ?

Les opérations qui **modifient** (POST, PATCH, DELETE) exigent la clé. La lecture (GET) est publique.

Les 6 manipulations du TP et le code que l'on obtient

SITUATION	CODE	POURQUOI
`GET /inventory` (page=1, limit=10 par défaut ; filtres `search`, `category`)	**200**	Lecture réussie, 5 articles au départ
`GET /inventory/9999`	**404**	« Article #9999 introuvable dans le stock »
`POST /inventory` avec une **mauvaise** clé	**401**	Bloqué par `verify_api_key`
`POST /inventory` **sans** en-tête X-API-Key	**422**	L'en-tête est obligatoire (`Header(...)`) : la validation échoue avant même la vérification
`POST /inventory` correct	**201**	id attribué automatiquement (6), is_available = true car quantité 18 > 0
`POST` avec sku = "X" ou prix = −50	**422**	Pydantic rejette avant le contrôleur et la base
`PATCH /inventory/1` avec `{"quantity": 0}`	**200**	Les autres champs restent intacts grâce à `exclude_unset=True` ; is_available passe à false
`DELETE /inventory/6`, puis `GET /inventory/6`	**204** puis **404**	Supprimé sans corps de réponse, puis introuvable

Lancer le projet sous Windows (sans Docker)

```
python -m venv env
.\env\Scripts\Activate.ps1
pip install -r requirements.txt
uvicorn app.main:app --reload --port 8000
python -m pytest tests/test_inventory.py -v
```

À quoi ça sert ?

venv isole les bibliothèques du projet du reste de la machine. **Uvicorn** est le serveur qui exécute l'API ; `--reload` redémarre à chaque sauvegarde. Si PowerShell bloque : `Set-ExecutionPolicy RemoteSigned -Scope Process`.

Un conteneur = l'application + ses dépendances, qui partage le noyau de la machine

Machine virtuelle

Matériel › **hyperviseur** › un **OS invité complet** par VM › librairies › appli. Lourd : plusieurs Go, RAM réservée en permanence.

Conteneur

Matériel › OS hôte (**noyau Linux partagé**) › Docker Engine › librairies › appli. Léger : un processus isolé, quelques dizaines de Mo. Ordres de grandeur du cours : ~90× plus rapide à démarrer, ~50× plus léger sur disque.

DOCKERFILE

La **recette** écrite

→ `build`

IMAGE

Le **modèle figé**, en lecture seule (comme une **classe** en POO), stocké sur un registre (Docker Hub)

→ `run`

CONTENEUR

L'**instance** qui tourne (comme un **objet**) ; 1 image → N conteneurs

```
FROM python:3.11-slim          # image de départ officielle
WORKDIR /app                    # dossier de travail
COPY requirements.txt .         # d'abord les dépendances…
RUN pip install -r requirements.txt  # exécuté au BUILD
COPY . .                       # …puis le code
EXPOSE 8000                    # port d'écoute
CMD ["python", "serveur.py"]    # exécuté au DÉMARRAGE
```

COMMANDE	RÔLE
`docker build -t api:1.0 .`	Construire l'image
`docker pull postgres:16`	Télécharger une image
`docker run -p 8000:8000 api:1.0`	Lancer un conteneur (port hôte:conteneur)
`docker ps` / `ps -a`	Conteneurs actifs / aussi les arrêtés
`docker stop`	Arrêter proprement

Trois conteneurs démarrés d'une seule commande : `docker compose up -d`

SERVICE	IMAGE	PORT HÔTE → CONTENEUR	STOCKAGE
db	postgres:16-alpine	5432 → 5432	Volume nommé `postgres_data`
api	construite localement (python:3.12-slim)	8000 → 8000	Bind mounts `./app:/app/app`
pgadmin	dpage/pgadmin4	5050 → 80	Volume nommé `pgadmin_data`

Pourquoi « db » et pas « localhost » ?

Dans un conteneur, localhost = le conteneur lui-même. Compose crée un réseau où le **nom du service** sert d'adresse (DNS interne).

healthcheck + service_healthy

`pg_isready` teste PostgreSQL. Sans `condition: service_healthy`, l'API démarrerait avant que la base soit prête → erreur de connexion.

Bind mount ou volume ?

Bind mount = un dossier de votre PC vu dans le conteneur (le code, pour le rechargement à chaud). **Volume nommé** = géré par Docker, les données **survivent** à la suppression du conteneur.

Utilisateur appuser (UID 10001)

Ne pas tourner en **root** : principe du moindre privilège. Si l'appli est piratée, l'attaquant a peu de droits.

Ce qui a changé dans le code

Seulement `database.py` (SQLAlchemy + PostgreSQL). Routeur et schémas intacts : c'est le **couplage faible** du thème 1 !

Stop, down, down -v

`stop` : pause, tout gardé.
`down` : conteneurs supprimés, **volume gardé**.
`down -v` : volumes supprimés, base vierge.

Douze phrases à savoir reconnaître vraies ou fausses

AFFIRMATION	VERDICT
Un POS fonctionnel montre les serveurs et les bases	FAUX
10 applis en point-à-point = 45 liaisons	VRAI
Un fichier Excel métier est une architecture 1-Tier	VRAI
Les microservices sont le bon choix par défaut pour une PME	FAUX
En microservices, chaque service a sa propre base	VRAI
Un serveur REST garde la session de l'utilisateur	FAUX

AFFIRMATION	VERDICT
POST est idempotent	FAUX
Une suppression réussie renvoie souvent 204, sans corps	VRAI
Prix « abc » envoyé à FastAPI → 422	VRAI
La clé d'API est vérifiée dans `schemas.py`	FAUX dependencies.py
Un conteneur embarque son propre OS complet	FAUX c'est la VM
`docker compose down` efface les données PostgreSQL	FAUX il faut -v

LA VEILLE DE L'EXAMEN

Si vous savez expliquer ces 6 lignes à voix haute, le 4,5 est à portée de main.

URBANISATION

4 visions (métier, fonctionnelle, applicative, infra) · Zone › Quartier › Îlot · 3 règles d'or.

ARCHITECTURES

1/2/3-Tiers et leurs couches P-M-D · monolithe modulaire vs ESB vs microservices · « le plus simple qui suffit ».

REST

Ressources = noms, actions = verbes · sans état · idempotence · 200/201/204/401/403/404/422/500.

FASTAPI

Starlette + Pydantic · path vs query · Depends() · /docs · uvicorn --reload.

GENEVALOGISTICS

main / schemas / dependencies / database / routers · X-API-Key · 422 sans clé, 401 mauvaise clé.

DOCKER

VM vs conteneur · Dockerfile › image › conteneur · Compose : db, api, pgadmin · volume vs bind mount.

Vendredi : arriver avant 9h05 (« soyez ponctuels », disent vos enseignants).