Java Enterprise Edition - Architectures et données

Gaël Guibon

E-mails: prenom.nom@lis-lab.fr

Aix*Marseille université

1. **Qu'est-ce que Java Entreprise Edition ?**
2. **Architecture JEE**
3. **Les données - beans**
4. **Les données - traitements**
5. **Accès aux données - JDBC**
6. **Architecture de Spring**
7. **JDBC Template**

Qu'est-ce que Java Entreprise Edition ?

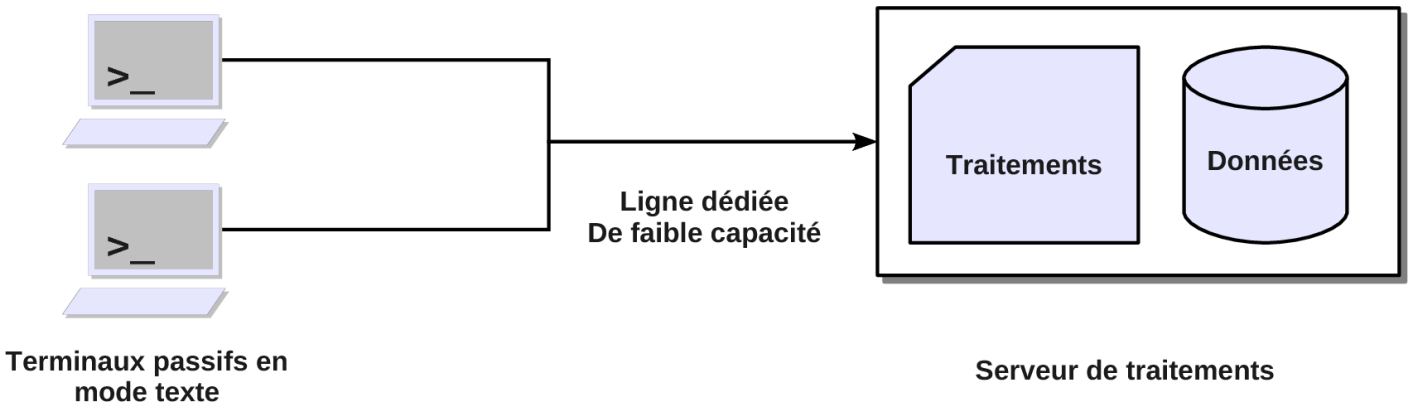
Java

- Java Standard Edition (JSE)
  - Applications classiques de bureau
- Java Micro Edition (JME)
  - Applications embarquées
- **Java Enterprise Edition (JEE)**
  - Applications d'entreprise connectées

JEE c'est...

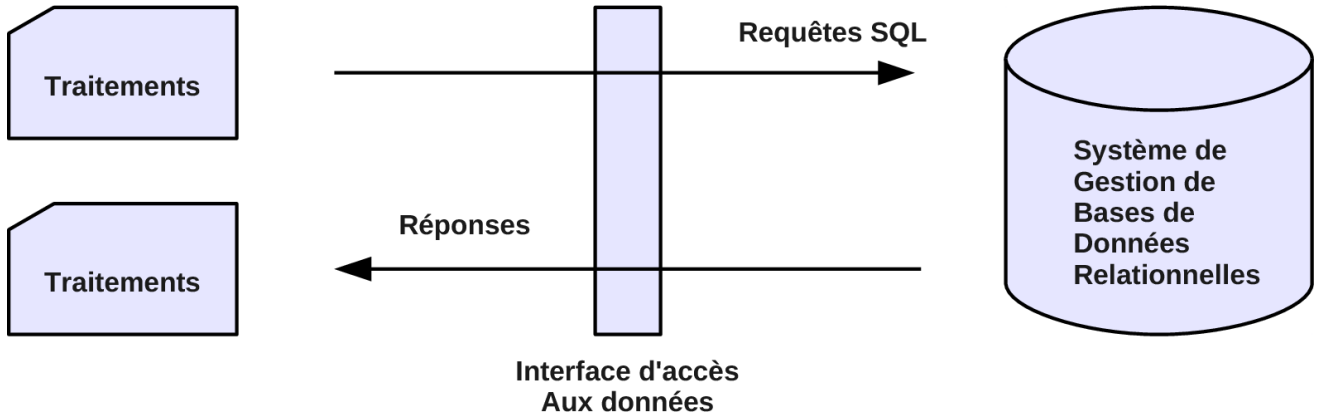
- Java
- Extension du Java Standard Edition (Java SE)
- Une architecture complexe ayant eu beaucoup d'évolutions
- Une architecture répondant à plusieurs besoins

Évolution des architectures - années 70



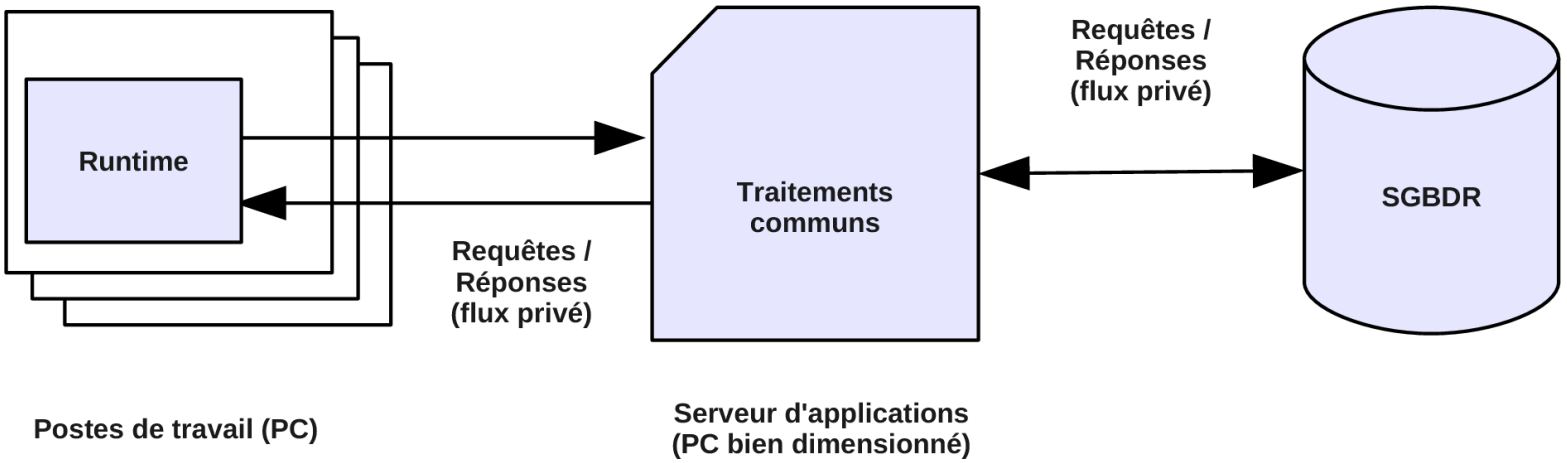
- Le service informatique gère des serveurs de traitements qui alimentent des terminaux.
- Les traitements et les données sont centralisés sur des serveurs d'infrastructure.
- Les données sont éventuellement centralisées sur des serveurs spécifiques (serveurs de fichiers).

Évolution des architectures - années 80



- L'interface d'accès aux données s'améliore (vues, trigger, etc)
- La gestion des données est la partie la plus importante.
- Méthodes de conception orientées données: Entités / Associations, Merise / Modèle Conceptuel des Données
- Création du métier d'administrateur des données (DBA).

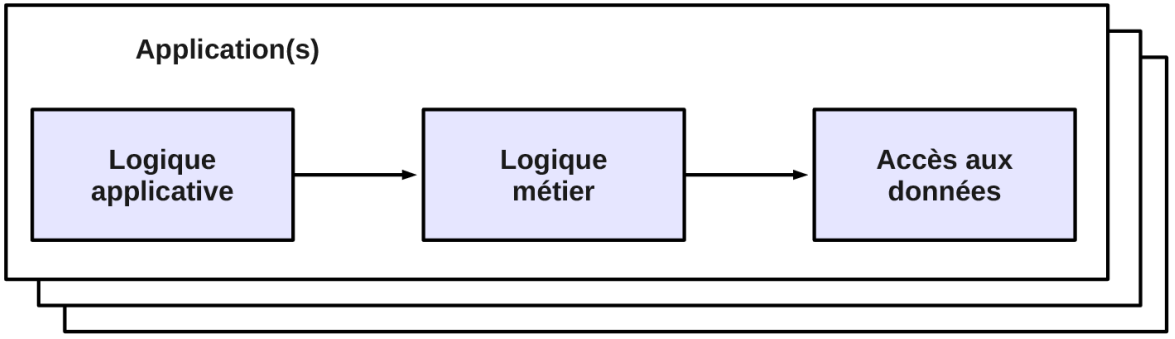
Évolution des architectures - années 90



- Serveur de taille moyenne (baisse du cout),
- Intégration des PC (hétérogènes) dans le SI.

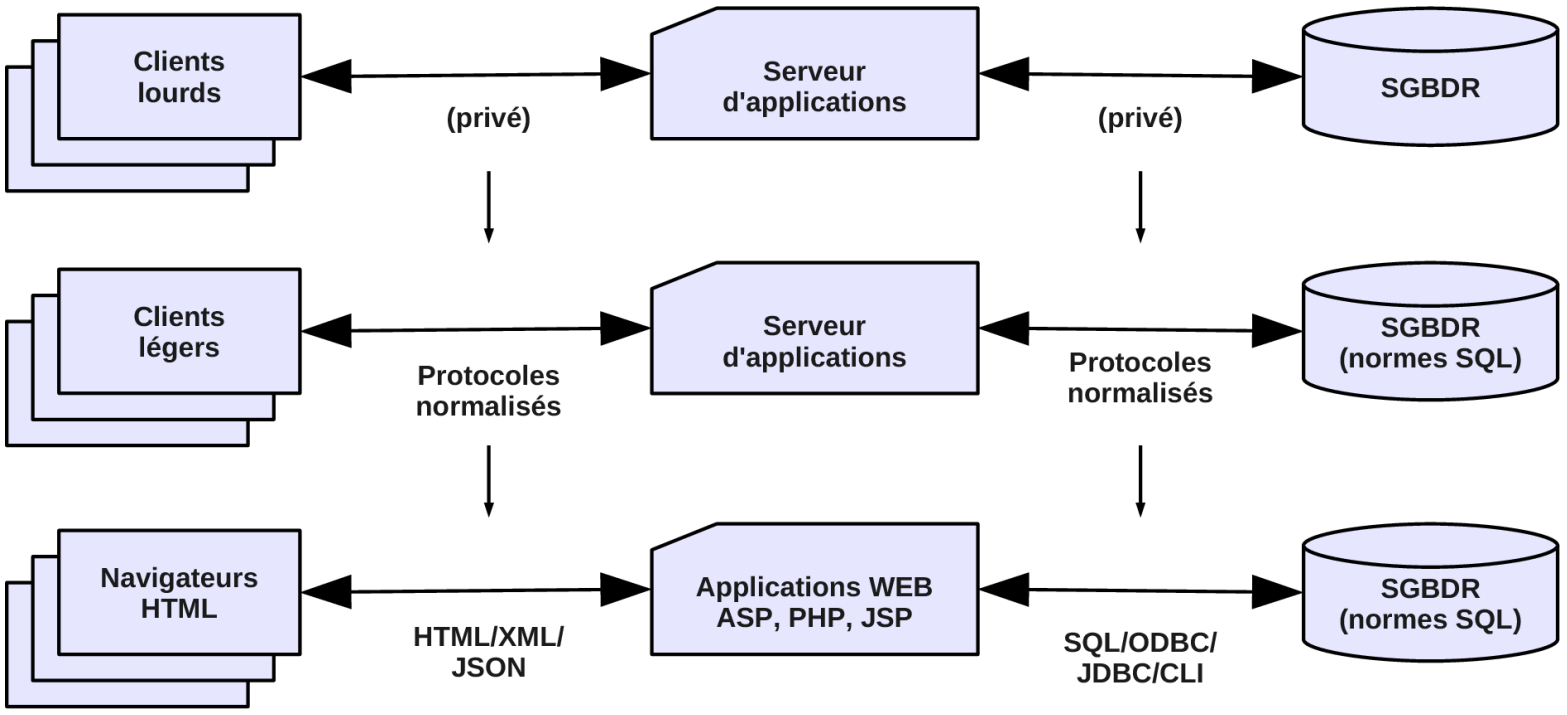
- Le poste de travail est qualifié de client lourd,
- Le coût d'exploitation est très important (dur à maintenir),

Évolution des architectures - années 2000

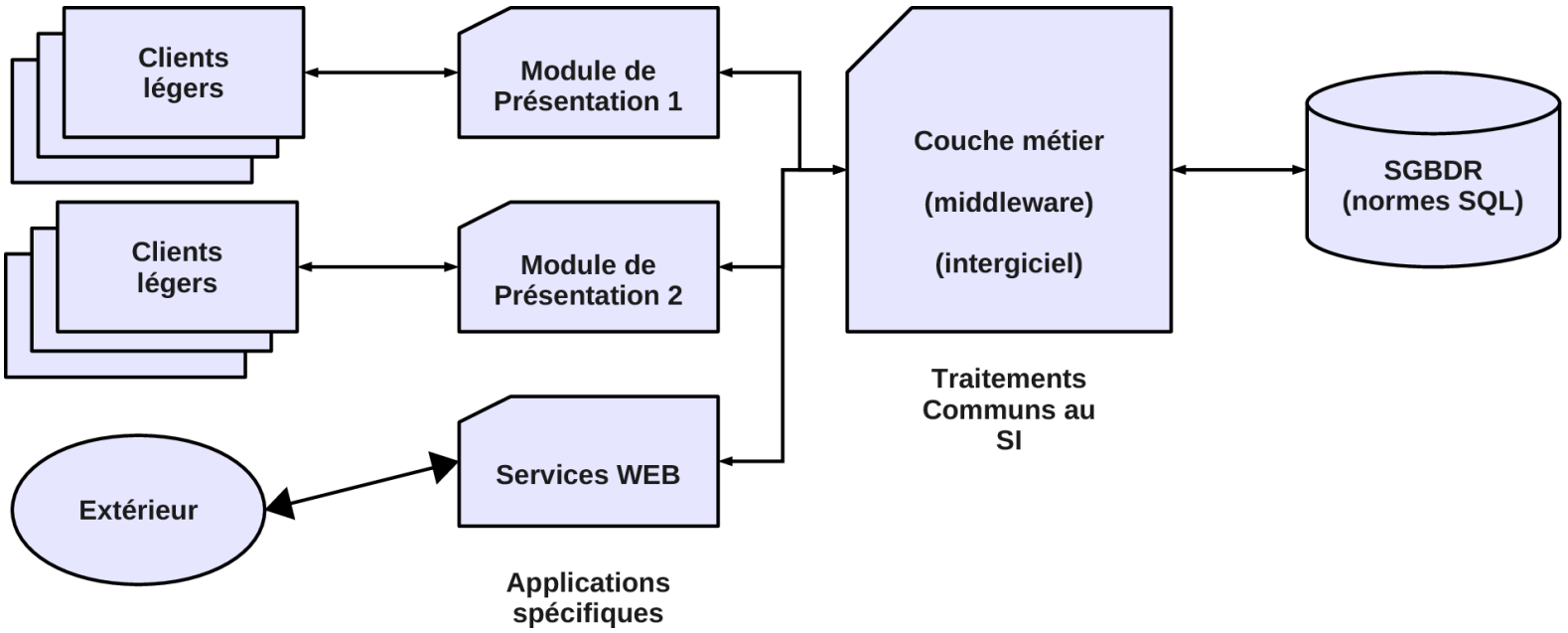


- Les parties métier et données sont dupliquées dans plusieurs applications,
- Il est très difficile de maintenir ces implantations (technologies, personnes, objectifs différents),
- La couche de stockage est le seul élément commun.

Évolution des architectures - années 2000



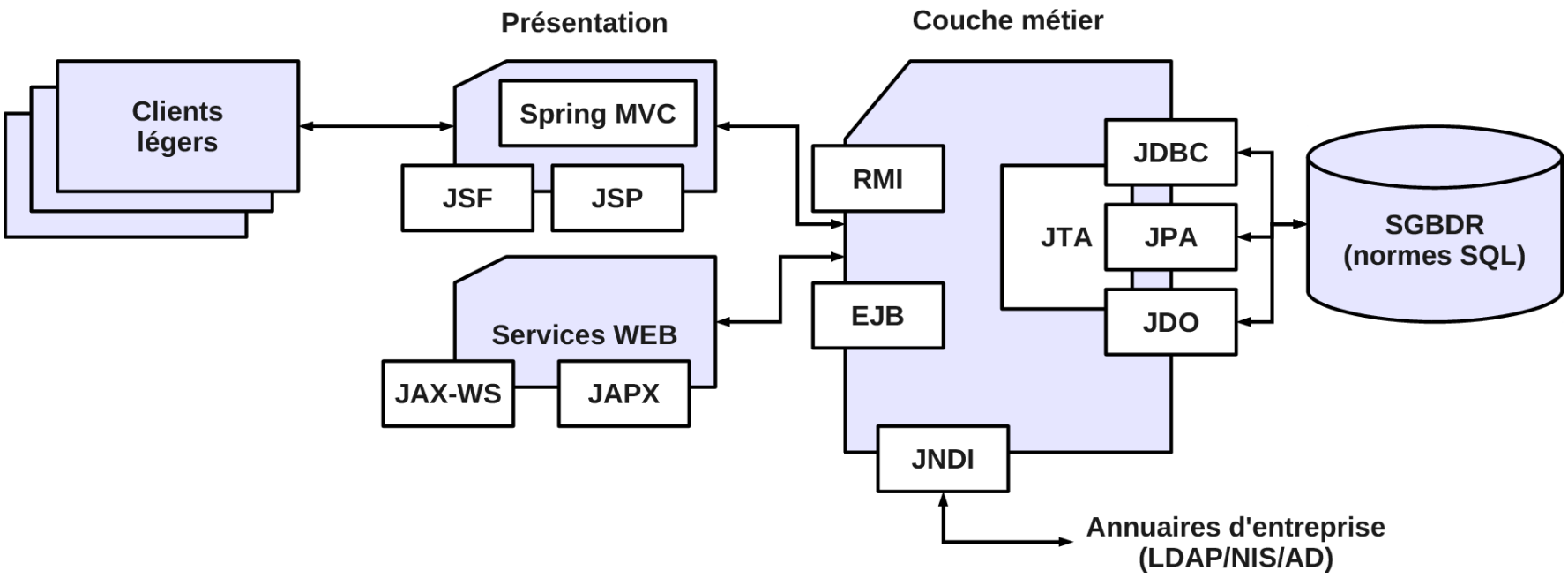
Architecture à trois niveaux



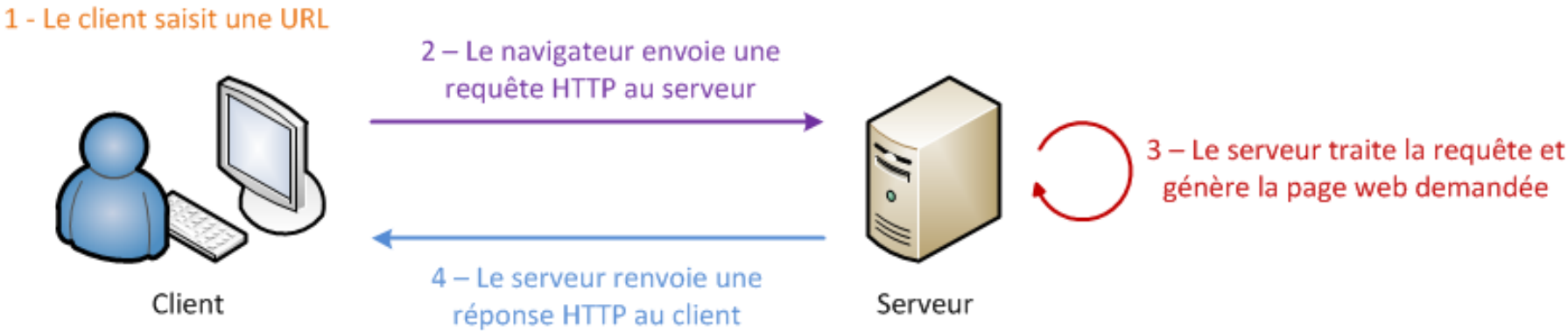
- Avantages : Simplifications des app, stockage indépendant, meilleure sécurité, ...
- Inconvénients : choix techno et couche de transport, middleware, empilements de couches

Architecture JEE

Architecture JEE



Flux de base d'une application web



Intérêt du JEE

Java	**JavaEE**
Socle de base	Extension de Java
Programmes locaux	Programmes connectés
Entrée par méthode main()	Entrées par chaque servlet
HTML	**JavaEE**
Site web statique	Site web dynamique
Visible	Opaque et sécurisé

Architecture JEE

Données

- Objectif : Représenter les données manipulées par l'application sans référence aux traitements
- UML / JavaBeans : classes de valeurs

Traitements

- Objectif : Effectuer les traitements sur les données, distribuer les données dans l'app
- Services d'accès aux donénes

Interface

- Objectif : Adapter l'interface, aiguiller l'utilisateur
- Contrôleurs, Vues

Architecture JEE - plan général

Dans ce cours..

- Données :
  - Beans
  - DAO
  - JDBC (template)
- Vues et contrôleurs :
  - Servlets
  - JSTL / EL
- Fiabilité :
  - Tests unitaires automatisés
- Spring MVC

Architecture JEE - plan général

Dans le cours suivant

- Données et métier : EJB, JPA
- Vues : Framework JSF, Jquery
- Modularité : Web Service REST

Architecture JEE

Ce qui nous intéresse aujourd'hui :

Données

- Beans
- Services
- Connexion à une Base de données (BDD)

Outils

- Eclipse IDE
- Maven

Les données - beans

Les beans

Stocker les données

- Réutilisable ! (objet java)
- Persistant ! (serialization)
- Paramétrable ! (propriétés)

Respecte plusieurs règles

- C'est une classe publique.
- Implémente *Serializable* pour pouvoir être sauvegardé.
- AUCUN champ public ! (getter et setters)
- Possède un constructeur par défaut public

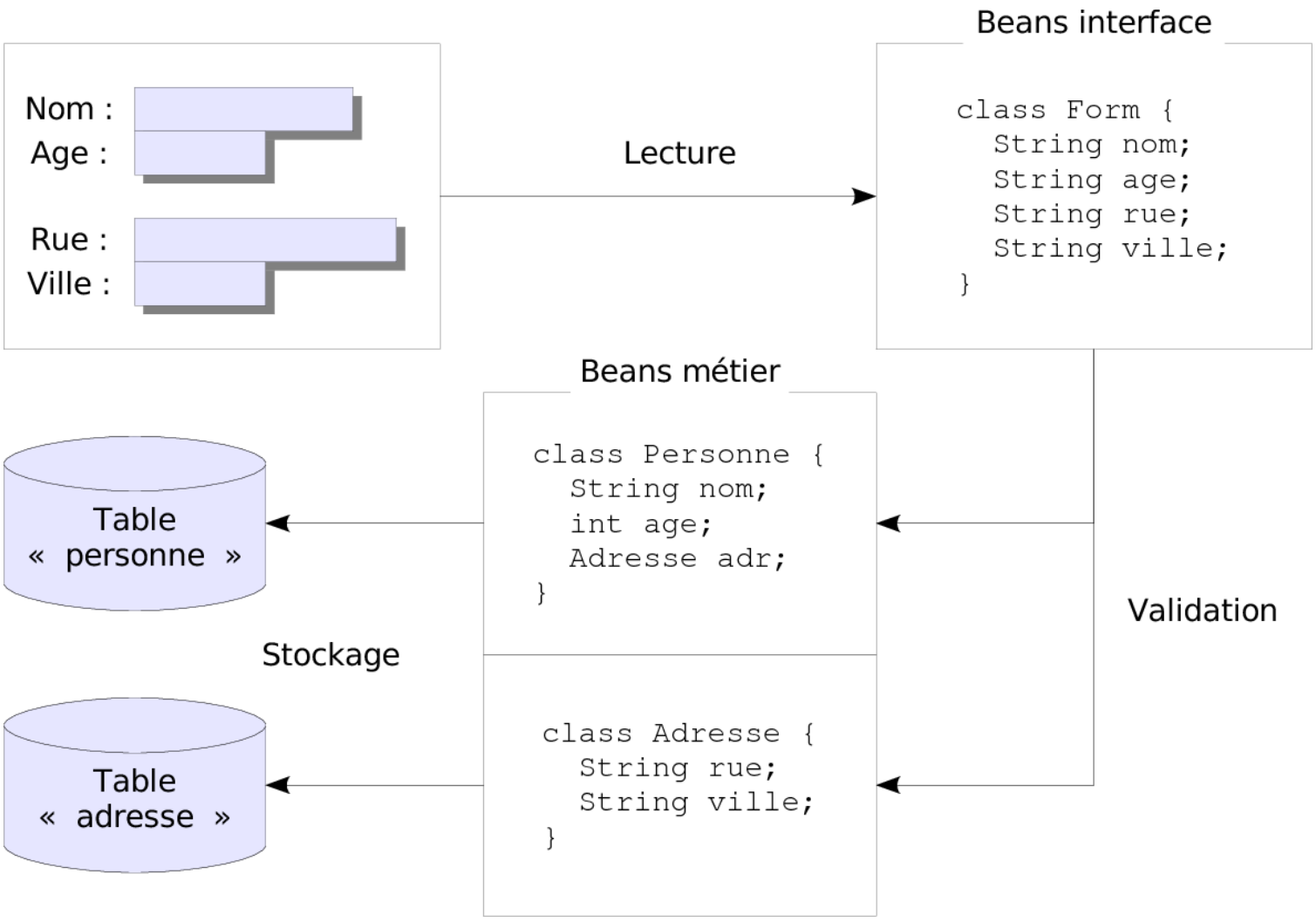
Les beans

```
public class Person implements Serializable {
private static final long serialVersionUID = 1L;
// properties
private String name;
private boolean student;
private Set<Person> friends;
// constructor
public Person() { }
// getters
public String getName() { return name; }
public boolean isStudent() { return student; }
public Set<Person> getFriends() { return friends; }
// setters
public void setName(String name) { this.name = name; }
public void setStudent(boolean student) { this.student = student;
    }
public void setFriends(Set<Person> friends) { this.friends =
    friends; }
}
```

Les beans

- Pour les collections, les interfaces sont à privilégier.
- L'accès aux propriétés passe obligatoirement par les méthodes (l'implantation est donc forcément privée).
- Les méthodes fixent le nom des propriétés (l'anglais est à favoriser).
- Il n'est pas nécessaire de maintenir la validité des données représentées.
- Les environnements de développement offrent toujours des facilités pour rédiger les JavaBeans.

Les beans


Beans interface
Nom :
Age :
Rue :
Ville :
Lecture
class Form {
String nom;
String age;
String rue;
String ville;
}
Beans métier
class Personne {
String nom;
int age;
Adresse adr;
}
Table
« personne »
Stockage
Validation
class Adresse {
String rue;
String ville;
}
Table
« adresse »

Les données - traitements

Les traitements

Comment implanter une méthode de sauvegarde ?

```
public class Person {
public void save() { ... }
}
```

Informations sur le système de persistance

- Nature des informations ? (BDR, XML, etc.)
- Paramètres ? (login, mot de passe, etc.)

Impossible dans un bean

- Ce ne sont pas des données de l'application,
- Elles seraient dupliquées dans chaque instance,

Les traitements

Astuce possible...

```
public class Person {
public void save(JDBCParameters where) { ... }
}
```

Mais...

- `where` est **obligatoire** car il indique où effectuer la sauvegarde (classe JDBCParameters)
- Bean pollué : il n'est censé uniquement représenter des valeurs
- Persistance dupliquée dans chaque bean
- Dépendance artificielle donnée-manipulation

Les traitements

Solution

Une classe personnalisée

```
public class JDBCStorage {
...
public void save(Person p) {
   ...
}
...
}
```

Plusieurs versions, une seule interface.

```
public class JDBCStorage implements Storage {
   ...
 }
 public class FileStorage implements Storage {
   ...
 }
```

Les services

Les traitements passent par le biais de services.

Un service

- Une spécification (*0,n* interfaces)
  - Visible par l'utilisateur
- Une ou plusieurs implantations (*0,n* classes de service qui agissent sur les données
  - Invisible pour l'utilisateur

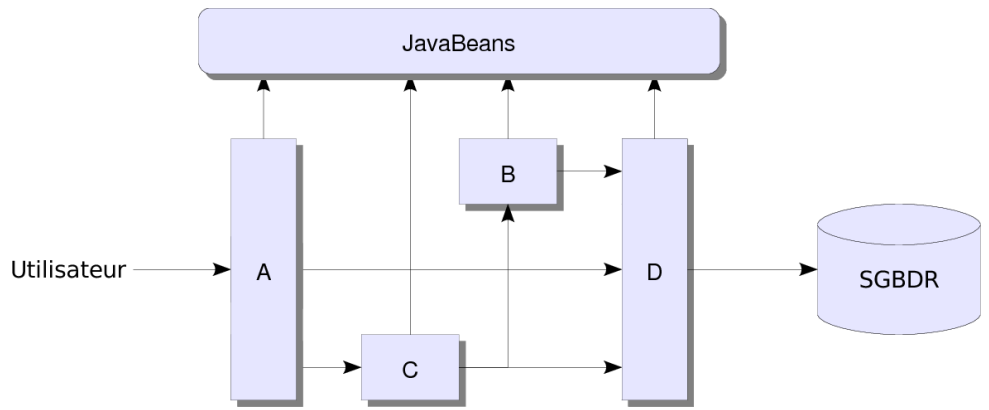
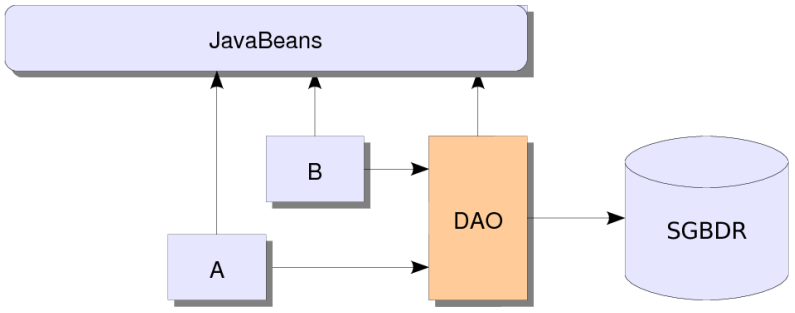


Figure: Une app == un ensemble de services

Types de services



Accès aux données - DAO

- centralisation des accès aux données,
- simplification de l'accès aux données,
- abstraction du support de stockage,
- travail sur les entités principales,

Types de services - DAO

Data Access Object (DAO)

Interface

```
// Abstract class DAO Factory
public abstract class DAOFactory {

public static final int CLOUDSCAPE = 1;
public static final int ORACLE = 2;
public static final int SYBASE = 3;
...
public abstract CustomerDAO getCustomerDAO();
public abstract AccountDAO getAccountDAO();
public abstract OrderDAO getOrderDAO();
...
public static DAOFactory getDAOFactory(
    int whichFactory) {
  switch (whichFactory) {
    case CLOUDSCAPE:
        return new CloudscapeDAOFactory();
    case ORACLE   :
        return new OracleDAOFactory();
    case SYBASE   :
        return new SybaseDAOFactory();
     ...
    default              :
        return null;
```

Types de services - DAO

Implémentation

```
import java.sql.*;

public class CloudscapeDAOFactory extends DAOFactory {
  public static final String DRIVER=
    "COM.cloudscape.core.RmiJdbcDriver";
  public static final String DBURL=
    "jdbc:cloudscape:rmi://localhost:1099/CoreJ2EEDB";
  // method to create Cloudscape connections
  public static Connection createConnection() {
    // Use DRIVER and DBURL to create a connection
    // Recommend connection pool implementation/usage
  }
  public CustomerDAO getCustomerDAO() {
    // CloudscapeCustomerDAO implements CustomerDAO
    return new CloudscapeCustomerDAO();
  }
  public AccountDAO getAccountDAO() {
    // CloudscapeAccountDAO implements AccountDAO
    return new CloudscapeAccountDAO();
  }
  public OrderDAO getOrderDAO() {
    // CloudscapeOrderDAO implements OrderDAO
    return new CloudscapeOrderDAO();
  }
  ...
```

Types de services - DAO

Meilleure pratique pour chaque bean :

- 1 Bean
- 1 DAO (interface)
- 1 DAO (implémentation)

Donc :

- PersonBean.java (nom, prenom, ... setters getters)
- PersonDao.java (addPerson(), changeName(), etc)
- PersonDaoImpl.java (addPerson() avec connexion SQL etc.)

Types de services - DAO

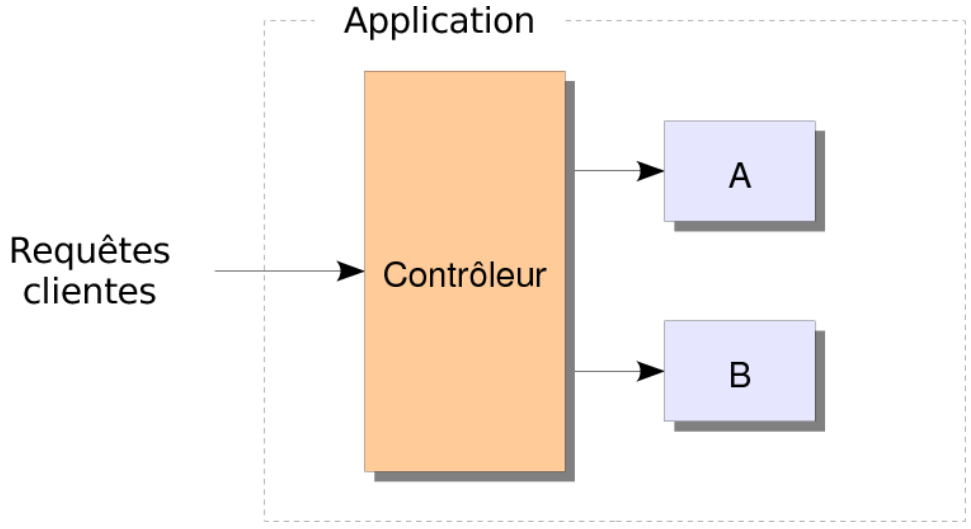
Meilleure pratique pour chaque bean :

- 1 Bean
- 1 DAO (interface)
- 1 DAO (implémentation)

Donc :

- PersonBean.java (nom, prenom, ... setters getters)
- PersonDao.java (addPerson(), changeName(), etc)
- PersonDaoImpl.java (addPerson() avec connexion SQL etc.)

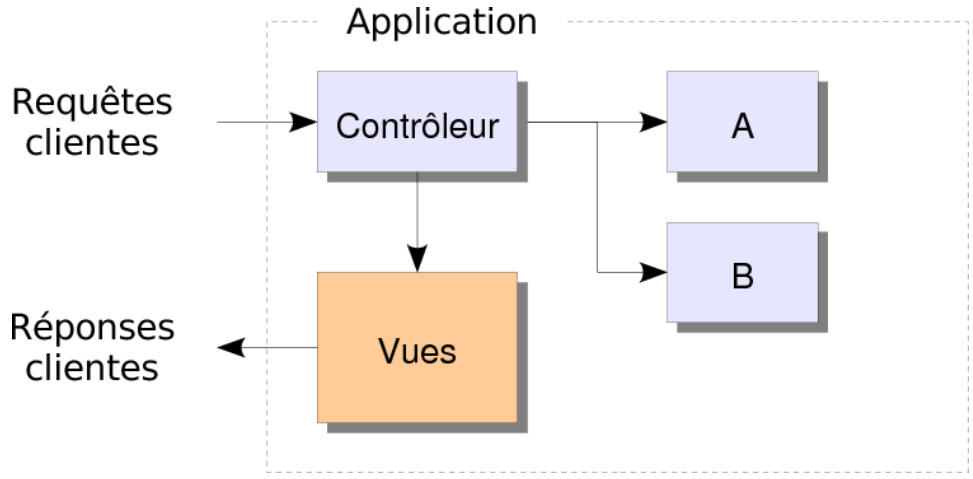
Types de services



Contrôleurs

- l'implantation du protocole d'entrée,
- le traitement et la validation des requêtes clientes,
- l'appel aux couches internes.

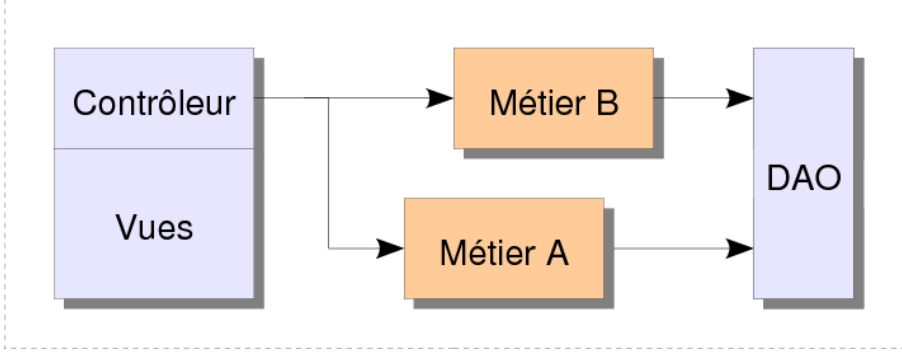
Types de services



Vues

- l'implantation du protocole de sortie,
- la construction des résultats à partir des données,
- l'envoi de ces résultats.

Types de services



Services métiers

- Ils sont indépendants d'une source de données.
- Ils sont indépendants de la logique applicative (suite de requêtes clientes).
- Ils sont réutilisables.

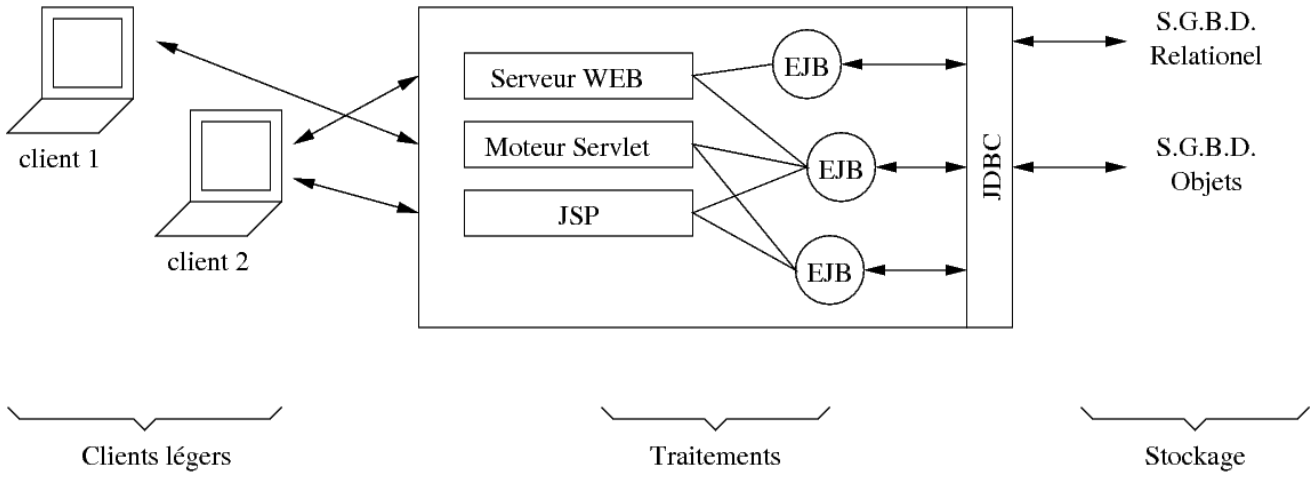
Types de services

Rôles possibles

- offrir des fonctions spécialisées (DAO, contrôleur, métier, etc.),
- simplifier un service trop complexe (facade),
- enrichir les fonctions d'un service existant (decorator),
- rechercher un service (locator),
- se charger de l'accès à un service (proxy),
- construire un service particulier (factory).

Accès aux données - JDBC

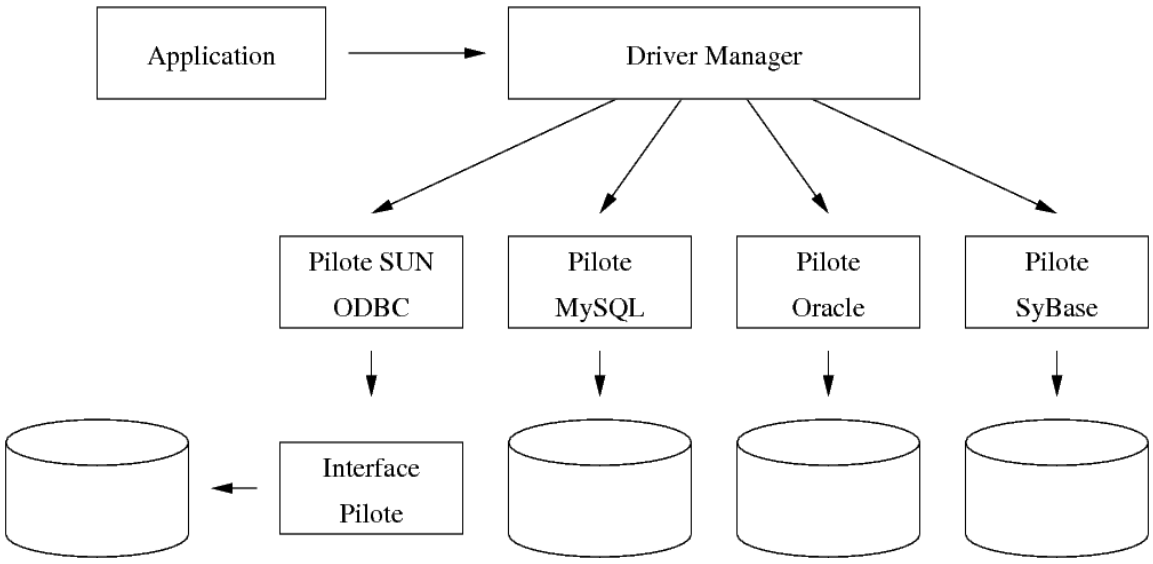
JDBC



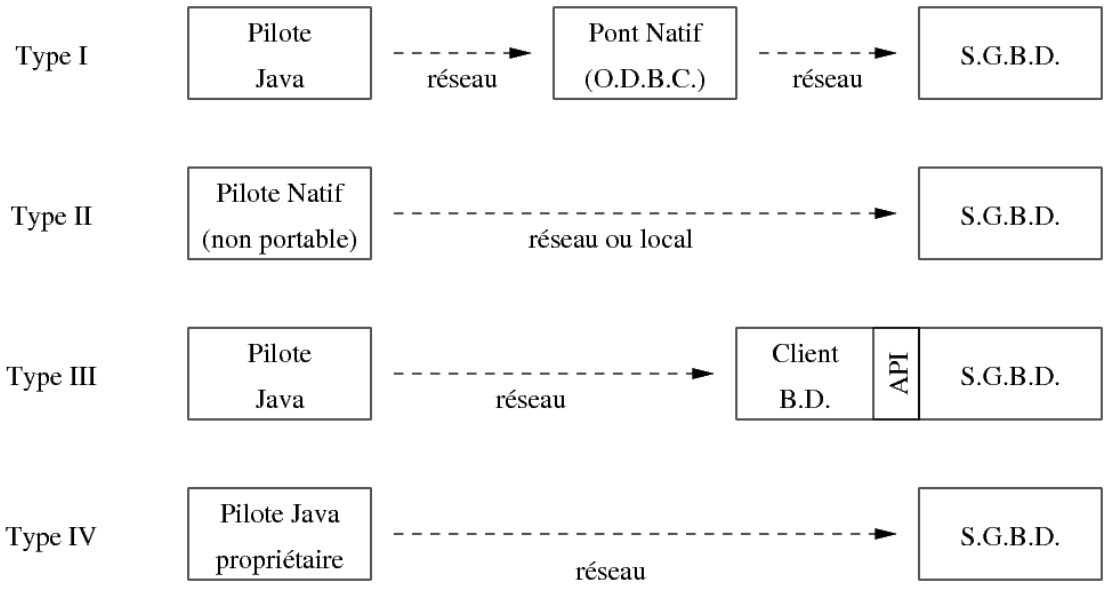
Utilité

- Lien entre les traitements et l'accès aux données
- Permet de requêter directement les bases de données

Architecture JDBC



Pilotes JDBC



- Plusieurs pilotes selon les besoins
- Le pilote doit être déclaré dans le code

Code JDBC

Listing 1: Squelette

```
import java.sql.DriverManager; // gestion des pilotes
import java.sql.Connection;  // une connexion àla BD
import java.sql.Statement;   // une instruction
import java.sql.ResultSet;   // un résultat (lignes/colonnes)
import java.sql.SQLException; // une erreur
public class JdbcSample {
    // chargement du pilote
    // ouverture de connexion
    // exécution d'une requête
    // programme principal
}
```

Code JDBC

Listing 2: Chargement du pilote

```
private String driverName = "com.mysql.jdbc.Driver";

void loadDriver() throws ClassNotFoundException {
    Class.forName(driverName);
}
```

Code JDBC

Listing 3: Connexion à la BD

```
private String url    = "jdbc:mysql://localhost/dbessai";
private String user   = "bduser";
private String password = "SECRET";

Connection newConnection() throws SQLException {
    Connection conn = DriverManager.getConnection(url, user,
        password);
    return conn;
}
```

Code JDBC

Listing 4: Requête

```
final String PERSONNES = "SELECT nom,prenom,age FROM personne ORDER BY age";

public void listPersons () throws SQLException {
    Connection conn = null;
    try {
        // create new connection and statement
        conn = newConnection();
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery(PERSONNES);

        while (rs.next()) {
            System.out.printf("%-20s | %-20s | %3d\n", rs.getString(1), rs.getString("prenom"),
                    rs.getInt(3));
        }
    } finally {
        // close result, statement and connection
        if (conn != null) conn.close();
    }
}
```

Nécessaire d'itérer sur les ResultSet

Code JDBC

Listing 5: Programme principal

```
public static void main(String[] Args) {
    JdbcSample test = new JdbcSample();
    try {
        test.loadDriver();
        test.listPersons();
        ...
    } catch (ClassNotFoundException e) {
        System.err.println("Pilote JDBC introuvable !");
    } catch (SQLException e) {
        System.out.println("SQLException: " + e.getMessage());
        System.out.println("SQLState:  " + e.getSQLState());
        System.out.println("VendorError: " + e.getErrorCode());
        e.printStackTrace();
    }
}
```

Résumé JDBC

Avantages

- Gérer en détail la connexion avec la BDD
- Propre à JavaEE
- Intégration de plusieurs pilotes et structures de BDD

Inconvénients

- Connexion, Statement, Request, Close, ErrorHandling.... à chaque fois = fastidieux !
- La gestion des erreurs rend le code brouillon.

Solution

Spring JDBC Template

Architecture de Spring

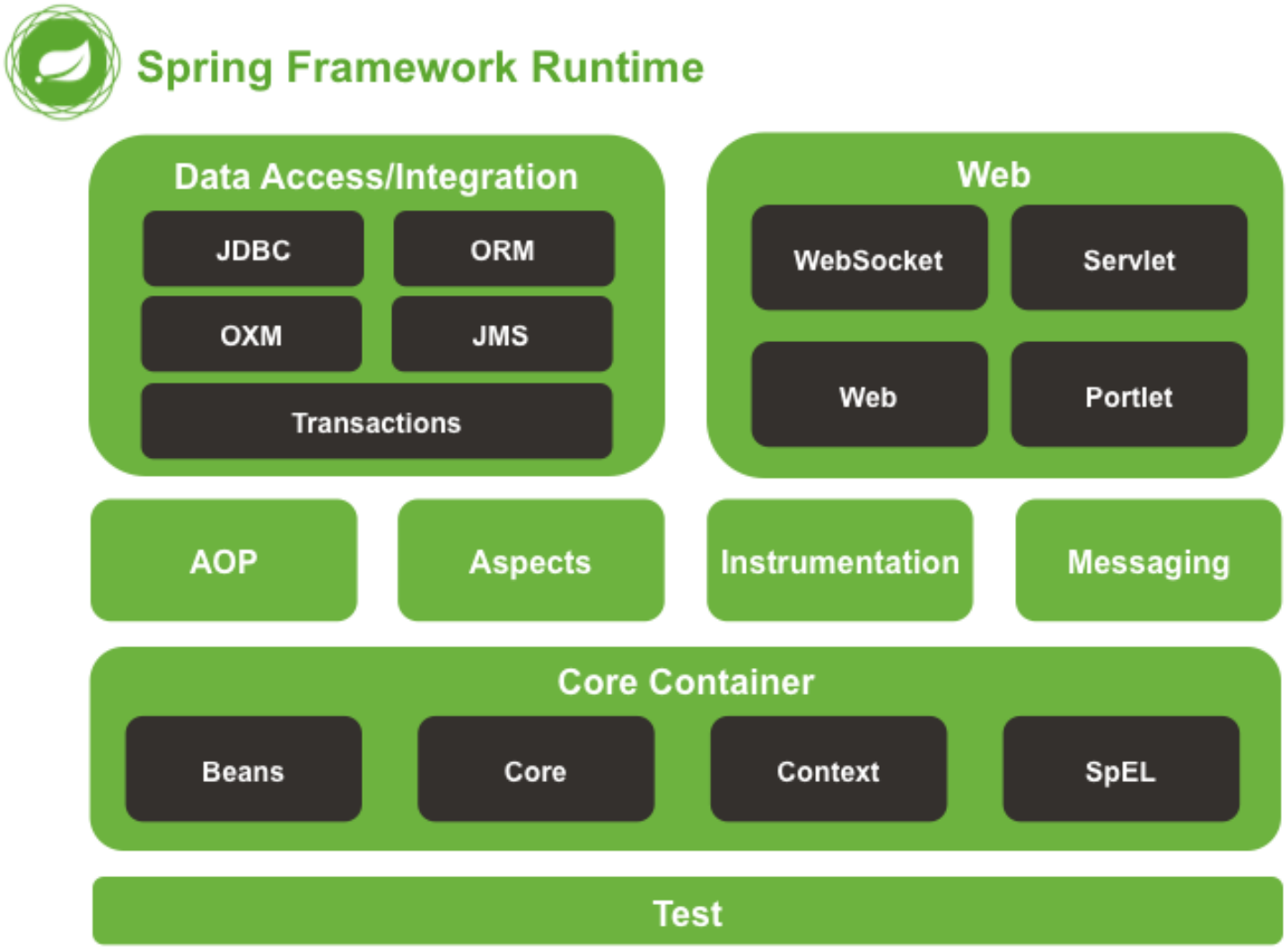
Spring Framework

Qu'est-ce que c'est ?

- Une couche logicielle faite d'interfaces et implémantations
- Simplifie la réalisation d'application ainsi que le code (moins *bloated*)
- Organise le code
- Réduit les dépendances
- Facilite les tests

Dans ce cours nous allons utiliser Spring pour simplifier la réalisation, et Maven pour simplifier la compilation et le déploiment.

Architecture de Spring



Architecture de Spring

- Gestion des instances de classes (JavaBean et/ou métier)
- Programmation orientée Aspect (AOP)
- Modèle MVC et outils pour les applications WEB
- **DAO (JDBC)**
- ORM (Hibernate, iBatis, ...)
- Outils pour les applications JEE (JMX, JMA, JCA, EJB, ...)

JDBC Template

Facilite l'accès aux requêtes SQL

Définir la DataSource
Ouvrir la connexion
Spécifier la requête SQL
Déclarer les paramètres et fournir leur valeur
Préparer et exécuter la requête (Statement)
Boucler, si besoin, sur les résultats (ResultSet)
Faire ce qu'il y a à faire à chaque itération
Fermer la connexion, le Statement et le ResultSet
Prendre en charge les transactions
Traiter les exceptions

Table: Source : openclassroom

JDBC Template

Avantages

- Nécessite une dépendance

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-jdbc</artifactId>
</dependency>
```

- Permet de créer des *templates* de requêtes avec paramètres
- Une seule classe de gestion des requêtes
  - Plus simple à mettre en place, vérifier et tester

JDBC Template

Méthode avec

- Requête SQL native
- Paramètres SQL en Java
- Retour de la valeur et de son type

JDBC Template

Listing 6: Exemple de template

```
public class TicketDaoImpl extends AbstractDaoImpl implements
    TicketDao {
  @Override
  public int getCountTicket(RechercheTicket pRechercheTicket) {
      String vSQL
          = "SELECT COUNT(*) FROM ticket"
          + " WHERE auteur_id = :auteur_id"
          + "   AND projet_id = :projet_id";
      NamedParameterJdbcTemplate vJdbcTemplate = new
           NamedParameterJdbcTemplate(getDataSource());
      MapSqlParameterSource vParams = new MapSqlParameterSource();
      vParams.addValue("auteur_id", pRechercheTicket.getAuteurId());
      vParams.addValue("projet_id", pRechercheTicket.getProjetId());
      int vNbrTicket = vJdbcTemplate.queryForObject(vSQL, vParams,
           Integer.class);
      return vNbrTicket;
  }
}
```

JDBC Template

Listing 7: Rows

```
@Named
public class TicketDaoImpl extends AbstractDaoImpl implements
     TicketDao {
    // ...
    @Override
    public List<TicketStatut> getListStatut() {
        String vSQL = "SELECT * FROM public.statut";
        JdbcTemplate vJdbcTemplate = new JdbcTemplate(getDataSource());
        RowMapper<TicketStatut> vRowMapper = new
             RowMapper<TicketStatut>() {
            public TicketStatut mapRow(ResultSet pRS, int pRowNum)
                 throws SQLException {
                TicketStatut vTicketStatut = new
                     TicketStatut(pRS.getInt("id"));
                vTicketStatut.setLibelle(pRS.getString("libelle"));
                return vTicketStatut;
            }
        };
        List<TicketStatut> vListStatut = vJdbcTemplate.query(vSQL
```

JDBC Template

Listing 8: Update

```
public void updateTicketStatut(TicketStatut pTicketStatut) {
    String vSQL = "UPDATE statut SET libelle = :libelle WHERE id =
        :id";
    MapSqlParameterSource vParams = new MapSqlParameterSource();
    vParams.addValue("id", pTicketStatut.getId());
    vParams.addValue("libelle", pTicketStatut.getLibelle());
    NamedParameterJdbcTemplate vJdbcTemplate = new
        NamedParameterJdbcTemplate(getDataSource());
    int vNbrLigneMaJ = vJdbcTemplate.update(vSQL, vParams);
}
```

Sources

- Documentation Oracle
- Cours JEE année précédente
- Cours de J.L Massat
- Cours OpenClassroom