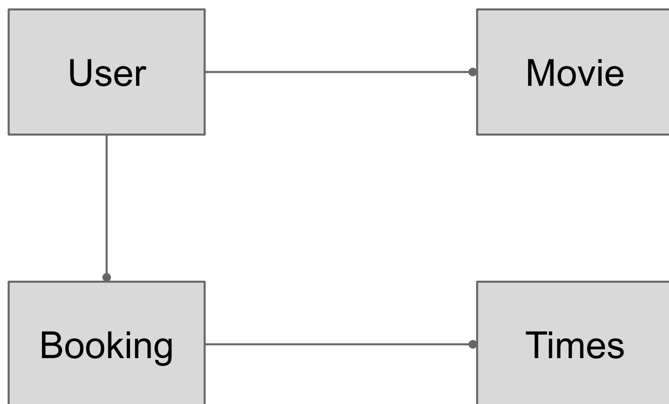
API et communications

Un peu d'histoire

Composant logiciel et architecture distribuée



- Chaque composant a un rôle qui lui est propre et peut être codé par des développeurs différents,
- ce qui est utilisé (client) et ce qui est fourni (serveur) par le composant est clairement spécifié,
- les composants peuvent être distants, écrits dans des langages différents, et hébergés sur du matériel ou des OS différents.

Exemples d’architectures à composants

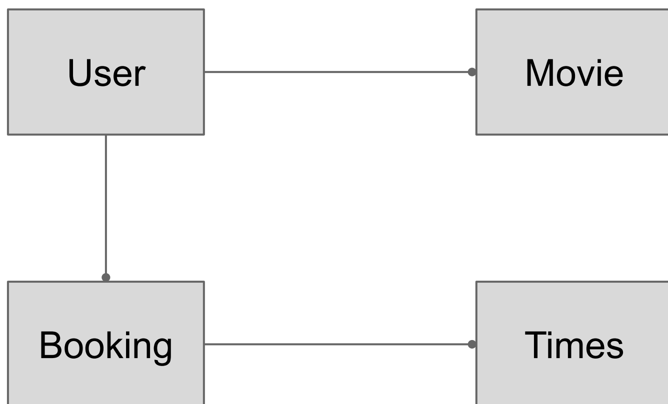
- Architecture client/serveur
- Service-Oriented Architecture (SoA)
- Microservices
- Workflow et dataflow (Stream processing, IoT)
- etc.

Cas particulier des applications parallèles

- L’exécution du programme crée un ensemble de processus distants qui exécutent en parallèle une partie du programme, ou bien le même programme sur des données différentes.
- Nous ne sommes plus sur la logique de composants ou d’objets distants à proprement parlé mais sur une logique de parallélisation, d’accélération.
- “Message Passing Interface” (MPI 1991) est la solution généralement utilisée pour créer le programme distribué parallèle et pour faire communiquer les processus créés.

En dehors du scope de ce cours (Parallélisme, calcul distribué, simulation numérique, calcul haute performance)

Faire communiquer des composants ?

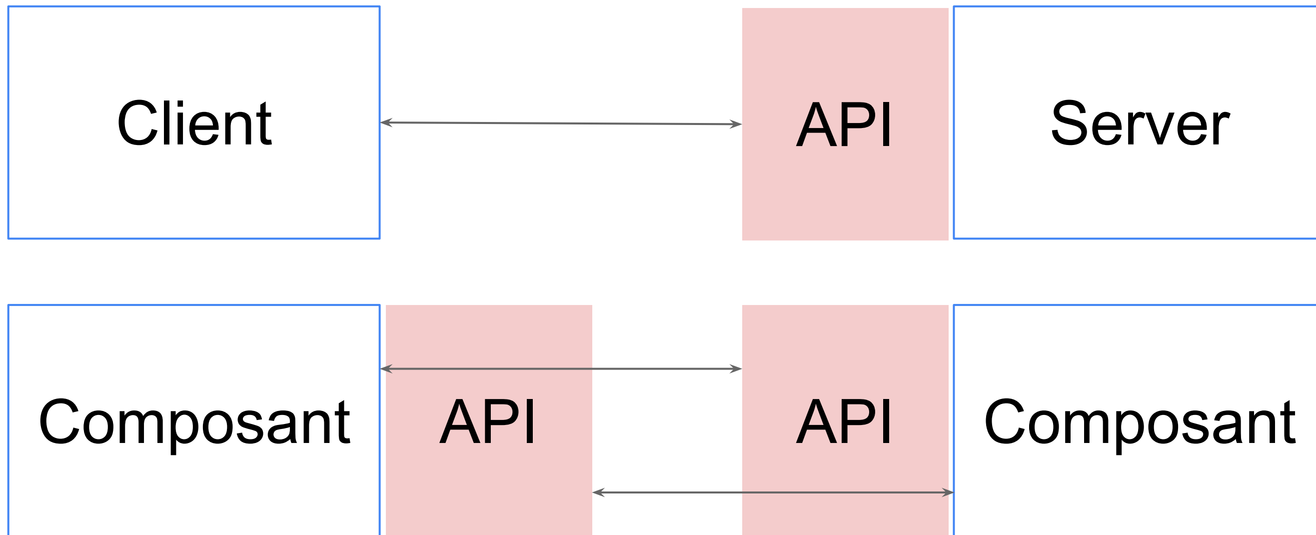


- Coder à la main l'utilisation de protocoles réseau ? (TCP/IP socket, websocket, HTTP etc.)
  - **Besoin d'abstraction !**
- Comment connaître les interactions possibles avec les autres composants ?
  - **La notion d'API "Application Programming Interface"**

Utilisation de bibliothèques et de framework pour régler ces questions !

API

Accord ou contrat entre un client et un serveur sur les interactions possibles



CORBA

"Common Object Request Broker Architecture" (OMG **1991**)

- Remote Procedure Call (**RPC 1980**)
  - appel à distance par un **client** d'une méthode/procédure fournie par un **serveur**
  - définition d'une **API de communication** par "Interface Definition Language" (**IDL**)
  - basé sur le pattern de **proxy** (wrapper) qui englobe les composants client/serveur
  - le proxy, qui est généré automatiquement, cache la gestion des communications
  - basé sur **TCP/IP**
- Composants tournant sur des OS différents
- Composants écrits dans différents langages
- Composants sur des réseaux différents

Plus tard dans ce cours nous verrons une version moderne du RPC et de CORBA avec gRPC.

SOAP

“Simple Object Access Protocol” (**1998** avec les “web services”)

- La réponse de Microsoft à CORBA
- Standard d’échange de messages entre “web services” utilisant le format **XML**
- Repose sur le protocole **HTTP** de la couche applicative OSI
    - HTTP disponible sur n’importe quel OS et pour n’importe quel langage
    - Portabilité des communications
- Même principe d’invocation de méthodes/procédures (**RPC**)
    - SOAP est le successeur de XML-RPC
    - définition d’une **API de communication** par “Web Service Definition Language” (**WSDL**)
- SOAP est toujours réputé pour sa **sécurité** et utilisé par exemple pour les systèmes bancaires ou d’assurances !

Le web et l’ émergence des API REST

REST

“REpresentational State Transfer” (Roy Fielding **2000**)

Tirer profit de la **flexibilité du web** pour le design d’applications et de systèmes distribués :

> *Penser son application/système comme une page web.*

Permet de résoudre le problème principal lié au RPC :

- le manque de flexibilité de l’interface
  - besoin d’un accord/contrat entre le code client et le code serveur
  - un changement change le client et le serveur

REST

"REpresentational State Transfer" (Roy Fielding 2000)

Style architectural de systèmes distribués soumis à 5 principales contraintes :

1. "Client-server"
   - \+ séparation entre l'interface vs le calcul et stockage de données
   - \+ meilleure portabilité des interfaces

2. "Stateless"
   - \+ visibilité et observation facilitée
   - \+ fiabilité
   - \+ passage à l'échelle
   - \- performance du réseau

3. "Cache"
   - \+ performance du réseau
   - \- fiabilité

4. "Uniform interface"
   - \+ découplage implémentation et service rendu
   - \- efficacité

5. "Layered system"
   - \+ réduction de la complexité du système
   - \+ évolutivité
   - \+ passage à l'échelle
   - \- efficacité (surcoût et latence)

REST en pratique

- [Niveau 1] Orienté ressources
  - pas d'utilisation de verbes comme dans RPC
  - un point d'accès (une adresse) pour chaque ressource
  - Utilisation des URI comme adressage
- [Niveau 2] Utilisation des opérations standardisées de HTTP
  - les actions/opérations/verbes sont ceux de HTTP (GET/POST/PUT/DELETE)
  - les opérations sont sûres et suivent le standard
  - par exemple en utilisant GET on est certain que l'on peut mettre en cache
- [Niveau 3] Découverte de l'API du serveur
  - comme pour le web une réponse donne les URI de ce qui peut être fait ensuite
  - flexibilité et évolutivité
  - nouvelles URI peuvent être ajoutées

Très bon article à lire : https://www.martinfowler.com/articles/richardsonMaturityModel.html

Tuto

Python Flask

Sujet sur Moodle !

Remarques sur FastAPI

Une autre possibilité que Flask aurait été d'utiliser FastAPI.
Nous n'aurons pas le temps de regarder mais cette solution semble très intéressante pour coder une API REST et ensuite extraire automatiquement sa documentation.

https://fastapi.tiangolo.com/

Documenter son API

OpenAPI

https://swagger.io/docs/specification/about/

Autre standard : RAML

Tuto

OpenAPI

Sujet sur Moodle !
Créer éventuellement un compte sur SwaggerHub.

https://app.swaggerhub.com/home

Est-ce que l’on fait vraiment du REST ?

Discussion sur le niveau 3 de REST

Pour atteindre le niveau 3 il faut pouvoir découvrir les points d'entrée d'une API. Il y a 2 types de découvertes possible :

- découverte de l'API faite par le développeur :
    - documentation de l'API
    - exploration des URI dans la réponse aux requêtes
    - dans tous les cas il semble qu'il faille "hardcoder" les URI dans le code client
        - découplage client/serveur ?
        - flexibilité tant recherchée ?
        - différence avec RPC ?
- découverte de l'API automatique :
    - semble être le "graal" de REST
    - en pratique il faut implémenter des mécanismes de découverte
    - Pour plus d'infos HATEOAS et JAREST

ATTENTION *: à ne pas confondre avec la* ***découverte de services*** *qui concerne la découverte des adresses IP des services ! On parle ici de la* ***découverte de l'API*** *c'est un peu différent.*

Beaucoup de débats sur Internet

- REST API
  - dans la pratique très peu d'API suivent l'architecture REST car ne répondent pas au niveau 3 de façon automatique.
- OpenAPI
  - la plupart des API que l'on peut trouver s'arrêtent au niveau 2 et sont parfois appelées simplement des API OpenAPI. La différence avec RPC se situe principalement dans l'orientation ressources.
- RPC et gRPC
  - gRPC (cours suivants) implémente le principe RPC sur HTTP/2 et est de plus en plus utilisé pour sa performance et son fonctionnement pour beaucoup de langages.

TP #1

Sujet sur Moodle !