R4.01 : Architecture logicielle

Nicolas Le Sommer
Nicolas.Le-Sommer@univ-ubs.fr
Université Bretagne Sud, IUT de Vannes, Département Informatique

Plan du cours

- Patrons architecturaux
- Patrons de conception et boîtes à outils
- Programmation orientée composants et orientée services
- Services Web
- Service RESTFul et API REST
- Gestion des dépendances logicielles
- Sécurisation d'une API REST

Patrons architecturaux

Patrons architecturaux et applications graphiques

- Les applications graphiques mettent généralement en œuvre l'un des principaux patrons architecturaux suivants :
  - MVC : Modèle-Vue-Contrôleur
  - MVP : Modèle-Vue-Présentation
  - MVVM : Modèle-Vue-Vue-Modèle
- Quelques variantes de ces patrons architecturaux
  - PAC (*Presentation-Abstraction-Control*)
  - HMVC (*Hierarchical Model-View-Controller*)
  - MVPVM (*Model-View-Presenter-View-Model*)
- La plupart des cadres de conception Web côté client mettent en œuvre des patrons architecturaux MVC, HMVC ou MVVM

Objet des patrons architecturaux MVC, ...

- Les patrons architecturaux (MVC, HMVC, MVVM, ...) visent à séparer la logique métier de la logique de présentation
- Logique métier
  - Spécifique au domaine d'application
  - Ne traite pas les problèmes d'interactions avec l'utilisateur
- Logique de présentation
  - Logique spécifiant comment l'interface utilisateur réagit
    - aux interactions utilisateurs
    - aux changements induits dans le modèle de données

Le patron MVC (Modèle, Vue, Contrôleur)

- Patron architectural défini en 1979 et permettant de séparer
  - les données et la logique métier
  - l'interface homme-machine
  - la logique de contrôle
- Le modèle
  - Logique métier et manipulation des données
- La vue
  - Interface utilisateur affichant les données retournées par le modèle
  - Plusieurs vues possibles pour un même modèle
- Le contrôleur
  - définit les interactions entre la vue et le modèle
  - interprète la requête du client et retourne la réponse associée
-

Le patron MVC (Modèle, Vue, Contrôleur)

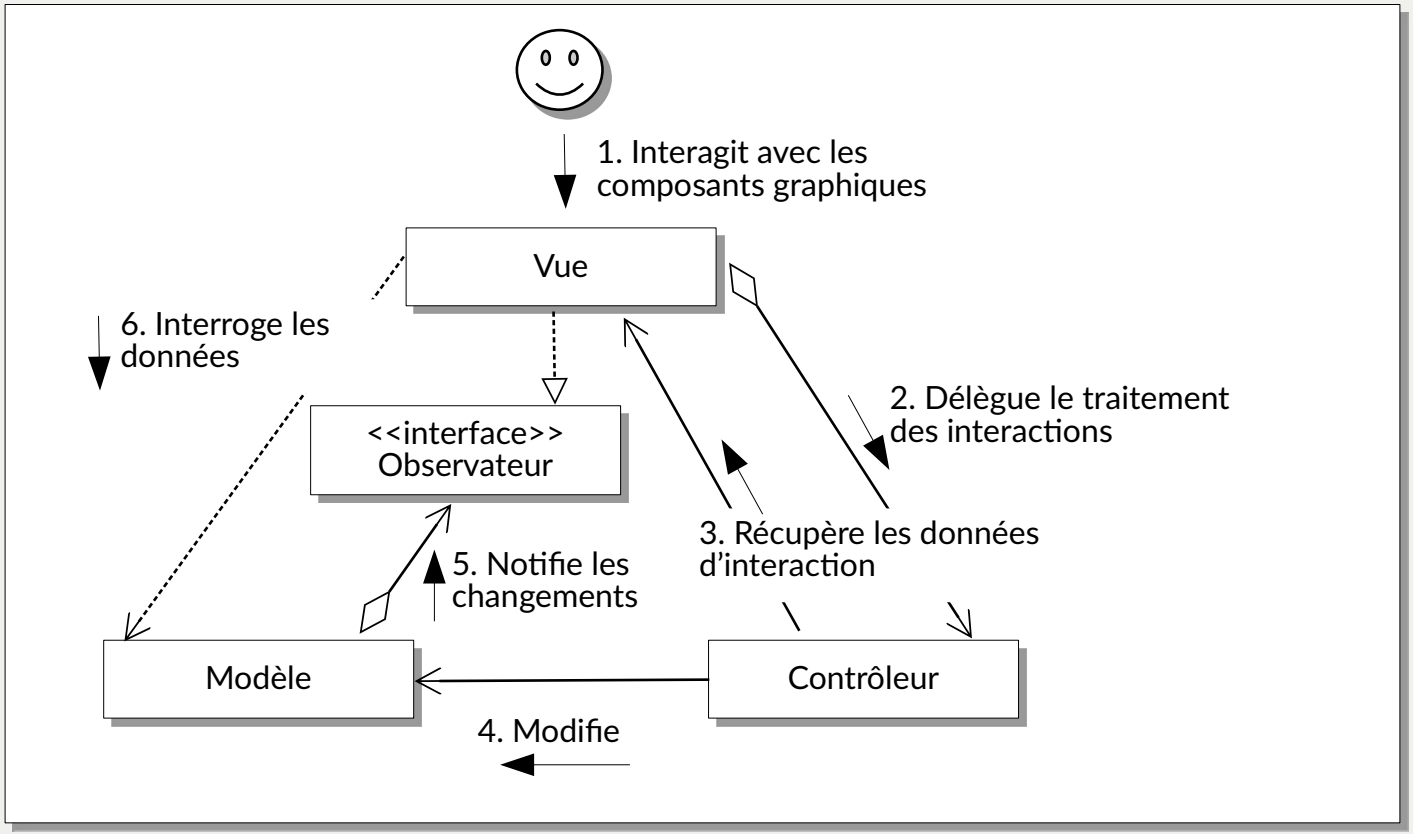
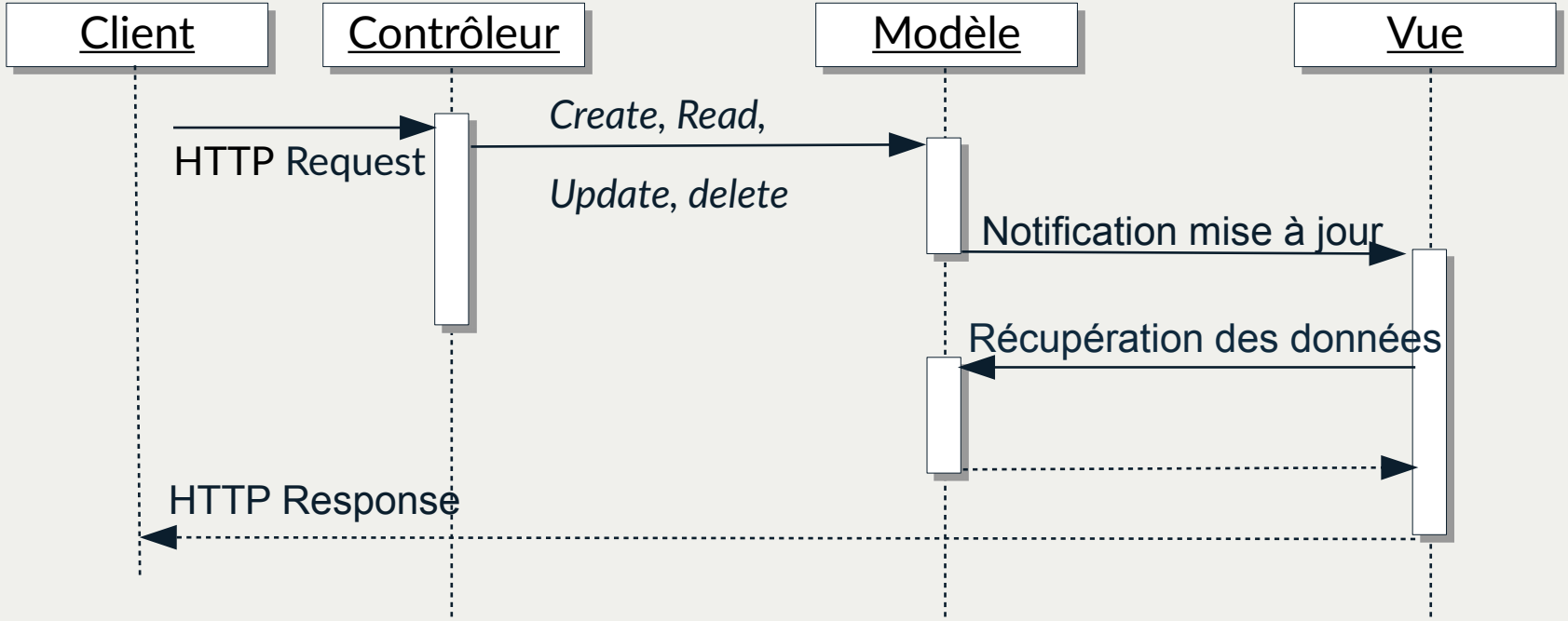


Diagramme de communication du patron MVC

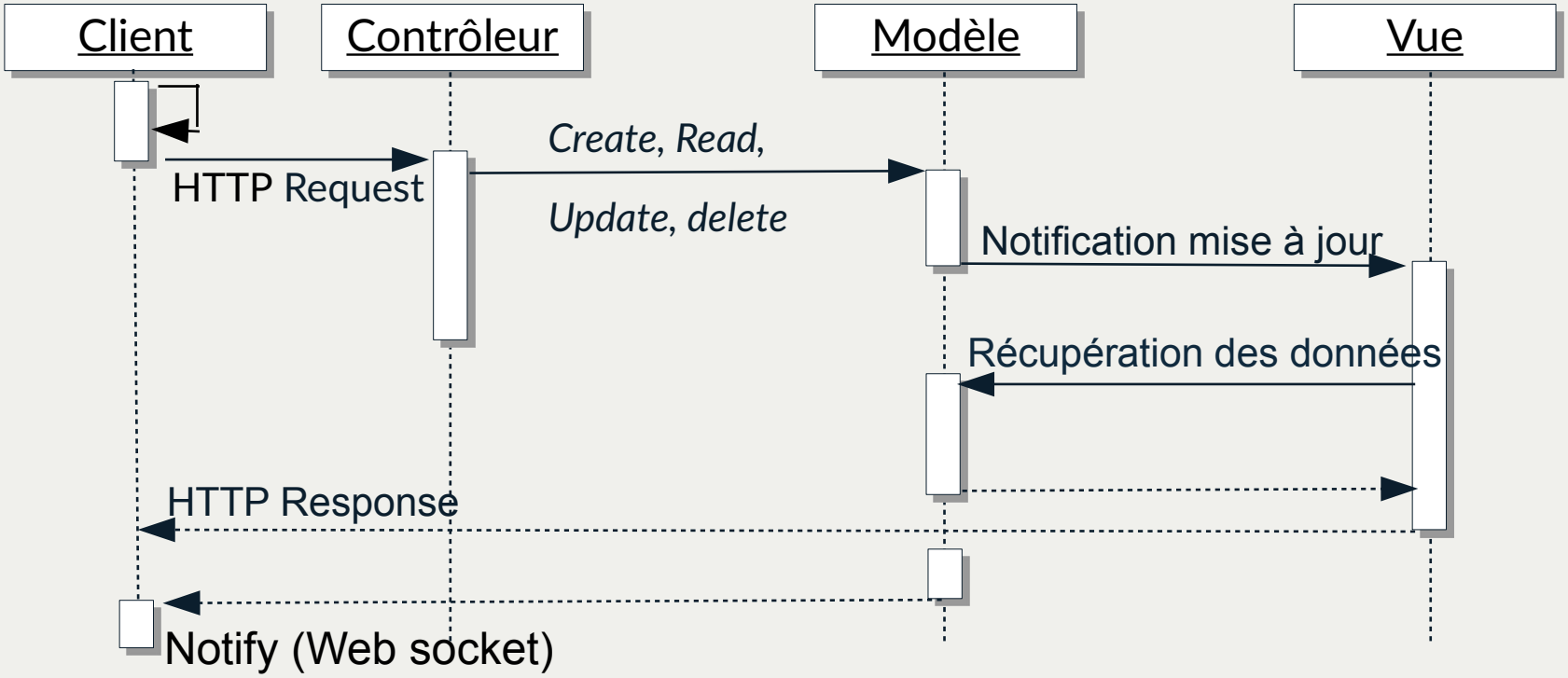
MVC et application Web

- Illustration du fonctionnement du patron MVC pour une application Web simple à travers un diagramme de séquences



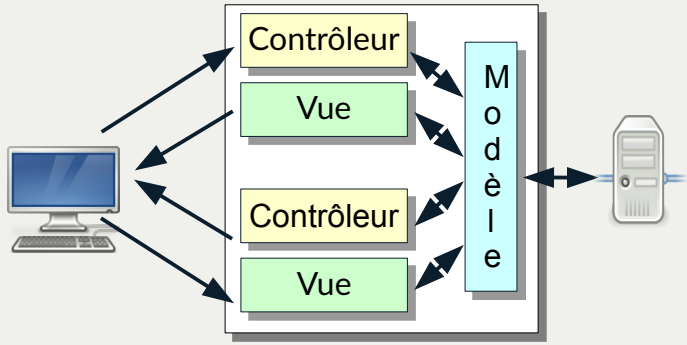
MVC et application Web

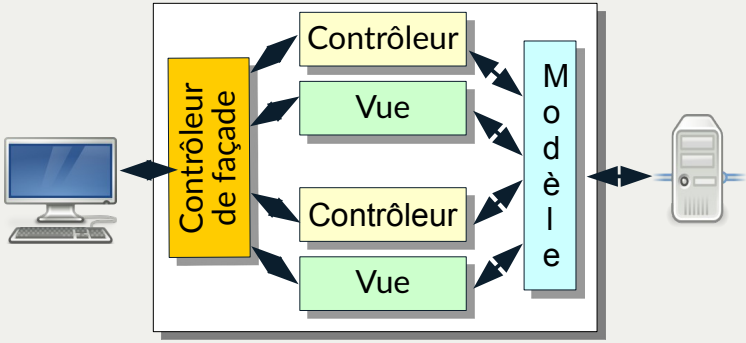
- Illustration du fonctionnement du patron MVC pour une application Web riche à travers un diagramme de séquences



MVC 1 *vs* MVC 2

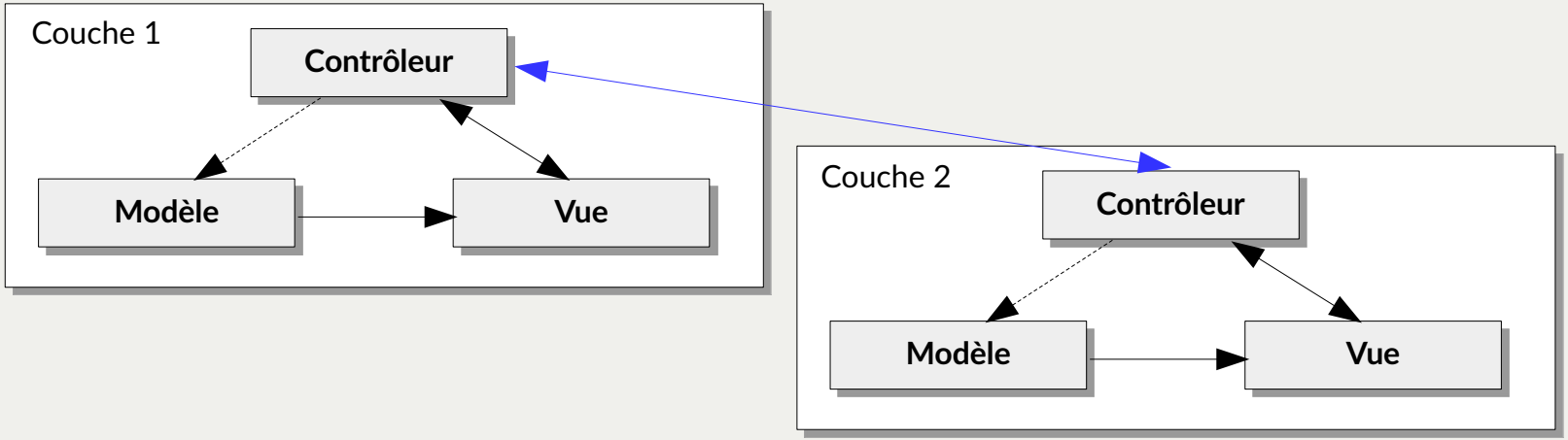
- 2 implantations du modèle MVC dans les applications Web
- MVC 1
  - Multiples points d'entrée dans l'application
  - Un contrôleur par action possible
- MVC 2
  - Contrôleur de façade = LE point d'entrée de l'application
    - Reçoit toutes les requêtes, et les redirige vers les bons contrôleurs
    - Regroupement des actions du point de vue fonctionnel





Le patron HMVC

- Évolution du modèle MVC (proposée en 2000)
  - répondant au problème de passage à l'échelle du MVC
  - principalement utilisée dans le développement d'applications Web
- Réinterprétation du modèle PAC (Presentation-Abstraction-Control) présenté en 1987
- Une couche peut requérir le traitement d'une autre via son contrôleur



Le patron MVP (Modèle / Vue / Présentation)

- Le modèle est centré sur la logique métier
- La vue est indépendante du reste de l'application
- La présentation
  - contient la logique de présentation et l'état courant du dialogue avec l'utilisateur
  - réagit aux événements de l'utilisateur en modifiant les données si nécessaire, et en répercutant les effets sur la vue
- Le patron MVP permet de réduire le couplage entre la vue et le modèle
  - Les interactions passent par le contrôleur

Le patron MVP (Modèle / Vue /Présentation)

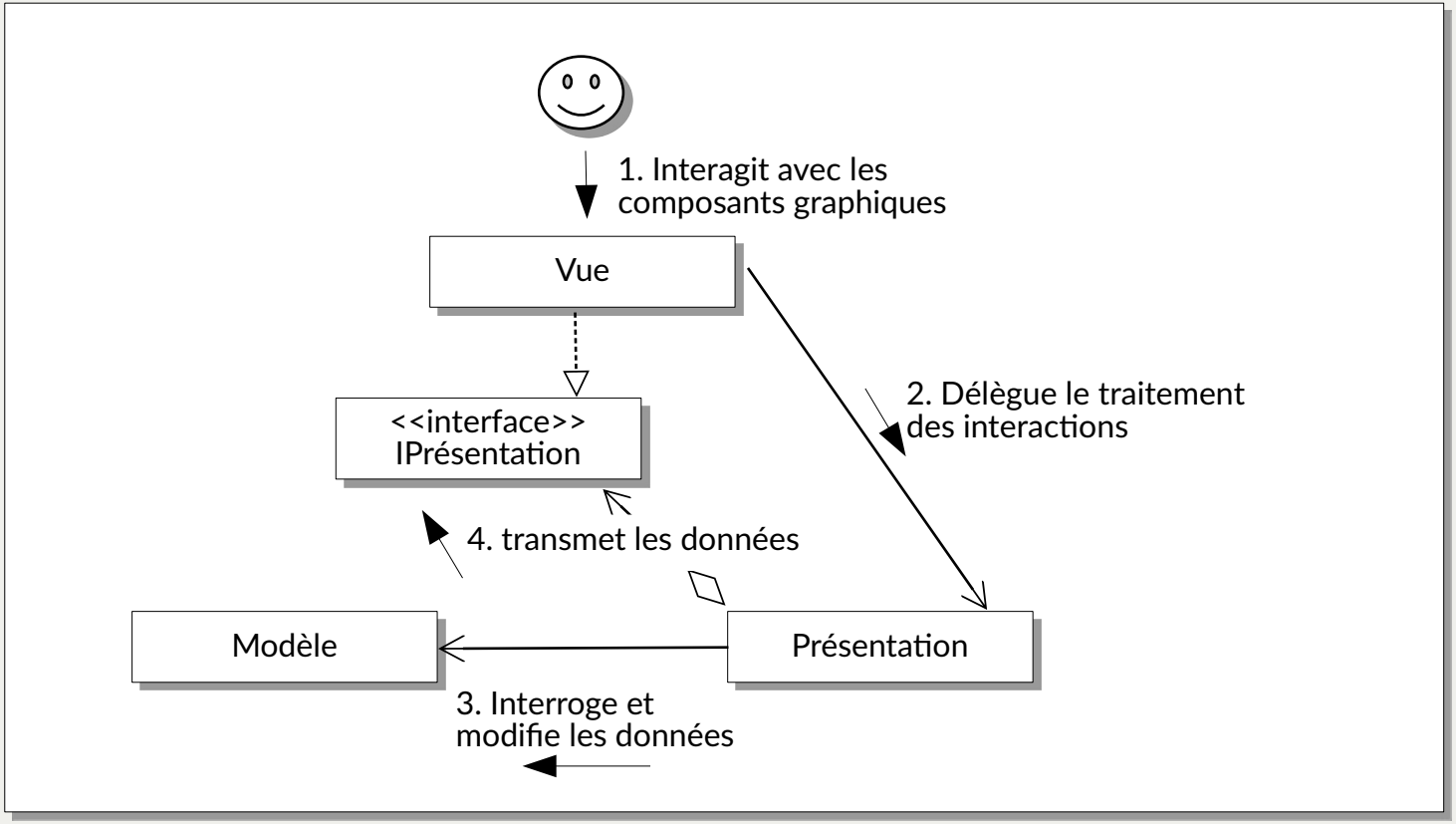


Diagramme de communication du patron MVP

Exemple de mise en œuvre du patron MVP

```
// modèle de données
class Car{
      private String model ;
      private int year;
      private String matriculation ;
      …
      public Car(String m, int year, String mat){..

      public String getModel(){...}
      …
}
```

```
// présentation
class CarListPresenter{
      private CarListView view;
      private List<Car> cars;

      public CarListPresenter(CarListView v, List<Car> c){
            this.view = v;
            this.cars = c;
      }

      public void onCarSelected(String m) {
            List<Car> res = new ArrayList<>() ;
            for(Car c : this.cars){
                  if(c.getModel().equals(m)){
                        res.add(c) ;
                  }
            }
            this.view.showList(res) ;
}
```

```
// vue
class CarListView implements Ipresentation<Car>{
      @Override
      public void showList(List<Car> cl){
      ...};
      ...
}
```

Le patron MVVM (Modèle/Vue/Vue/Modèle)

- Le modèle
  - Comparable au modèle de données défini dans MVP
    - Contient uniquement la logique métier
- La vue
  - La différence avec MVP réside dans l'utilisation de liaisons dynamiques (*binding*) entre des composants de la vue et des attributs de la vue-modèle
- La vue-modèle
  - Contient la logique de présentation et l'état de l'interface comme la présentation de MVP
- De nombreux cadres de conception implantent le modèle MVVM
  - ZK, ReactJS, ReactNative, AngularJS, Angular, Vue.js, Flutter, JavaFX, ...

Le patron MVVM (Modèle/Vue/Vue/Modèle)

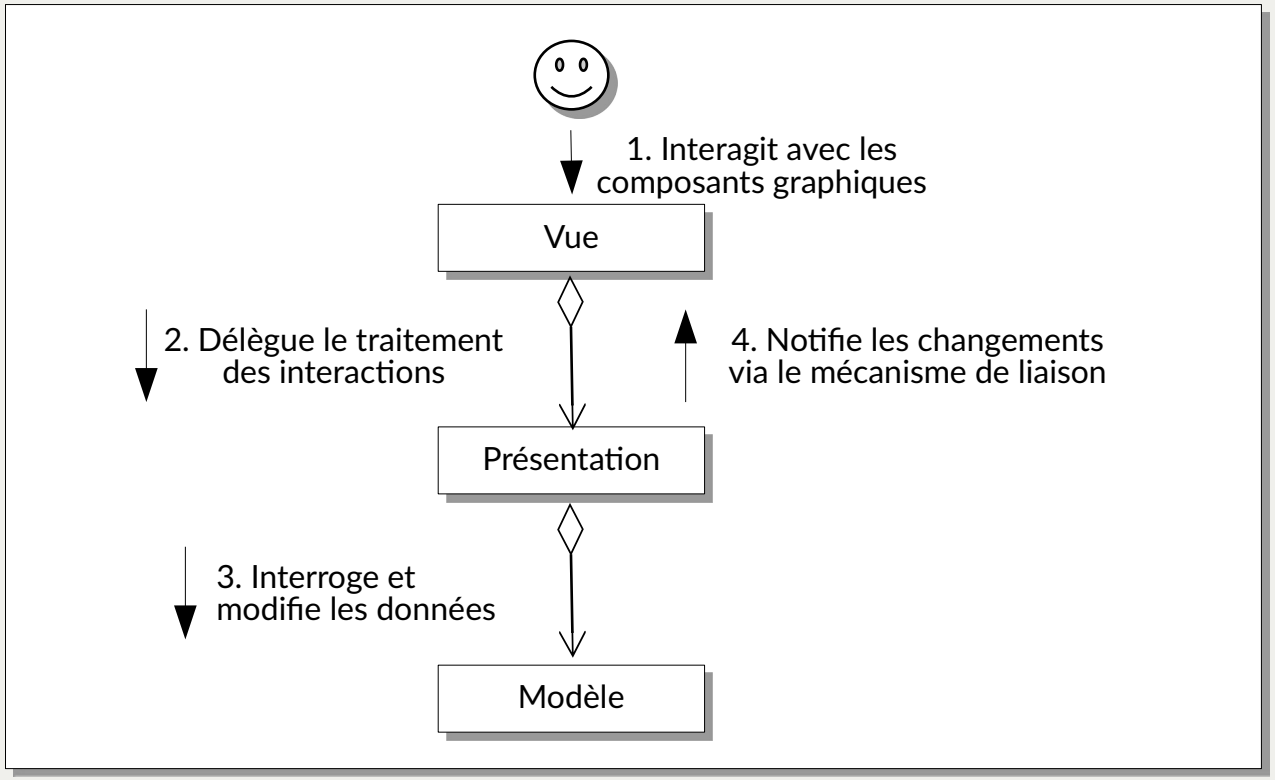


Diagramme de communication du patron MVVM

Exemple de mise en œuvre du patron MVVM

```
//classe vue-modèle
public class SearchViewModel {

    private String keyword;
    private List<Car> carList;
    private Car selectedCar;

    private CarService carService = new CarServiceImpl();

    public void setKeyword(String keyword) {
        this.keyword = keyword;
    }
    public String getKeyword() { return keyword; }

    public List<Car> getCarList(){ return carList; }

    public void setSelectedCar(Car selectedCar) {
        this.selectedCar = selectedCar;
    }
    public Car getSelectedCar() { return selectedCar; }


    @Command
    @NotifyChange("carList")
    public void search(){
        carList = carService.search(keyword);
    }
}
```

Attributs liés à la vue

Notification des changements de la liste

```
// modèle de données
class Car{
      private String model ;
      private int year;
      private String matriculation ;
      …
      public Car(String m, int year, String mat){...   }

      public String getModel(){...}
      …
}
```

Exemple de modèle, *framework* zk (https://www.zkoss.org/)

Exemple de mise en œuvre du patron MVVM

Nom de l'instance de la classe vue-modèle

Classe de liaison

Classe Vue-Modèle

Attribut lié

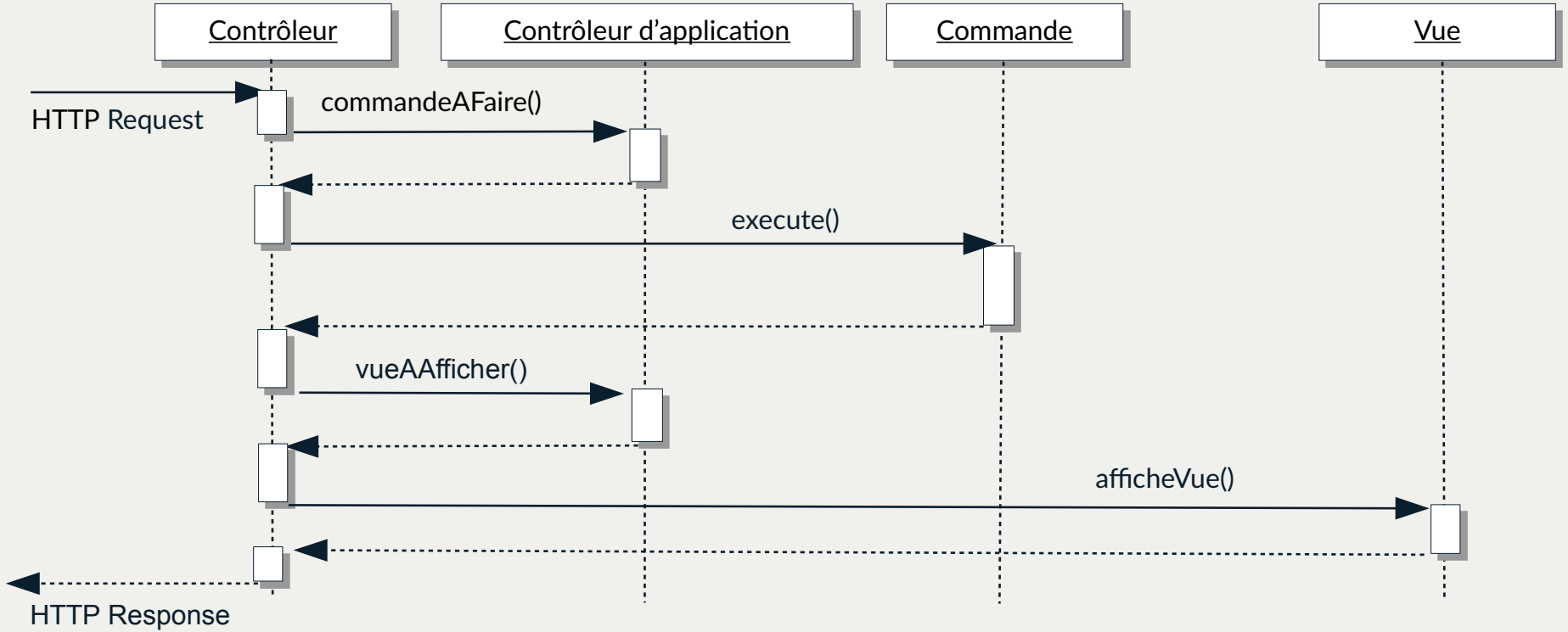
```
<window title="Search" width="600px" border="normal" apply="org.zkoss.bind.BindComposer"
    viewModel="@id('vm') @init('demo.getting_started.mvvm.SearchViewModel')">
    <hbox align="center">
        Keyword:
        <textbox value="@bind(vm.keyword)" />
        <button label="Search" image="/img/search.png" onClick="@command('search')" />
    </hbox>
    <listbox height="160px" model="@bind(vm.carList)" emptyMessage="No car found in results"
    selectedItem="@bind(vm.selectedCar)" style="margin-top:10px">
        <listhead>
            <listheader label="Model" />
            <listheader label="Make" />
            <listheader label="Price" width="20%"/>
        </listhead>
        <template name="model">
            <listitem>
                <listcell label="@bind(each.model)"></listcell>
                <listcell label="@bind(each.make)"></listcell>
                <listcell>$<label value="@bind(each.price)" /></listcell>
            </listitem>
        </template>
    </listbox>
</window>
```

Patron	Avantages	Inconvénients
MVC	• Facilité pour ajouter des vues	• Boucle d'action complexe • Séparation des préoccupations incomplètes puisque la vue doit récupérer les données
MVP	• Réduit le couplage entre la vue et le modèle (les interactions passent par le contrôleur) • Vue simplifiée et dépourvue de logique applicative • La vue est découplée de la présentation par l'utilisation de l'interface « IPrensentation » • La réimplantation de la vue avec une autre bibliothèque est facilitée	• La présentation contient beaucoup de code redondant (la présentation doit modifier la vue lorsqu'une information doit être retournée à l'utilisateur)
MVVM	• Réduit la redondance de code dans la présentation par rapport à MVP	• Mise en œuvre complexe des liaisons si elles ne sont pas fournies par le cadre de conception • Difficulté d'implanter le patron Observateur sur tous les attributs de la vue

Quelques autres patrons de conception utiles

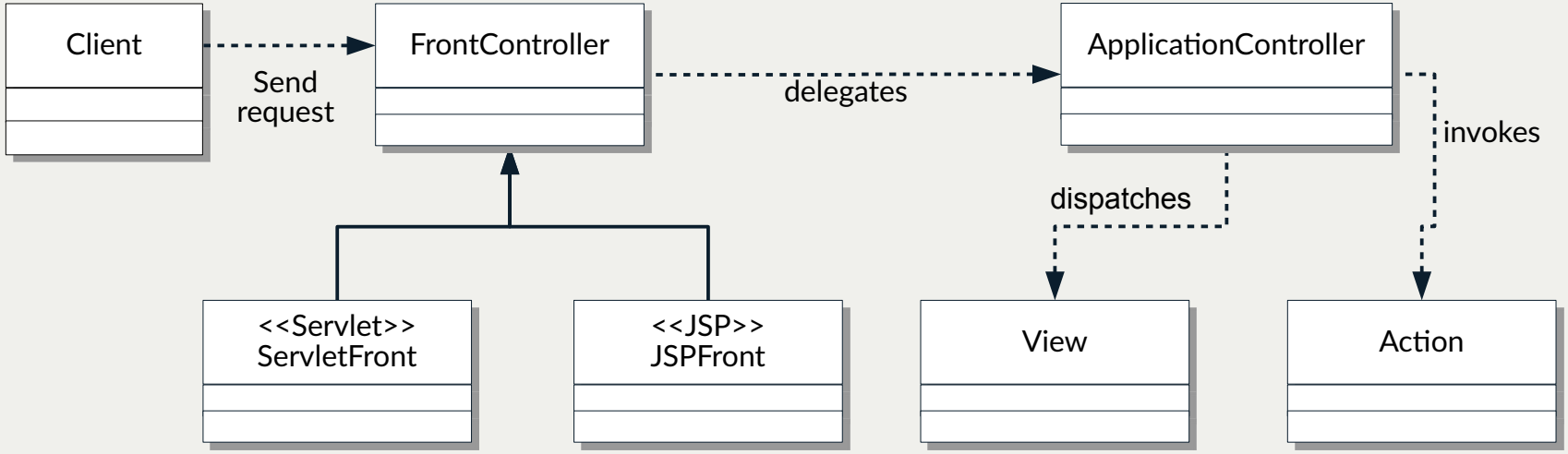
Le patron de conception « contrôleur d'application »

- Contrôleur d'application (*Application Controller*) : point central d'une application définissant les traitements à effectuer et les vues à afficher (en fonction d'un flux d'enchaînement d'actions)



Le patron de conception « contrôleur de façade »

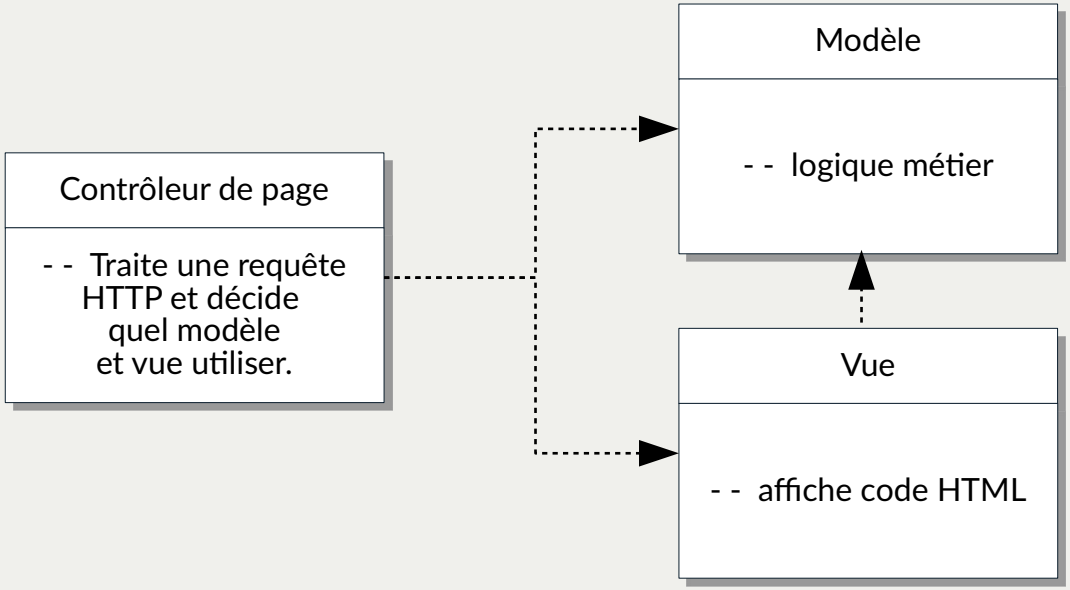
- Permet d'avoir un seul point d'entrée dans l'application
- Délègue le traitement des actions à des « gestionnaires » spécialisés



Modélisation du *FrontController* dans J2EE

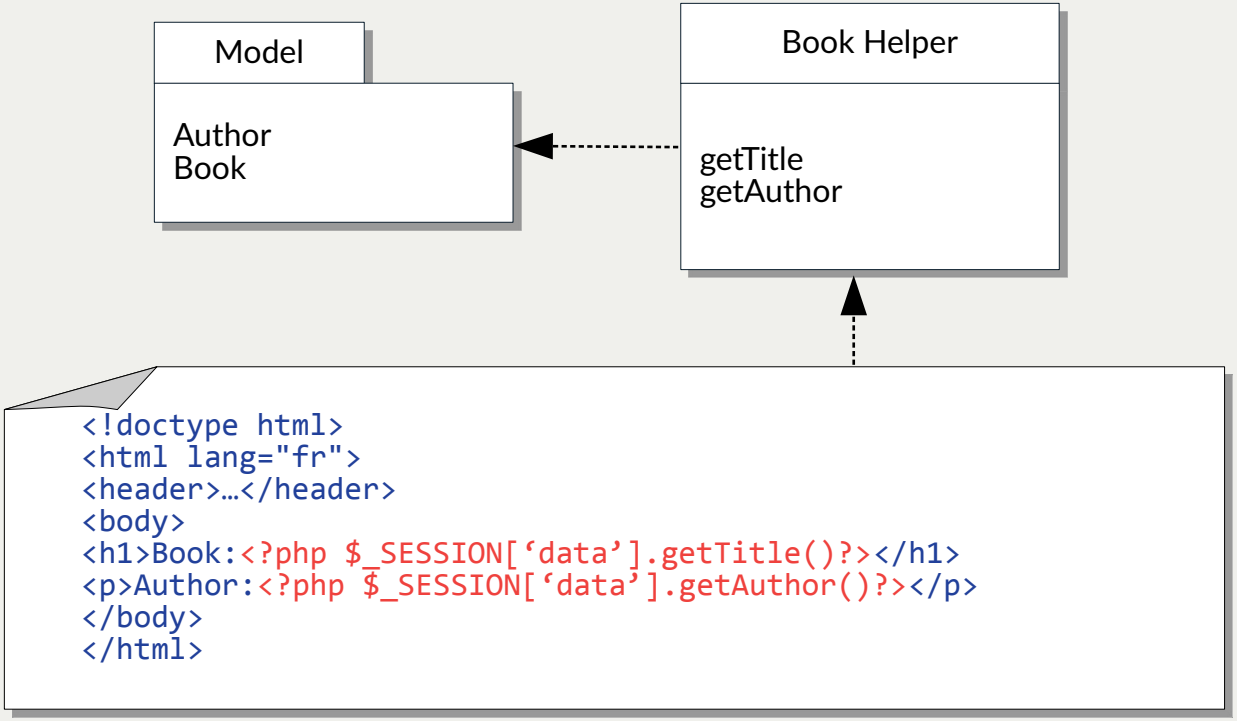
Le patron de conception « contrôleur de page »

- Contrôleur de page : objet permettant de traiter une requête pour une page spécifique d'une application Web



Le patron de conception « modèle de vues »

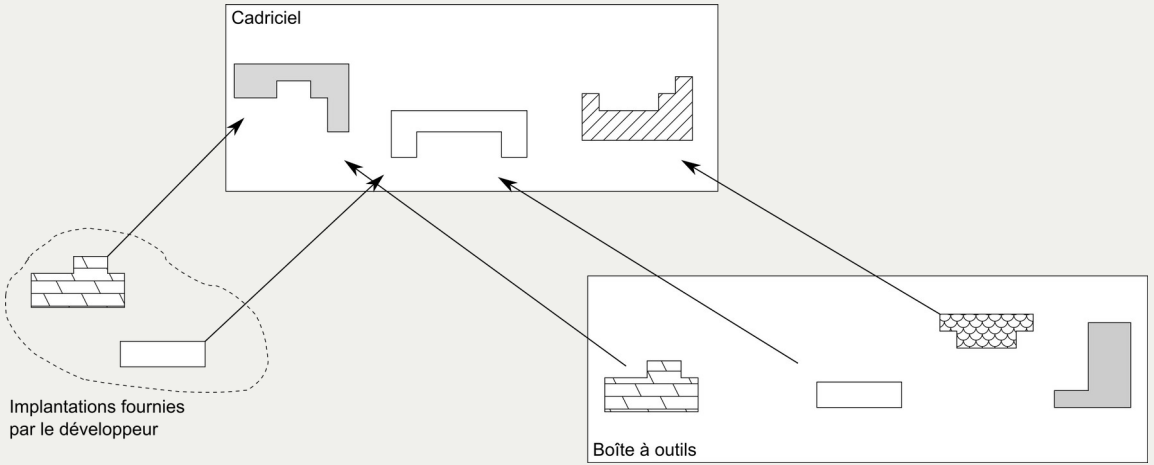
- Modèle de vue : objet produisant du code HTML à partir de code HTML statique, et incorporant dynamiquement du code HTML pour mettre en forme et afficher des données issues du modèle



Cadriciel, boîtes à outils, composants et services

Cadriciel et boîte à outils

- Cadriciel / Cadre de conception (*framework*) : définition
  - Un cadriciel est un ensemble structuré d'éléments logiciels (interfaces, classes, composants, services) définissant en tout ou partie l'architecture d'une application
- Boîte à outils (*toolkit*) : définition
  - Une boîte à outils est un ensemble d'éléments logiciels (classes, composants, services) fournissant des implantations concrètes de fonctionnalités
  - Les éléments d'une boîte à outils peuvent compléter un cadriciel



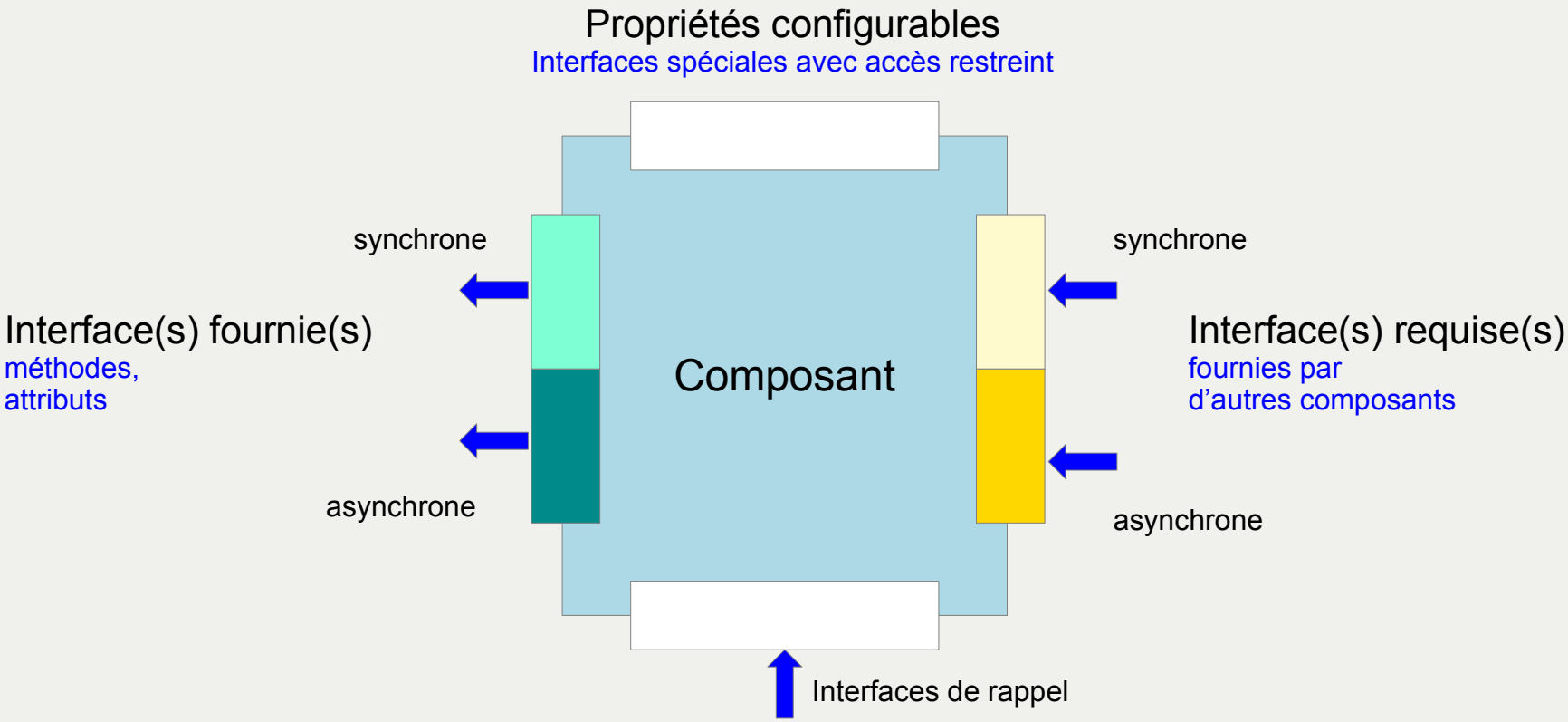
Limites de la programmation par objets

- Pas d'expression des ressources requises
  - Seules sont définies les interfaces fournies, non celles requises
- Absence de vision globale de l'application
  - Les principaux concepts sont définis au niveau d'un objet individuel
  - Pas de notion de description globale de l'architecture
- Difficulté d'évolution
  - Conséquence de l'absence de vision globale
- Absence de certains mécanismes pour certaines applications/infrastructures
  - Les mécanismes doivent être réalisés par les développeurs (persistance, sécurité, tolérance aux fautes, etc.)
- Absence d'outils d'administration
  - Composition, déploiement
- Conclusion : charge importante pour le développeur, incidence sur la qualité de l'application, une partie du cycle de vie n'est pas couverte

Composant logiciel : définition

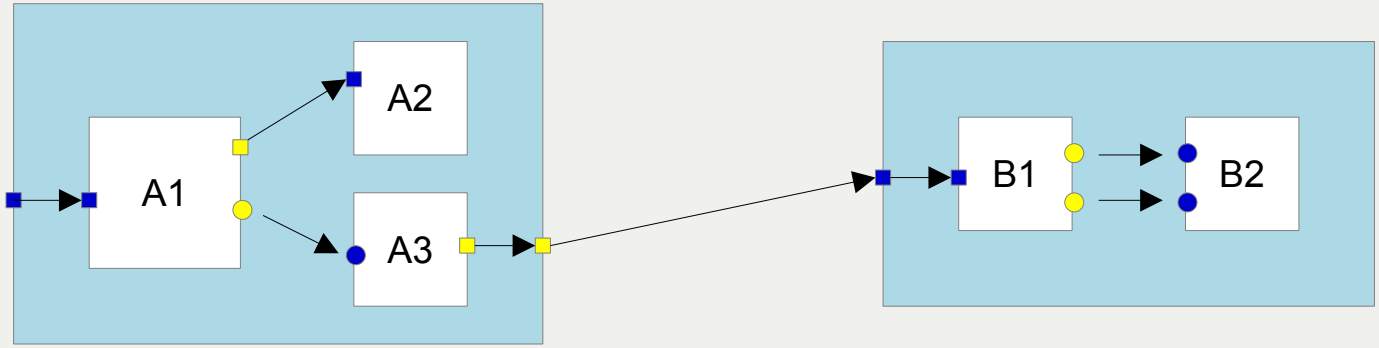
- Un composant logiciel est un module logiciel autonome
  - Unité de déploiement
    - installation sur différentes plates-formes
  - Unité de composition
    - combinaison avec d'autres composants
- Propriétés d'un composant
  - Spécifie explicitement la (ou les) interface(s) fournie(s) (attributs, méthodes)
  - Spécifie explicitement la (ou les) interface(s) requise(s) pour son exécution
  - Peut être configuré
  - Capable de s'auto-décrire
- Intérêt
  - Permettre la constructions d'applications par composition de briques de base configurables
  - Séparer les fonctions des fournisseurs et d'assembleurs (conditions pour le développement d'une industrie des composants)

Composant : modèle générique



Composants : utilisation

- Composition hiérarchique et encapsulation
- Interconnexion de composants (connecteurs ou objets de communication)



Support logiciel pour composants

- Pour assurer leurs fonctions, les composants ont besoin d'un support logiciel
- On distingue généralement les conteneurs et les structures d'accueil
- Conteneur
  - encapsulation d'un composant
  - prise en charge des services du système
    - nommage, sécurité, transactions, persistance, etc.
  - prise en charge partielle des relations entre composants (connecteurs)
    - invocations, événements
  - techniques utilisées : interposition, délégation
- Structure d'accueil
  - espace d'exécution des conteneurs et composants
  - médiateur entre conteneurs et services du système

Différents modèles de composants

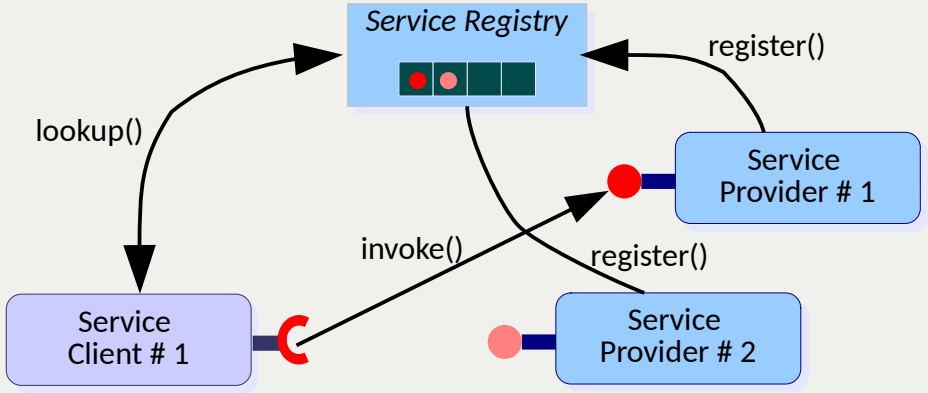
- Standards de facto côte serveur (issus de l'industrie)
  - J2EE(intègre de nombreuses technologies : EJB, JSP, JDBC, JPA, ...)
  - .NET (intègre COM+, ASP.NET, C#, ...)
- Côté client (moins développé)
  - Côté client
- Normalisation par des consortiums
  - OMG : composants CORBA (CORBA Component Model : CCM)
  - OSGI : Modèle plus léger pour applications embarquées, mobiles
- Modèles de composants issus de travaux de recherche
  - Fractal, Avalon

Langages de description d'architecture à composants

- MIL : *Module Interconnection Languages*
- ADL : *Architecture Description Languages*
  - *ACME*, *AADL*, *Darwin*, *Fractal-ADL*
- Objectifs
  - Description globale de l'architecture d'une application
    - Définition des composants
      - interfaces fournies, requises
    - Définition des interconnexions
  - Support d'outils pour la vérification
    - Définition des types des composants
    - Vérification de conformité des interfaces
  - Support d'outils pour le déploiement
    - Annotations spécifiant le déploiement
    - Génération automatique de scripts de déploiement

Programmation orientée services

- Programmation modulaire à l'aide de composants logiciels offrant une (ou des) interface(s) de service et requérant d'autres services
  - Les interfaces utilisent des normes de communication communes permettant aux services d'être rapidement incorporés à de nouvelles applications
- Mécanisme de liaison retardée (*late-binding*)
  - Assemblage dynamique des composants (services) à l'exécution
- La programmation orientée services apporte un dynamisme par rapport à la programmation orientée composants



Micro-services

- Les micro-services permettent de structurer une application sous la forme d'un assemblage de services faiblement couplés.
- Particularité des micro-services :
  - Ils ont un périmètre fonctionnel restreint (fonctionnalité élémentaire)
  - Ils sont déployés de manière indépendante pour faciliter leur mise-à-jour au cours de l'exécution d'une application
    - processus propre à chaque service, conteneur d'exécution propre à chaque service
- Contrairement aux services Web, les micro-services ne sont pas accessibles, et ne communiquent pas, forcément via des protocoles standards et normes issus du Web (HTTP, SOAP, WSDL, ...)
- Les micro-services peuvent communiquer par
  - appel de procédures (à distance) : RMI
  - bus de messages : JMS (*Java Message Service*), MQTT, AMQP, ...

Avantages des micro-services

- Les micro-services peuvent être développés, déployés, gérés et mis à l'échelle sans affecter le fonctionnement des autres services.
- Ils offrent une liberté technologique
  - Le meilleur outil peut être utilisé pour résoudre un problème spécifique

  Les micro-services offrent, comme les composants, une certaine réutilisabilité
- L'indépendance du service augmente la résistance de l'application face aux défaillances.

Web services

- Les services Web sont des services exposés et accessibles via des protocoles et normes standards du Web
- Deux types de services Web :
  - les services Web reposant sur SOAP (*Simple Object Access Protocol*)/HTTP(s)
    - XML-RPC/HTTP n'est plus beaucoup utilisé
  - les services reposant sur le modèle architectural REST (*Representational State Transfer*)
- Description des fonctionnalités des Web services
  - API REST (formalisée avec la spécification OpenAPI)
  - WSDL (*Web Service Description Language*)
- Annuaires de services (enregistrement/découverte des services)
  - UDDI (*Universal Description Discovery and Integration*)
  - Apache ZooKeeper

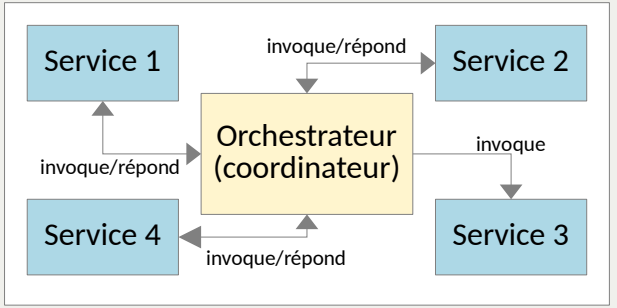
    service centralisé de configuration et nommage fournissant une synchronisation/coordination distribuée entre des groupes de services

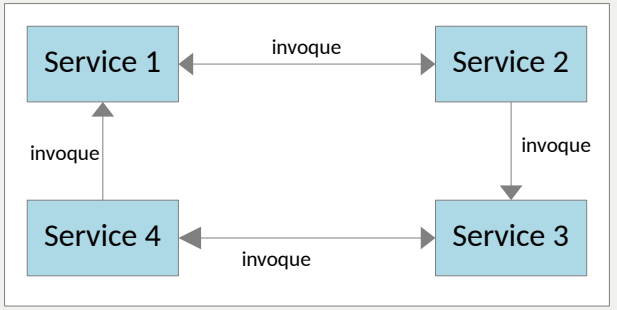
Architecture orientée services

- SOA : *Software Oriented Architecture*
- Création d'applications par assemblage de services
- BPEL (*Business Process Engineering Language*), issu des langages WSFL (*Web Service Flow Language*)
  - Description de l'assemblage de services, des services, des références et des inter-connexions
  - Description des interactions entre les services : orchestration ou chorégraphie

Orchestration *vs* chorégraphie avec BPEL

- Orchestration
  - Un processus central (peut être un autre service) qui contrôle et coordonne l'exécution des services
  - Les services ne savent pas qu'ils sont impliqués dans une composition (dans un service de plus haut niveau)
  - Seul le coordinateur connaît le but de la composition (définition des services et de l'ordre des invocations)
- Chorégraphie
  - Pas de coordinateur central
  - Chaque service connaît quand et avec quel service il doit échanger
  - Chaque participant doit connaître le processus métier, les opérations à exécuter, les messages à échanger





Spécification JAX-RS

- JAX-RS : *Java API for RESTful Web Services*
- JAX-RS est le résultat de la JSR 311 (Java Specification Request) pour Java EE 6 (*Java Entreprise Edition Platform*)
- JAX-RS repose sur des annotations Java
- JAX-RS permet
  - d'identifier les composants de l'application
  - d'aiguiller les requêtes vers les classes et méthodes appropriées (gestion des routes)
  - d'extraire les données des requêtes
  - de retourner des réponses contenant des données et de manipuler les méta-données associées aux réponses

Annotations JAX-RS et exemples d'utilisation

Annotation	Description
@Path	Route traitée par la classe
@PathParam	Les paramètres qui peuvent être passés dans l'URL
@FormParam	Les données indiquées dans le corps d'une requête HTTP POST
@Produces	Type des données retournées dans la réponse
@Get	Permet d'indiquer que la méthode Java traitera la requête GET pour la route définie par @Path
@Post	Permet d'indiquer que la méthode Java traitera la requête POST pour la route définie par @Path

```
package fr.ubs;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;
@Path("/cars")
public class CarService{
    @GET
    @Path("/{param}")
    public Response getMsg(@PathParam("param") String msg) {
        String output = "Parameters : " + msg;
        return Response.status(200).entity(output).build();
    }
}
```

Annotations JAX-RS et exemples d'utilisation

```
package com.javatpoint.rest;
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
@Path("/cars")
public class CarService{
    @POST
    @Path("/add")
    @Produces(MediaType.TEXT_HTML)
    public Response addUser(
        @FormParam("model") String model,
        @FormParam("matriculation") String matriculation,
        @FormParam("year") int year) {

        return "<html><body> Cars added successfuly! <br>"+
               " Model: "+model+"<br> Matriculation: " + matriculation+"<br> Year: "+year)"+
               "</body></html>" ;
    }
}
```

Service RESTFul et API REST

Modèle architectural REST

- Les applications développées selon le modèle architectural REST (*REpresentational State Transfer*) doivent respecter 6 contraintes.
- Contrainte 1 : « client/serveur »
  - séparation des responsabilités entre le client et le serveur.
    - Client : présentation
    - Serveur : sauvegarde et accès des données
- Contrainte 2 : « serveur sans états »
  - chaque requête d'un client contient toute l'information nécessaire pour permettre au serveur de traiter la requête
- Contrainte 3 : « cache »
  - chaque réponse d'un serveur contient une information indiquant si la réponse peut ou non être mise en cache côté client ou côté serveur.
  - Si une réponse peut être mise en cache, elle peut être réutilisée pour des requêtes ultérieures équivalentes.

Modèle architectural REST

- Contrainte 4 : « interface uniforme »
  - Identification unique des ressources
  - Manipulation des ressources par leur représentation
    - Les représentations des ressources permettent de les créer, les modifier ou les supprimer
  - Permet de découpler le client du serveur
- Contrainte 5 : « système en couche »
  - Le client ne doit pas savoir s'il est connecté directement au serveur
  - Des serveurs intermédiaires peuvent être ajoutés pour des raisons de performances et de passage à l'échelle
- Contrainte 6 (optionnelle): « code à la demande »
  - Les serveurs peuvent étendre une fonctionnalité en transférant du code au client (e.g., Javascript)

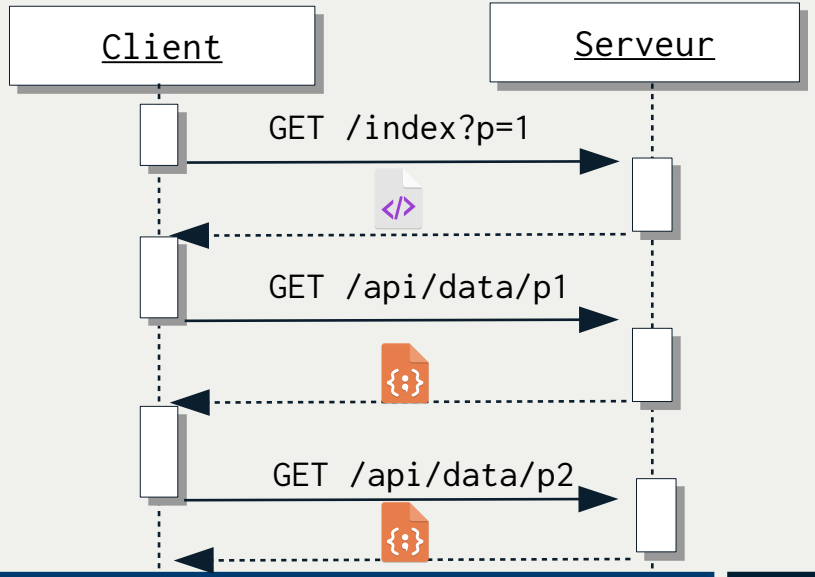
REST, HTTP et Opération « CRUD »

- Dans le modèle REST, les ressources sont identifiées par des URI
- On utilise les méthodes HTTP pour effectuer les opérations élémentaires de création, de consultation, de mise-à-jour et de suppression

Méthode HTTP	Opération CRUD
POST	Create
GET	Read
PUT	Update
DELETE	Delete

Avantage de l'approche

- Découplage complet entre le client et le serveur
- Le serveur retourne
  - du contenu statique (pages HTML, images, code CSS, ...)
  - des données (souvent au format JSON)
  - du code Javascript pour mettre en forme les données et communiquer avec l'application Web via son API REST
- Pas d'état à sauvegarder côté serveur
- Passage à l'échelle plus facile



Exemple d'API REST

Méthode HTTP	URI	Action
GET	/employees	Retourne la liste des employés
POST	/employees	Ajoute un employé à la liste
PUT	/employees	Méthode non autorisée
PATCH	/employees	Méthode non autorisée
DELETE	/employees	Supprime la liste des employés
PUT	*/employees*/employeeID	Modifie les informations de l'employé identifié par employeeID ou le créé
PATCH	*/employees*/employeeID	Modifie les informations de l'employé identifié par employeeID
POST	*/employees*/employeeID	Méthode non autorisée
GET	*/employees*/employeeID	Retourne les informations de l'employé identifié par employeeID
DELETE	*/employees*/employeeID	Supprime l'employé identifié par employeeID

Exemple d'application REST

```
<!doctype html>
<html lang="fr">
<head> <meta charset="utf-8"> </header>
 <body>
    <table border=1 id="tab"></table>
    <script>
      var tab = document.getElementById('tab');
      var xhr = new XMLHttpRequest();
      xhr.open("GET", "http://localhost:8000/employees");
      xhr.onreadystatechange = function () {
        if (this.readyState == 4) {
           let thead = "<tr><th>Nom</th><th>Prénom</th></tr>";
          let jsonResult = JSON.parse(this.responseText);
          let tbody="";
          for(let i=0; i < jsonResult.length; i++){
            tbody+= "<tr><td>"+jsonResult[i].lastname+"</td><td>"+jsonResult[i].firstname+"</td></tr>";
          }
          tab.innerHTML= thead+tbody;
        }
      };
      xhr.send(null);
    </script>
 </body>
</html>
```

OpenAPI

OpenAPI

- OpenAPI est une spécification pour la description des interfaces de programmation (REST)
  - Format de description ouvert et indépendant des services
  - https://spec.openapis.org/oas/latest.html
- Spécification qui découle d'un projet antérieur, nommé Swagger
- L'API est décrite en utilisant YAML ou JSON pour être facilement automatisée
- La spécification établit la structure des objets suivants :

Objet	Description
Info	Nom, version, etc.
Contact	Informations de contact du fournisseur de l'API
Licence	Licence sous laquelle l'API propose ses données
Server	Nom de l'hôte, structure URL et ports du serveur sur lequel sera interrogée l'API
Components	Composants encapsulés qui sont utilisés plusieurs fois dans la définition API
Paths	Chemins d'accès relatifs aux points de terminaison de l'API, utilisés avec le Server
Path Item	Les opérations autorisées à prendre un chemin spécifique, comme GET, PUT, POST, DELETE
Operation	Définit entre autres les paramètres et les réponses du serveur attendues pour une opération

OpenAPI : exemple GET simple

```yaml
openapi: 3.0.0
info:
  version: 1.0.0
  title: Simple Example
  description: A simple API to illustrate OpenAPI concepts

servers:
  - url: https://dptinfo.univ-ubs.fr/

# Basic authentication
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
security:
  - BasicAuth: []

paths:
 /cars:
    get:
      description: Returns a list of cars
      responses:
        '200':
          description: Successful response
```

```yaml
content:
   application/json:
     schema:
       type: array
       items:
             type: object
             properties:
                   model:
                    type: string
                   matriculation:
                    type: string
                   year:
                    type: integer
```

OpenAPI : exemple GET avec paramètres

```yaml
openapi: 3.0.0
info:
  version: 1.0.0
  title: Simple Example
  description: A simple API to illustrate OpenAPI concepts

servers:
  - url: https://dptinfo.univ-ubs.fr/

# Basic authentication
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
security:
  - BasicAuth: []

paths:
 /cars:
    get:
      description: Returns a list of cars
      parameters:
        - name: model
          in: query
          description: model of cars
          schema:
            type: string
```

```yaml
responses:
        '200':
          description: Successfully returned a list of cars
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    model:
                      type: string
                    matriculation:
                      type: string
                    year:
                      type: integer

        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
```

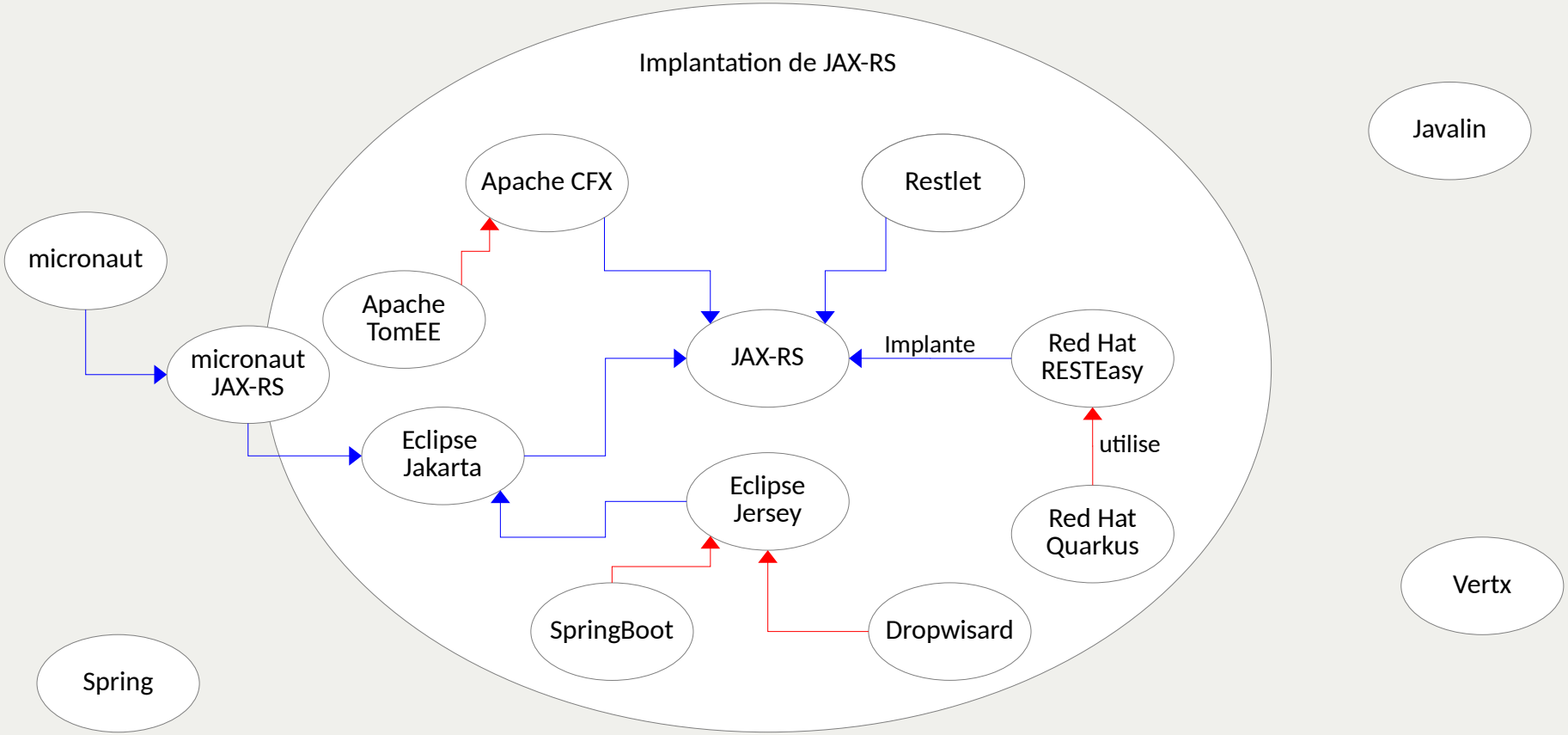
OpenAPI : exemple POST

```yaml
openapi: 3.0.0
info:
  version: 1.0.0
  title: Simple Example
  description: A simple API to illustrate OpenAPI concepts

servers:
  - url: https://dptinfo.univ-ubs.fr/

# Basic authentication
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
security:
  - BasicAuth: []

paths:
 /cars:
     post:
      description: Adds a new car
```

```yaml
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                model:
                  type: string
                matriculation:
                  type: string
                year:
                  type: integer

        responses:
        '200':
          description: Successfully added a new car

        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string
```

Panorama des technologies

Panorama des technologies Java



Panorama des technologies Javascript et Python

- Javascript
  - Express.js, Restify, Hapi, Sails, LoopBack, FeatherJS, KoaJS, Fastify, Moleculer
- Python
  - Flask, Falcon, Bottle, Nameko, CherryPy, FastAPi

Sécurisation d’une API REST

Pourquoi sécuriser les API ?

- Une API permet de rendre les fonctionnalités d'un service accessibles à d'autres services ou à des applications
- Il faut sécuriser une API pour éviter
  - la fuite/la divulgation de données
  - La corruption de données
- Il faut déterminer quels sont les services et les applications qui peuvent utiliser une API
- Pour sécuriser une API, il faut
  - utiliser une authentification et autorisation fortes
  - utiliser des jetons
  - appliquer le principe de moindre privilège
  - chiffrer et signer les échanges (avec TLS)
  - ne pas exposer plus de données que nécessaire
  - valider les données en entrée
  - limiter le trafic
  - utiliser un par-feu d'application Web capable de traiter les charges utiles des API.

Ressources publiques et privées

- Toutes les ressources exposées par une API ne nécessitent pas le même niveau de protection
- Il faut distinguer les ressources publiques des ressources privées
- Exemple :
  - « GET /products »: liste les produits en vente sur un site de commerce électronique (ressource publique)
  - « GET /users/id »: retourne les informations d'un utilisateur particulier (ressource privée)
- Application de la règle des 3-A (*Authentication*, *Authorization and Accountability*) aux ressources
  - Authentification : savoir qui accède à l'API
  - Autorisation : déterminer si l'accès à une ressource est autorisé à l'utilisateur authentifié, et si oui selon quelles modalités (lecture/GET, ajout/POST, mise-à-jour/UPDATE, suppression/DELETE)
  - Traçabilité : pour savoir qui à fait quoi, quand et avec quelle(s) ressource(s)
    - Fichier de logs

Comment sécuriser les API REST et SOAP

- Pour la délégation des accès, il existe une norme ouverte nommée OAuth (*Open Authorization*)
  - permet d'accorder à des tiers l'accès à des ressources web sans avoir à partager des mots de passe
  - OAuth2 : RFC 6749
- API REST
  - Les API REST reposent sur le protocole HTTP
  - TLS peut être utilisé pour chiffrer les échanges
- API SOAP
  - Les API SOAP utilisent un protocole de sécurité intégré : *Web Services Security*
    - Ajoute un mécanisme d'authentification et de confidentialité
  - Les API SOAP utilisent des normes définies par le W3C (*World Wide Web Consortium*) ou OASIS (*Organization for the Advancement of Structured Information Standards*)
    - Elles associent le chiffrement XML, les signatures XML et les jetons SAML pour vérifier l'authentification et les autorisations

Clé d'API

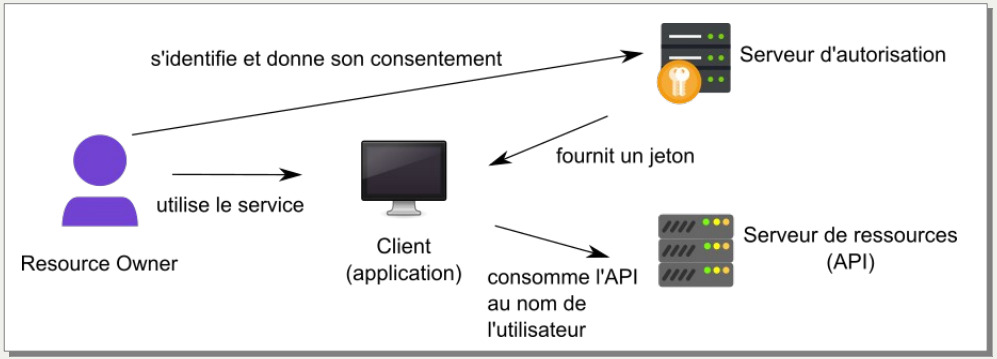
- Pas d'authentification et d'autorisation d'accès pour des ressources publiques, mais…
- … il peut être intéressant de gérer l'accès à l'API pour
  - éviter des abus d'utilisation
  - mettre en place des règles tarifaires sur la base de quotas d'utilisation
- Pour obtenir une clé d'API, un utilisateur doit généralement
  - s'inscrire sur un portail
  - obtenir une clé générée par le portail
  - utiliser cette clé dans les requêtes qu'il adresse à l'API
- Cette façon de faire à des limites :
  - identification de l'application et pas de l'utilisateur (utilisation de la même clé par plusieurs utilisateurs)
  - pas de date d'expiration de la clé
  - compromission de la clé, si la clé est stockée dans le code client de l'application

Les standards OAuth2 et OpenID Connect

- OAuth2 et OpenID Connect (OIDC) sont 2 standards industriels
- OAuth2 est dédié à la délégation de droits
- OIDC est dédié à la gestion des identités
  - Extension du protocole OAuth2
  - Permet d'utiliser un gestionnaire d'identités externe (e.g., Facebook, Google, Linkedin)

OAuth2

- OAuth2 est conçu pour autoriser des applications, ou services, à consommer des ressources fournies par un autre service au nom d'un utilisateur
- Exemple d'utilisation d'OAuth2:
  - Pour inscrire/authentifier un utilisateur, un site Web peut utiliser un réseau social connu ou un site Web connu (e.g. Facebook, Linkedin, Google)
    - Cette démarche permet à l'utilisateur de donner accès au site Web à ses informations personnelles déjà disponibles sur un réseau social ou sur un autre site Web.
- OAuth2 est défini par la RFC 6749



OAuth2 : Vocabulaire

- *Resource Owner* : propriétaire de la ressource (utilisateur)
  - qui utilise le client
  - qui est en mesure de donner accès à la ressource
- *Client* :application tierce qui demande accès à la ressource au nom du *Resource Owner*
- *Resource Server*: serveur qui expose les ressources consommées par le client
- *Autorization Server* : serveur
  - Qui délivre les droits d'accès aux ressources protégées au client après avoir authentifié le propriétaire de la ressource
  - qui fournit des jetons d'accès (*tokens*) aux clients
- *Scope* : une description de ce que l'utilisateur permet à l'application de faire en son nom
- *Consumer* : le développeur de l'application
- *Token* : suite de caractères qui relie un client, un ou plusieurs périmètres et éventuellement un utilisateur.

OAuth2 : Gestion des clients

- Une demande d'autorisation est toujours initiée par un client
- Les clients doivent être enregistrés auprès du serveur d'autorisation
- L'enregistrement d'un client nécessite au moins 3 informations
  - Identifiant du client
  - Le mot de passe, ou paire de clés (publique/privée) pour les clients de confiance
  - Une ou plusieurs URL(s) de redirection

OAuth2 : 3 types de clients

- Clients de confiance/clients confidentiels
  - Clients développés, déployés et exploités par l'organisation qui gère les ressources et le serveur d'autorisation
  - les identifiants du client sont transmis de manière sécurisée (client_id et client_secret)
  - conservent les identifiants secrets
  - les données utilisateur conservées par le client, le sont de manière sécurisée
  - ne conservent pas les identifiants des utilisateurs
- Clients externes
  - Clients utilisant d'anciennes spécifications OAuth0 et OAuth1
  - le serveur d'autorisation
    - peut faire confiance à ces clients externes pour transmettre les identifiants de manière sécurisée et les conserver secrets
    - ne peut pas faire confiance à ces clients pour mettre à jour rapidement les identifiants en cas de compromission
    - ne peut pas faire confiance à ces clients concernant la gestion des identifiants des utilisateurs

OAuth2 : 3 types de clients

- Clients publics
  - Clients auxquels on peut faire le moins confiance
  - Le serveur d'autorisation ne peut pas faire confiance à ces clients pour
    - transmettre de manière sécurisée les identifiants
    - conserver les identifiants secrets
    - pour mettre à jour rapidement les identifiants s'ils sont compromis
    - pour ne pas stocker des informations sur les utilisateurs

Les jetons

- La demande d'accès à une ressource protégée via Oauth se traduit par la délivrance d'un jeton au client
  - Chaîne de caractères unique permettant d'identifier le client et différentes informations utiles pour le processus d'autorisation
- 2 types de jetons : jetons d'accès et jetons de rafraîchissement
- Jetons d'accès
  - Permet au client d'accéder à une ressource protégée
  - Durée de validité limitée
  - Portée (scope) éventuellement limitée
    - Exemple : lecture seule
- Jetons de rafraîchissement
  - Permet au client d'obtenir un nouveau jeton d'accès après expiration du précédent
  - Durée de validité limitée
  - Obtention du jeton sans intervention de l'utilisateur

OAuth2 : Scénarios d'utilisation

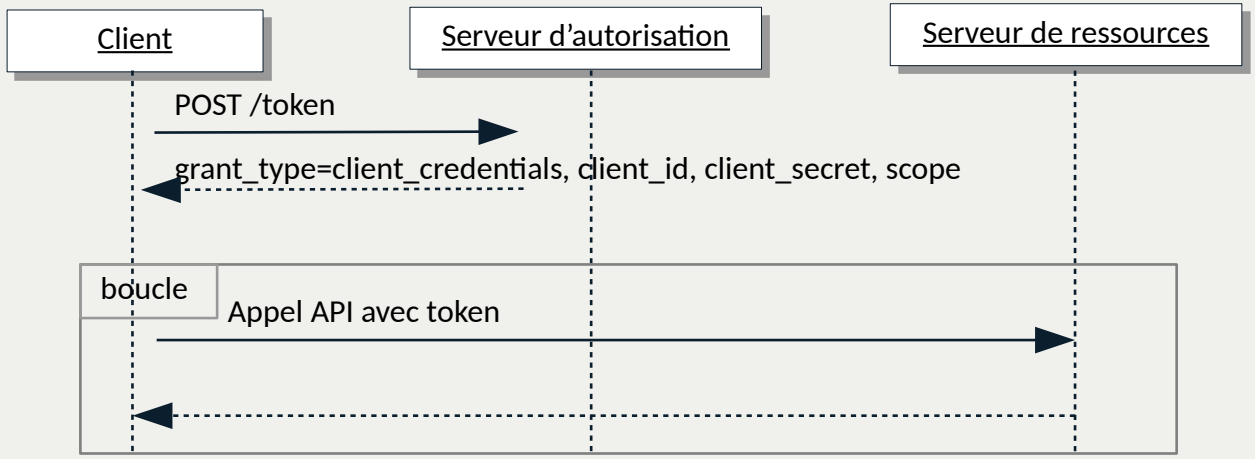
- OAuth2 définit 4 scénarios d'utilisation :
  - Autorisation avec un code (*Authorization Code Grant*)
  - Autorisation implicite (*Implicit Grant*)
  - Autorisation avec identifiant du propriétaire de la ressource (*Resource Owner Password Credentials Grant*)
  - Autorisation avec les identifiants du client (*Client Credentials Grant*)

Obtention des jetons : *Client Credentials Grant*

- Processus faisant intervenir deux acteurs uniquement : le client et le serveur d'autorisation
- Le propriétaire de la ressource n'est pas sollicité
  - Ne permet pas à un client d'effectuer des actions au nom d'un utilisateur
  - Ne permet pas en général d'accéder à des ressources protégées (sauf celles du client)
- Fournit un jeton à un client pour l'identifier auprès d'un serveur de ressources
- Utile pour des traitements effectués par des « utilisateurs techniques » (machines)
- Doit être utilisé par des clients de confiance ou externes
- Cas d'usage :
  - mode d'autorisation qui peut être utile si un serveur fournit une API publique mais veut ajouter une limitation d'usage par client

Obtention des jetons : *Client Credentials Grant*

- Principe de fonctionnement

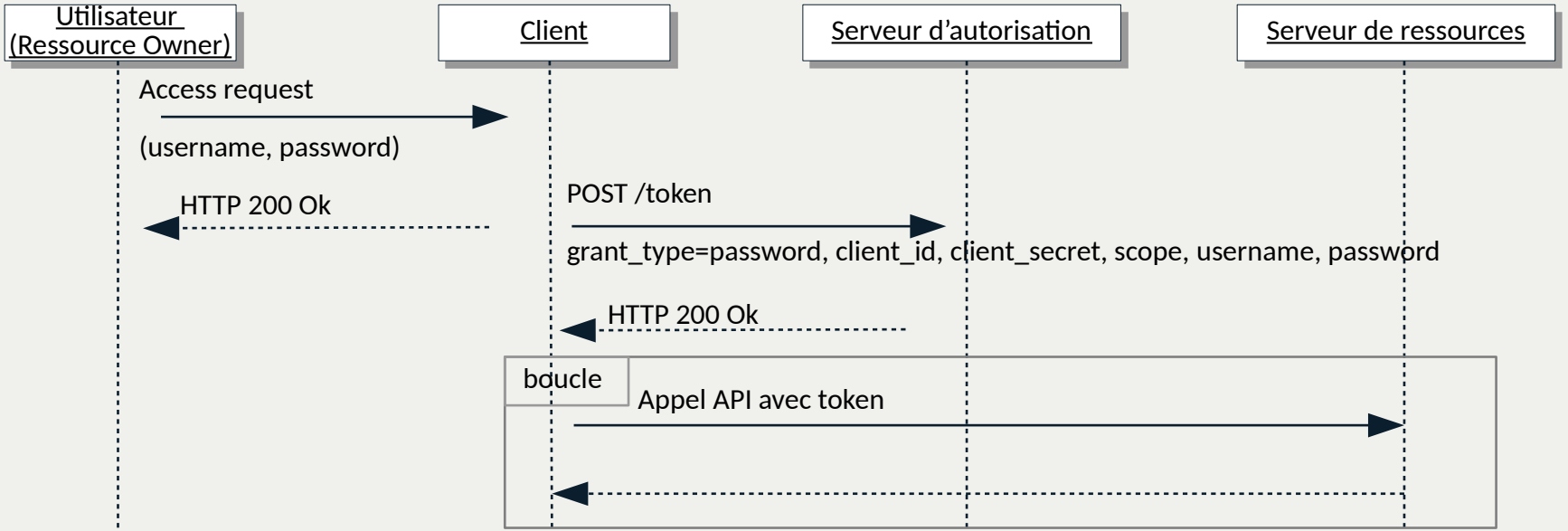


Obtention des jetons : *Resource Owner Credentials Grant*

- Processus qui ne nécessite pas de redirection du propriétaire de la ressource protégée vers le serveur d'autorisation
- Processus qui doit être utilisé uniquement pour les clients de confiance
  - Clients développés et gérés par l'organisme qui administre le serveur d'autorisation
  - Il doit y avoir une relation de confiance absolue entre ces deux acteurs car le propriétaire de la ressource doit fournir ses identifiants au client.
- Dans ce processus, le client
  - reçoit un jeton depuis un serveur d'autorisation
  - à la responsabilité de récupérer les identifiants de l'utilisateur

Obtention des jetons : *Resource Owner Credentials Grant*

- Principe de fonctionnement



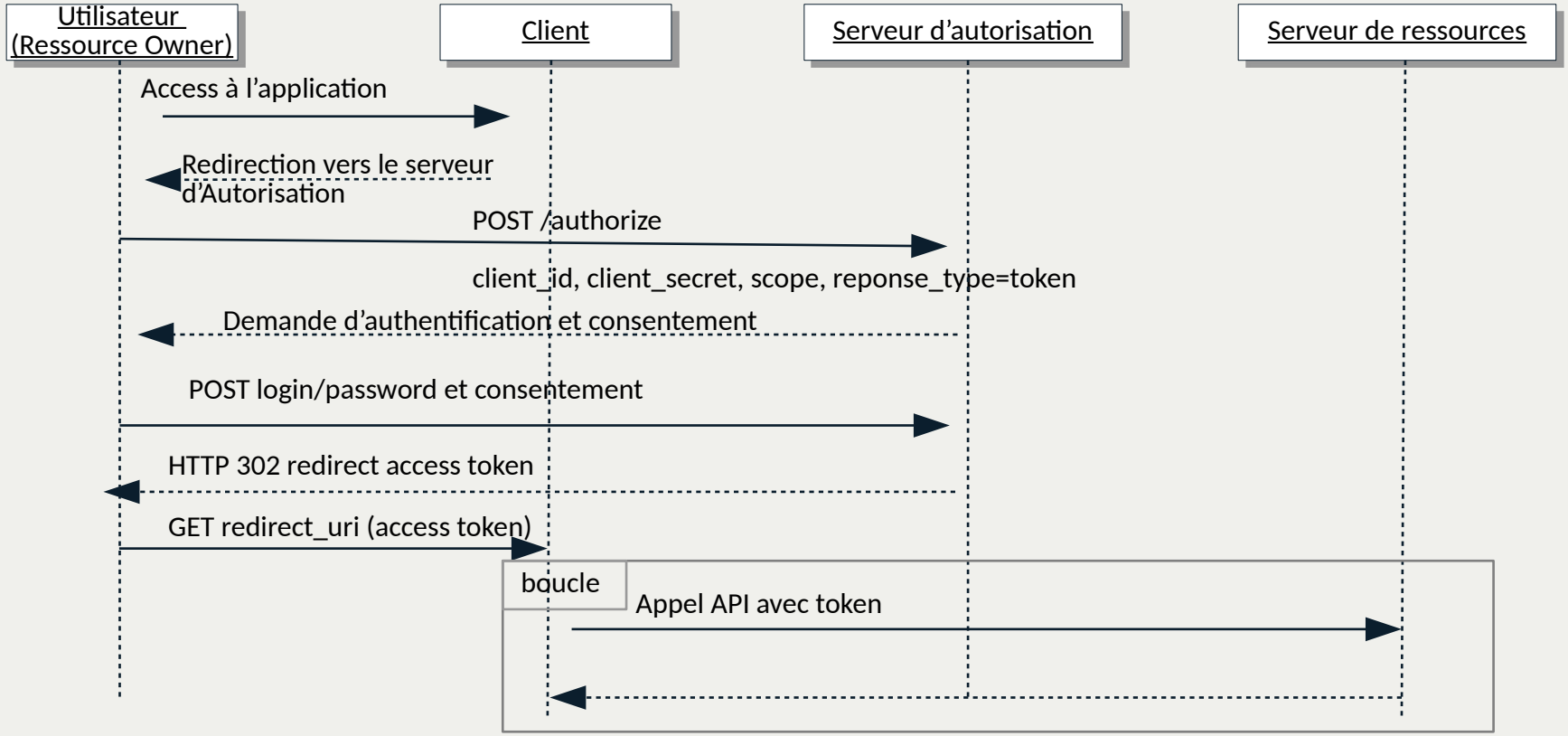

```
POST /token HTTP/1.1
Host: oauth2.myserver.com
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=user_1&password=pwd_user_1&client_id=id_1&client_secret=secret_1
```

Obtention des jetons : *Implicit Grant*

- Processus utile pour les applications publiques
- Processus faisant intervenir le client, le propriétaire de la ressource protégée et le serveur d'autorisation
- Le client étant publique, il ne peut pas s'authentifier de manière fiable.
- Le client ne peut fournir de mot de passe.
- Son identifiant et l'URL de redirection préenregistrée permettront de l'authentifier (d'où le nom d'autorisation implicite).
  - Quand le serveur d'autorisation redirige le propriétaire de la ressource vers une URL associée au client, il considère implicitement que le client a fait la demande d'autorisation et va traiter la réponse.
- Pas de code d'autorisation intermédiaire envoyé au client
- Jeton d'accès envoyé via une URL de confiance connue par le serveur d'autorisation (redirect URL)
- Processus adapté aux applications mobiles ou applications JavaScript exécutées sur le navigateur de l'utilisateur

Obtention des jetons : *Implicit Grant*

- Principe de fonctionnement

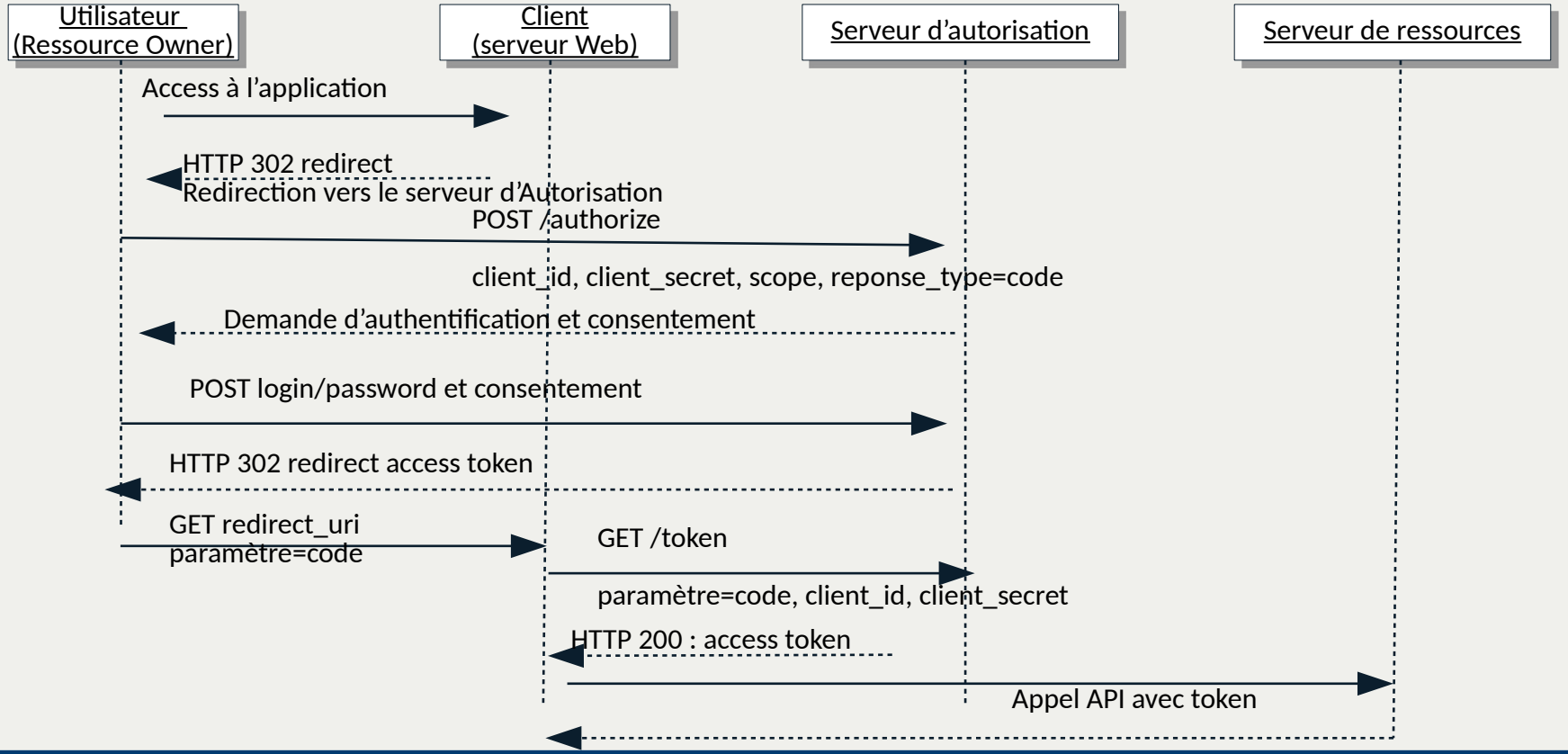


Obtention des jetons : *Authorization Code Grant*

- Processus utilisé principalement par les clients de confiance pour obtenir un jeton d'accès ou de rafraîchissement
  - Utilisable aussi par les clients externes
- Nécessite l'intervention du client, de propriétaire de la ressource et du serveur d'autorisation
- Utilisation adapté à un code exécuté côté serveur
  - L'accès au code serveur étant sécurisé, le client peut être considéré comme de confiance
  - Exemple : application PHP exploitant des services distants

Obtention des jetons : *Authorization Code Grant*

- Principe de fonctionnement

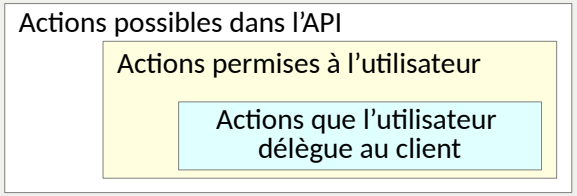


Utilisation d'un jeton

- Autoriser l'accès à une ressource
  - Quand un serveur de ressources reçoit un jeton dans les en-têtes d'une requête HTTP, il doit déterminer à quelle combinaison (client, utilisateur, périmètre) ce jeton correspond
    - Interrogation du serveur d'autorisation pour cette détermination
- Bloquer l'accès à une ressource
  - Si l'accès à la ressource nécessite un périmètre(lecture, écriture) qui n'est pas inclus dans le jeton d'accès, le serveur de ressources doit retourner une répons HTTP 403

Périmètres

- Un périmètre (scope) définit les limites des actions qu'octroie le propriétaire d'une ressource à un client
- Les périmètres peuvent être définis de différentes manières
  - Périmètres basés sur les actions
    - Repose sur le *pattern* ACL (*Access Control List*)
    - La chaîne de caractères peut être de la forme : ressource_rôle_action
      - Exemple : product_client_read
    - Appropriés pour configurer des droits utilisateurs, mais pas les droits des clients
  - Périmètres basés sur des rôles
    - Repose sur le *pattern* RBAC (*Role-Based Access Control*)
    - Exemple pour une application Web de type blog, on peut distinguer des rôles comme « auteur », « lecteur », « correcteur », etc.
- Périmètres et gestion des droits sont deux choses différentes



Consentement

- Le consentement c'est ce qu'autorise le propriétaire d'une ressource à faire le client en son nom
- Il existe 3 stratégies de consentement
  - Consentement implicite
    - Le serveur d'autorisation accorde toujours le périmètre demandé au client sans demandé confirmation à l'utilisateur
    - Doit être réservé aux clients de confiance
  - Consentement unique
    - On demande une seule fois le consentement à l'utilisateur (à la première utilisation)
    - C'est ce qui est utilisé par exemple par Google ou Facebook
  - Consentement systématique
    - On demande le consentement de l'utilisateur à chaque requête
    - Stratégie mise en place par exemple par FranceConnect

OpenID Connect : présentation

- OpenID Connect est une extension du protocole OAuth2 qui vise à permettre à une application cliente de récupérer de manière sécurisée l'identité de l'utilisateur, en se connectant au fournisseur d'identité
- OpenID Connect permet à une application cliente d'avoir une gestion des identités des utilisateurs sans avoir à mettre en place sa propre gestion des mots de passe
- OpenID Connect est lié à la notion de fédération d'identités (SSO : Single Sign-On)
- OpenID Connect n'est pas spécifié pour les scénarios
  - *Client Credentials Grants*
  - *Resource Owner Crendentials Grants*

OpenID Connect : principe de fonctionnement

- Le client demande au serveur d'autorisation un jeton avec un certain nombre de périmètres métiers, auquel vient s'ajouter un périmètre spécifique « openid »
- Le serveur d'autorisation génère un jeton (access_token), ainsi qu'un jeton d'identité (id_token)
  - Le jeton d'identité contient des détails sur l'identité de l'utilisateur et sur le processus utilisé pour l'authentifier
  - Le jeton d'identité prend la forme d'un JWT (*JSON Web Token*) signé
  - l'ID Token n'est à destination que du client, et ne devrait jamais être envoyé au serveur de ressources.
- La spécification OpenID connect définit un nouvel chemin (*endpoint*) « /userinfo » qui retourne l'identité de l'utilisateur