Spring Boot : Spring JDBC

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Spring Boot

JDBC

- **API** de **JSE**
- Permettant la connexion et l'exécution de requêtes **SQL** depuis un programme **Java**
- Composé de
  - `Driver`
  - `DriverManager`
  - `Connection`
  - `Statement`
  - `ResultSet`
  - `SQLException`
  - ...

Spring Boot

JDBC : avantages

- Multi-base de données
- Support pour les requêtes et les procédures stockées
- Fonctionnant en synchrone et asynchrone
- Pas besoin de convertir les données

JDBC : inconvénients

- Pas de driver universel
- Trop verbeux
- Code souvent redondant
- Complexe

Spring Boot

Spring JDBC

- Module de **Spring Framework**
- Proposé pour éliminer une bonne partie des inconvénients de l'**API JDBC** de **JSE**
- Composé de
  - `JdbcTemplate`
  - `NamedParameterJdbcTemplate`
  - `SimpleJdbcTemplate`
  - ...

Spring Boot

Spring JDBC

Spring Boot

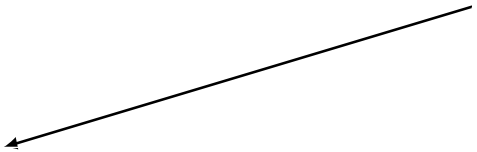
Spring JDBC

Plusieurs solutions

Spring Boot

Spring JDBC

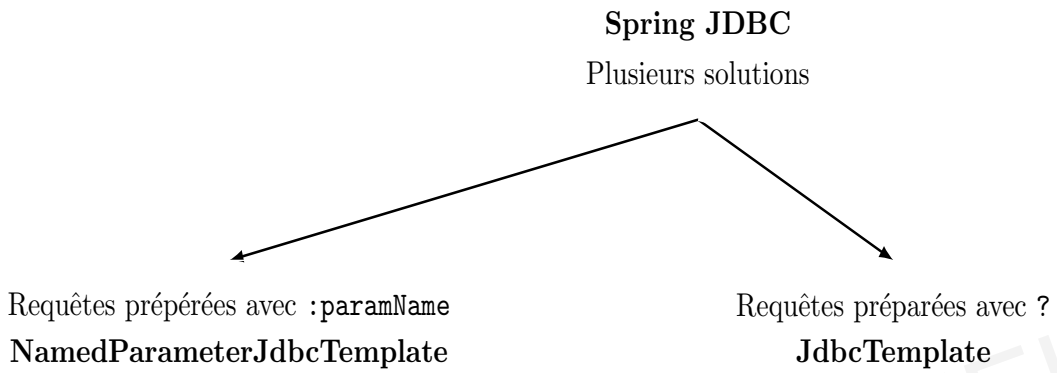
Plusieurs solutions



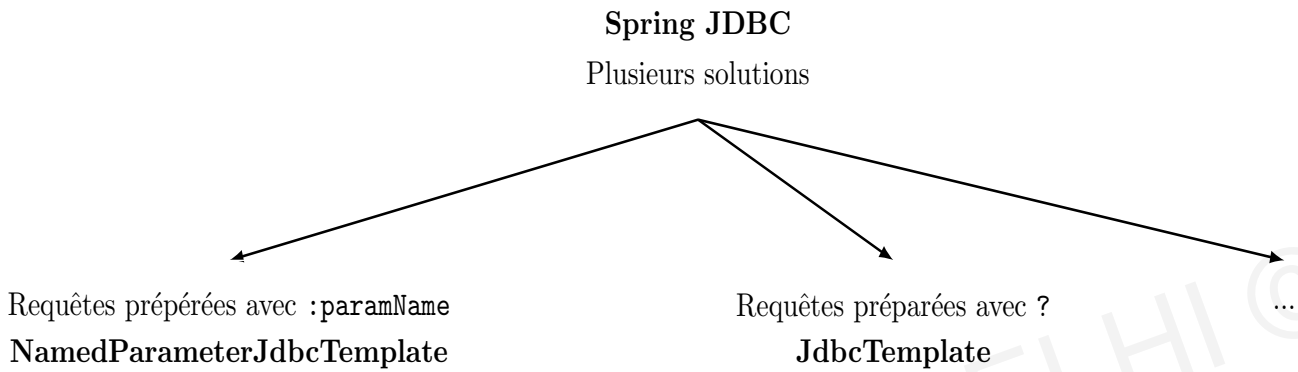
Requêtes prépérées avec `:paramName`

NamedParameterJdbcTemplate

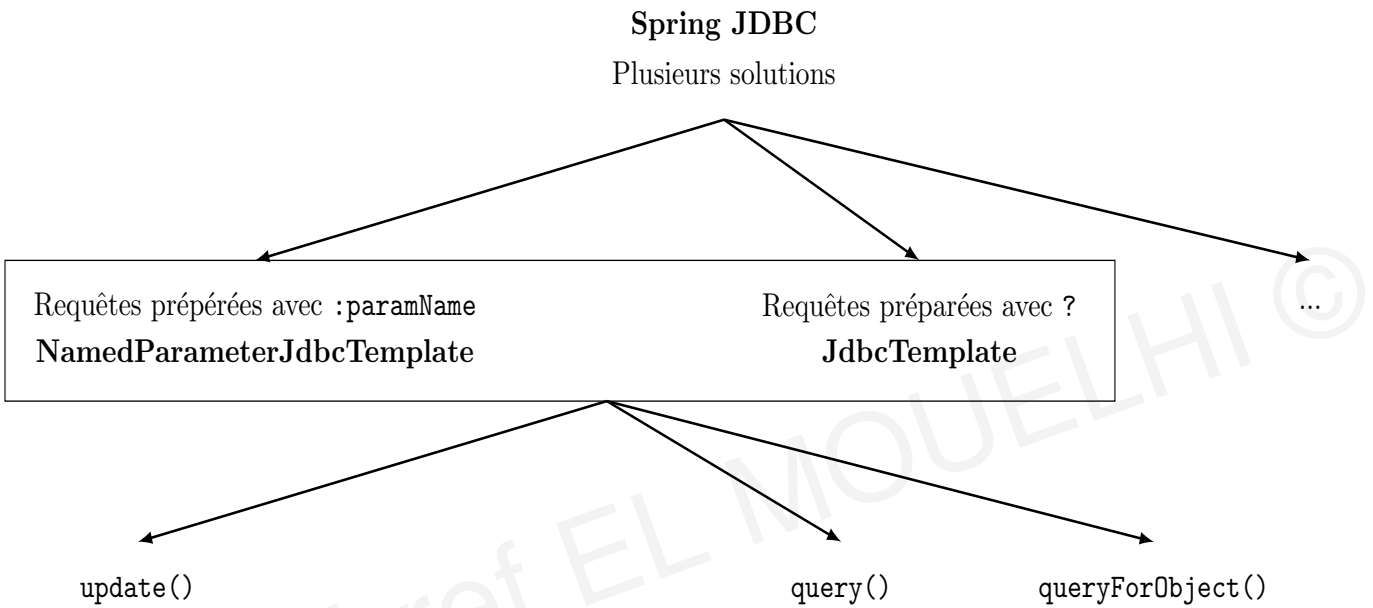
Spring Boot



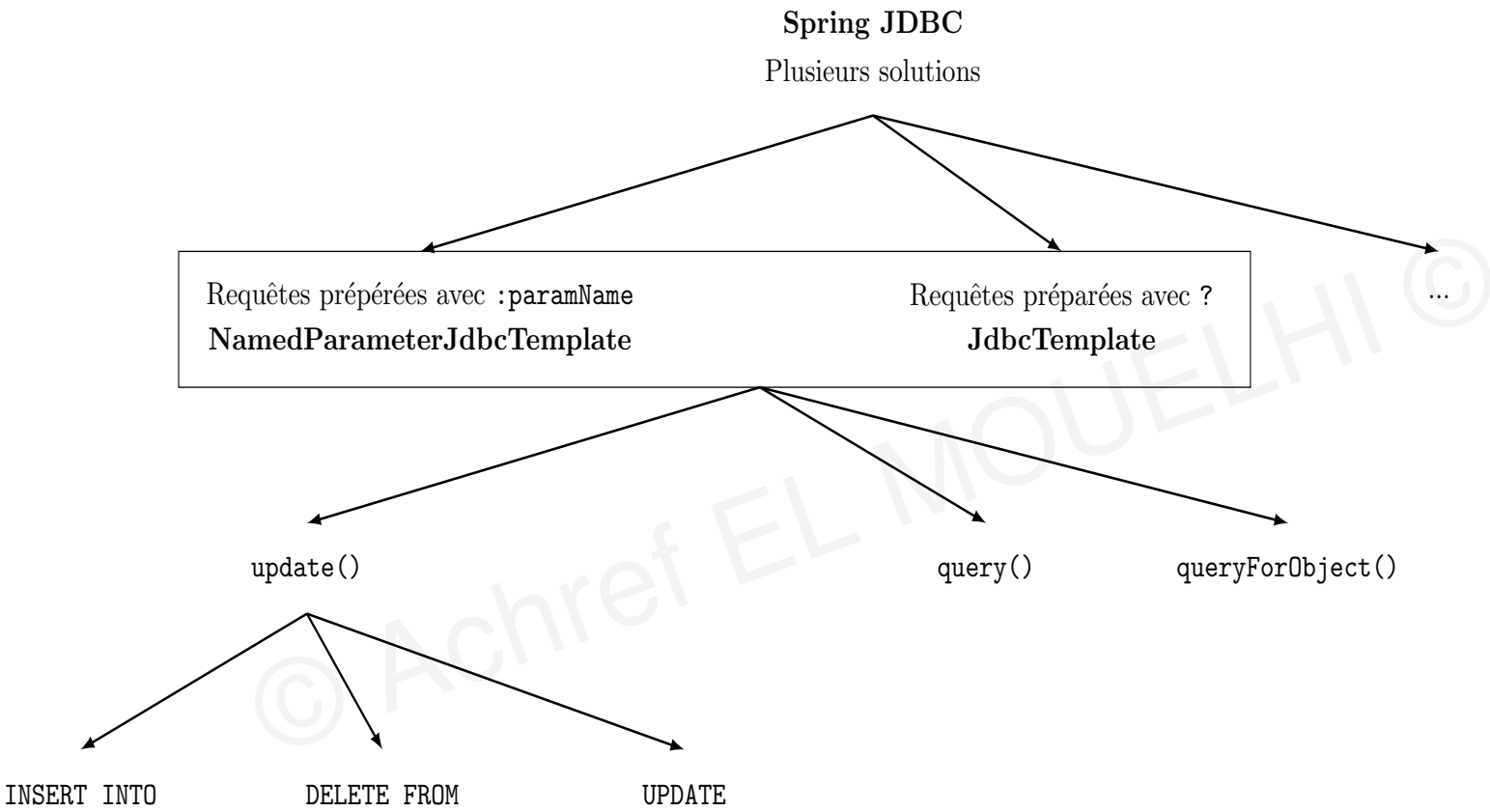
Spring Boot



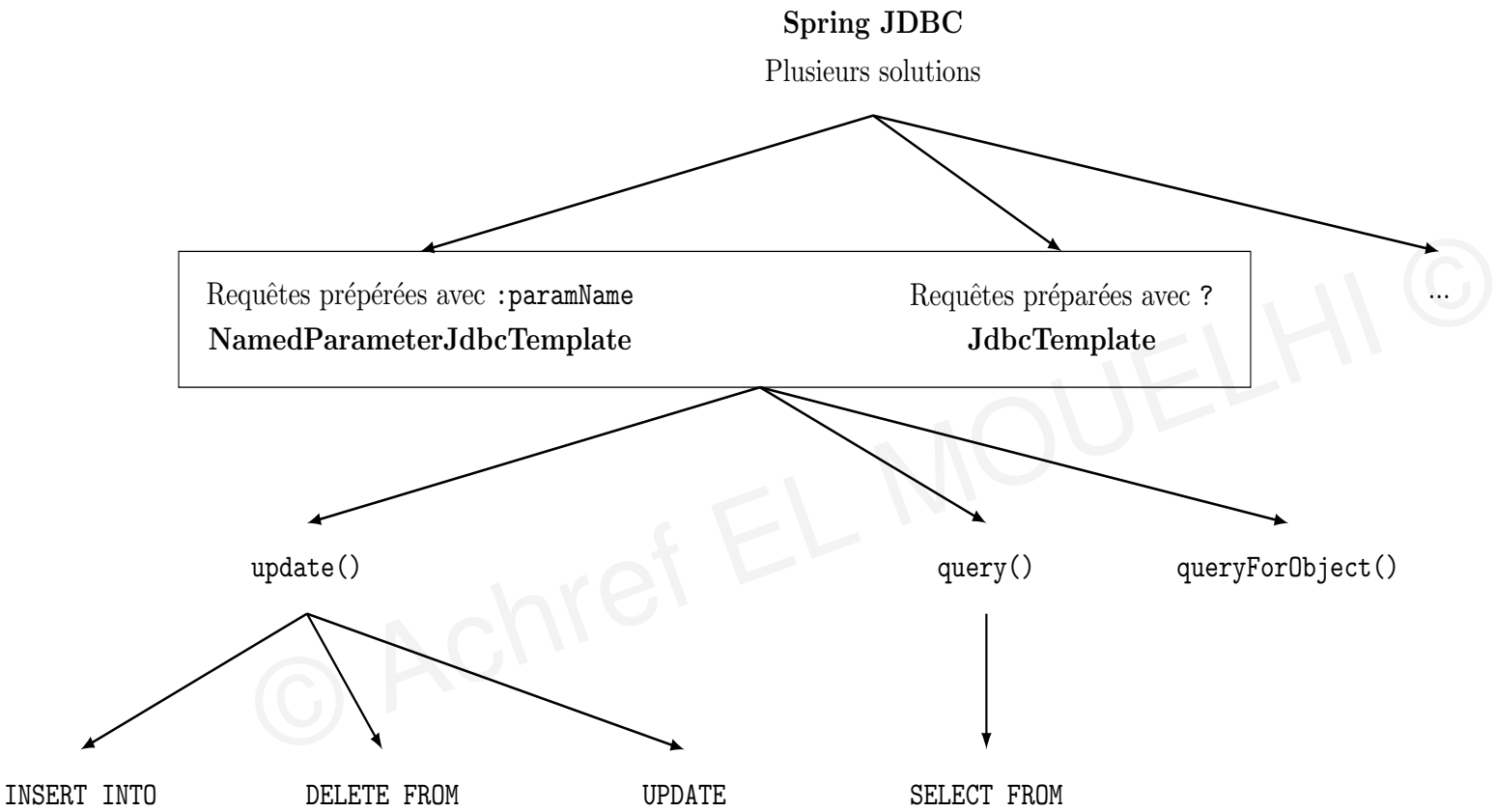
Spring Boot



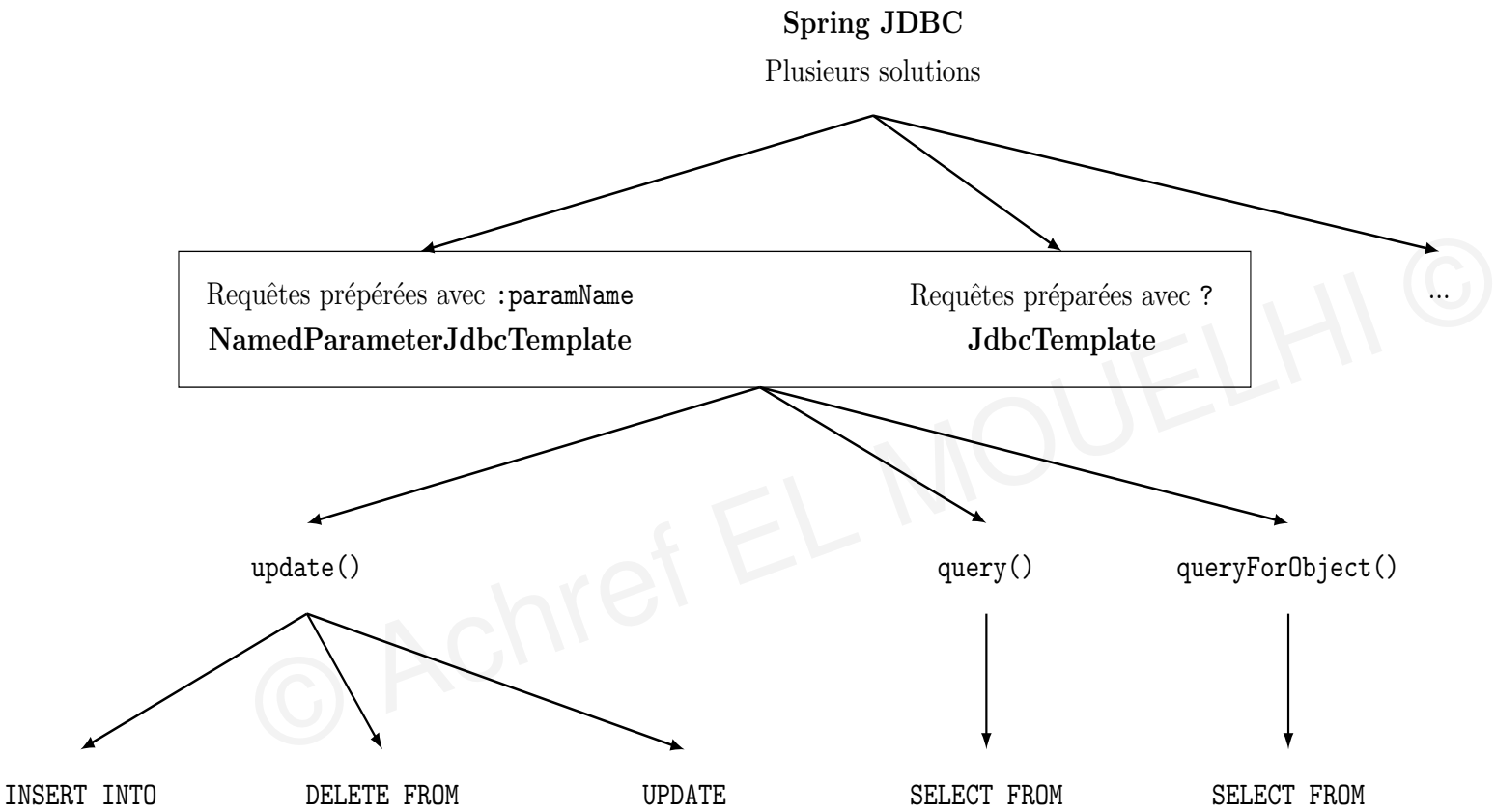
Spring Boot



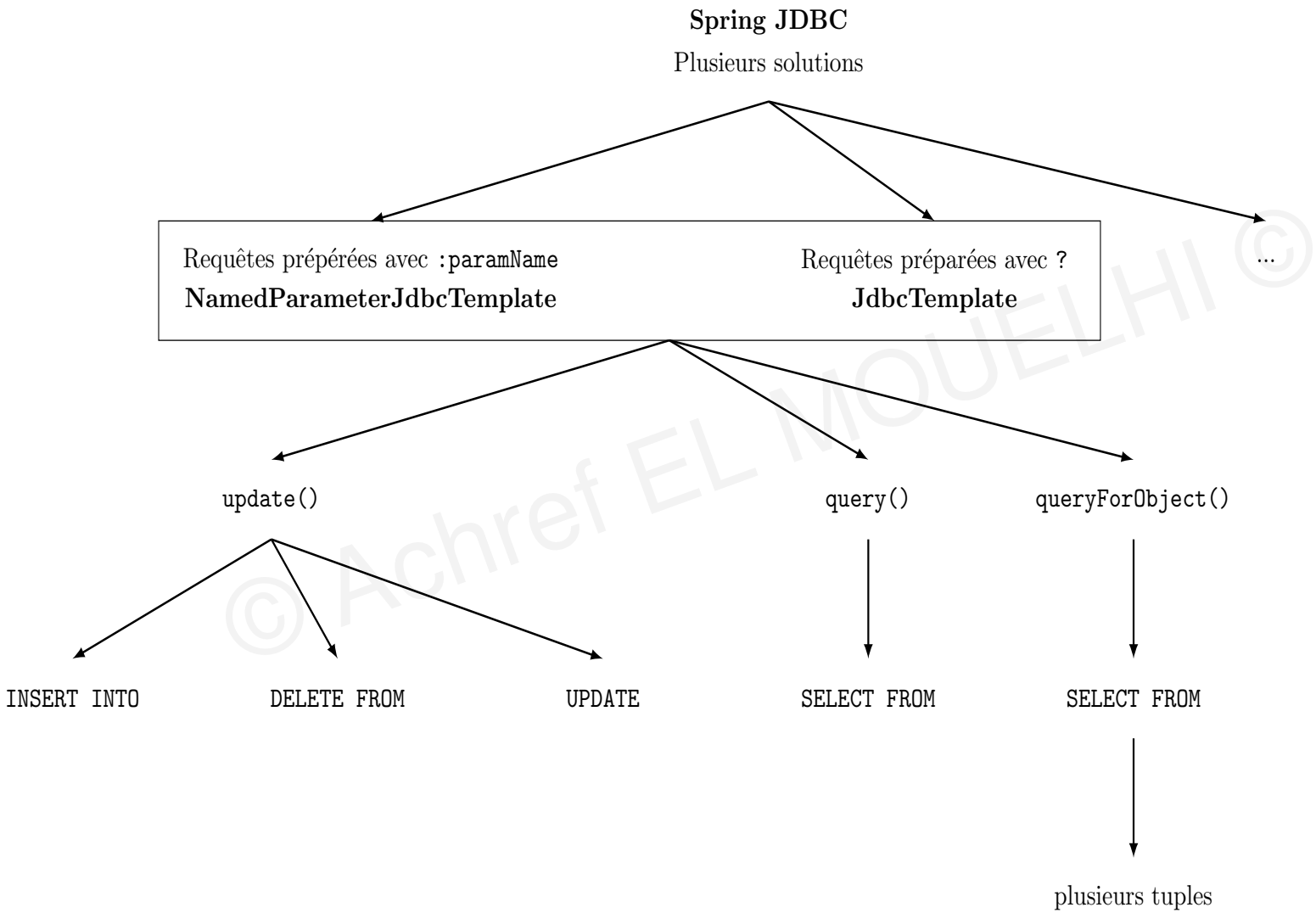
Spring Boot



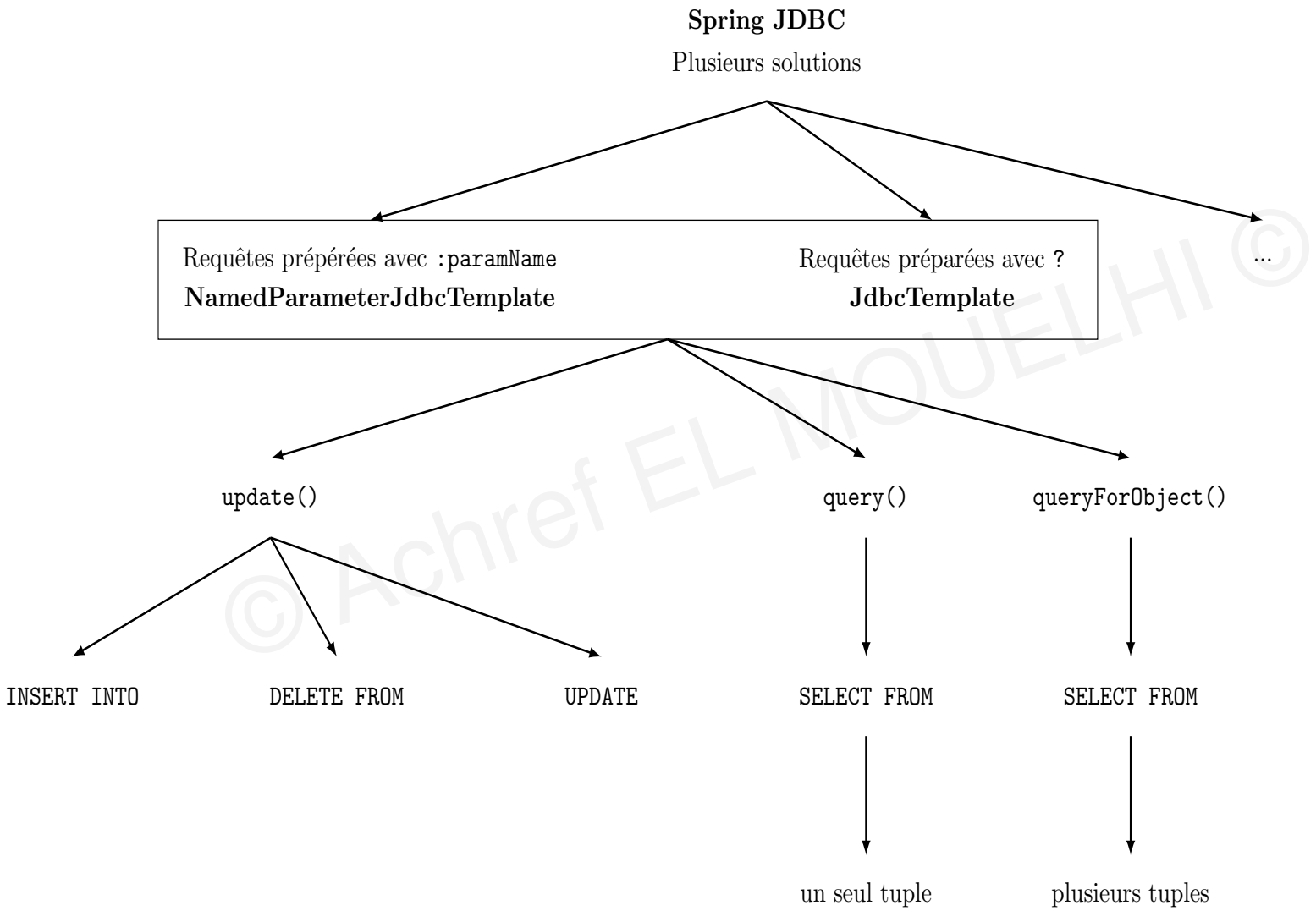
Spring Boot



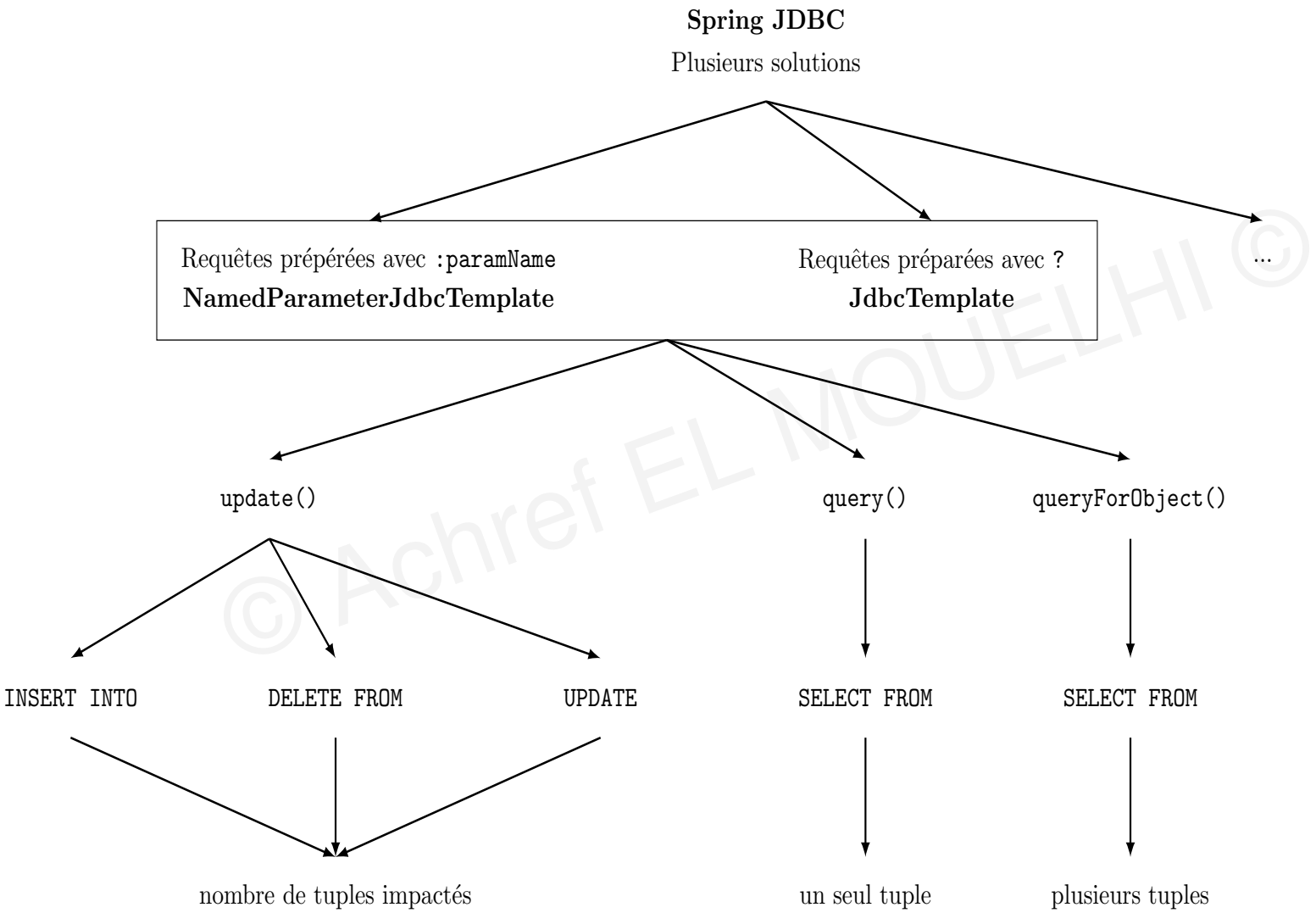
Spring Boot



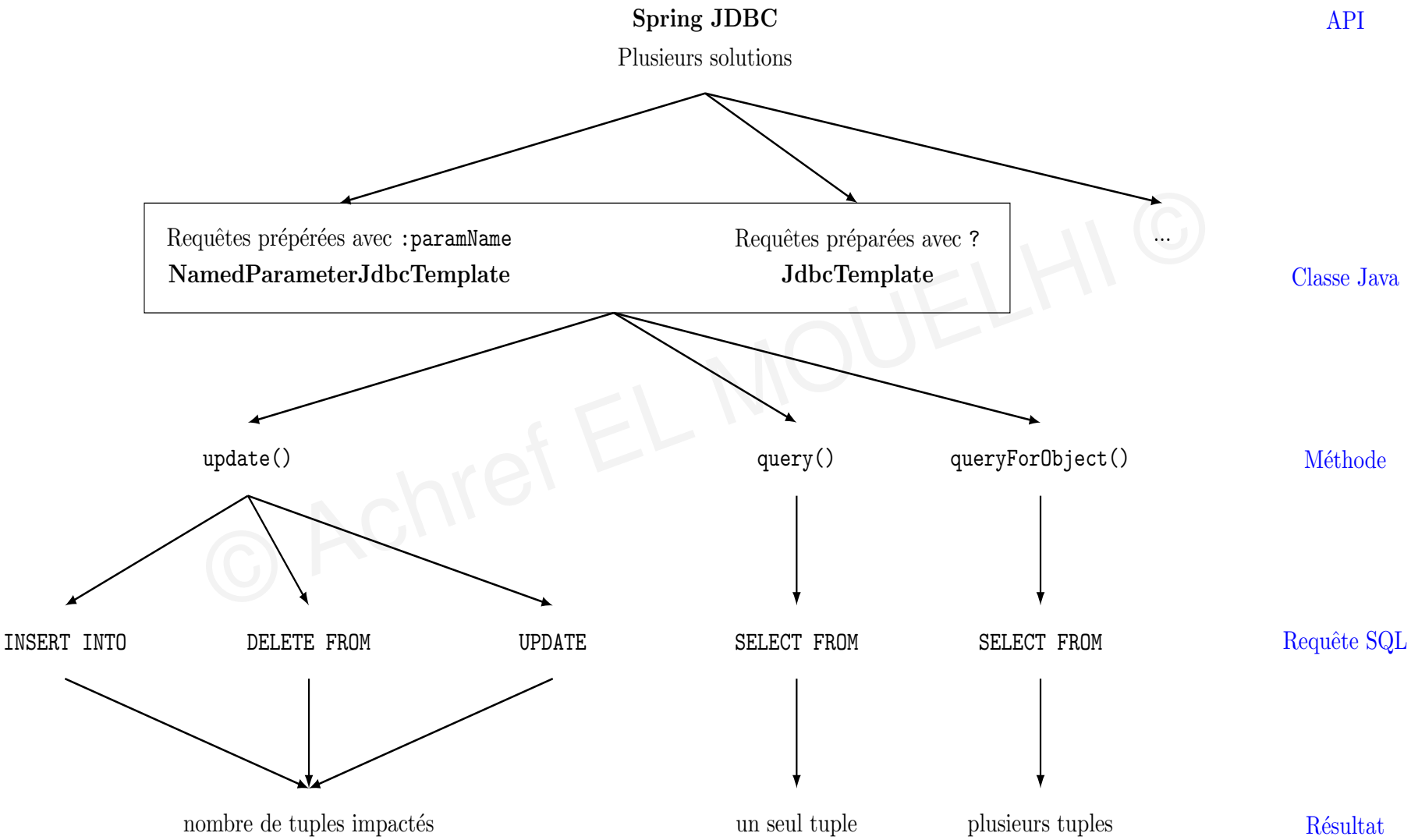
Spring Boot



Spring Boot



Spring Boot



Spring Boot

Création de projet **Spring Boot**

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `first-spring-jdbc` dans `Name`,
  - `com.example` dans `Group`,
  - `first-spring-jdbc` dans `Artifact`,
  - `com.example.demo` dans `Package`
- Cliquer sur `Next >`
- Chercher et cocher les cases correspondantes aux `Spring Web`, `JDBC API`, `MySQL Driver` et `Spring Boot DevTools` puis cliquer sur `Next >`
- Valider en cliquant sur `Finish`

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot

Organisation du projet

- Créons un premier répertoire `com.example.demo.model` dans `src/main/java` où nous placerons les entités
- Créons un deuxième répertoire `com.example.demo.dao` dans `src/main/java` où nous placerons les classes/interfaces (**Repository**) d'accès aux données **DAO**
- Créons un troisième répertoire `com.example.demo.controllers` dans `src/main/java` où nous placerons les contrôleurs
- Créons un troisième répertoire `com.example.demo.services` dans `src/main/java` où nous placerons les services

Spring Boot

Pour ajouter les dépendances **MySQL** et **API JDBC**

- Faire clic droit sur le projet et aller dans `Spring > Edit Starters`
- Cocher les cases respectives de `MySQL Driver` et `API JDBC`

Spring Boot

Pour ajouter les dépendances **MySQL** et **API JDBC**

- Faire clic droit sur le projet et aller dans `Spring > Edit Starters`
- Cocher les cases respectives de `MySQL Driver` et `API JDBC`

Ou ajouter les dépendances suivantes

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

Dans `application.properties`, on ajoute les données concernant la connexion à la base de données

```
spring.datasource.url=jdbc:mysql://localhost:3306/cours_spring_jdbc?
   serverTimezone=UTC&characterEncoding=UTF-8&useUnicode=yes&
   createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=root
```

Dans `application.properties`, on ajoute les données concernant la connexion à la base de données

```
spring.datasource.url=jdbc:mysql://localhost:3306/cours_spring_jdbc?
   serverTimezone=UTC&characterEncoding=UTF-8&useUnicode=yes&
   createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=root
```


Explication

- La base de données à utiliser dans ce cours s'appellera `cours-spring-jdbc`.
- La chaîne `serverTimezone=UTC` permet d'éviter un bug du connecteur `MySQL` concernant l'heure système.
- La chaîne `characterEncoding=UTF-8&useUnicode=yes` permet d'ajouter les caractères accentués dans la base de données.
- La chaîne `createDatabaseIfNotExist=true` permet de créer la base de données si elle n'existe pas.

Spring Boot

Pour afficher les requêtes exécutées dans la console, ajoutons la propriété suivante dans `application.properties`

```
logging.level.sql=debug
```

Spring Boot

Avant de commencer, voici le script SQL qui permet de créer la base de données utilisée dans ce cours

```
USE cours_spring_jdbc;

CREATE TABLE personne(
num INT PRIMARY KEY AUTO_INCREMENT,
nom VARCHAR(30),
prenom VARCHAR(30)
)ENGINE=InnoDB;

SHOW TABLES;

INSERT INTO personne (nom, prenom) VALUES ("Wick", "John"),
 ("Dalton", "Jack");

SELECT * FROM personne;
```

Créons une classe `Personne`

```
package com.example.demo.model;

public class Personne {

        private Long num;
        private String nom;
        private String prenom;

        public Personne() {
        }
        public Personne(String nom, String prenom) {
                this.nom = nom;
                this.prenom = prenom;
        }
        public Personne(Long num, String nom, String prenom) {
                this.num = num;
                this.nom = nom;
                this.prenom = prenom;
        }

  // + getters, setters et toString

}
```

Spring Boot

Créons une classe `PersonneDao` et injectons `JdbcTemplate`

```
package com.example.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Personne;

@Repository
public class PersonneDao {

        @Autowired
        private JdbcTemplate jdbcTemplate;

}
```

Spring Boot

Quelques méthodes de JdbcTemplate

- `update` pour insérer modifier ou supprimer des données.
- `execute` pour exécuter une requête **LDD**.
- `queryForObject` pour lire des données et retourner le résultat dans un objet.
- `query` pour lire des données et retourner le résultat dans une liste.
- ...

Spring Boot

Utilisons `JdbcTemplate` pour implémenter uen première méthode qui permet de persister les personnes

```
package com.example.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Personne;

@Repository
public class PersonneDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public int save(Personne personne) {
        String request = "INSERT INTO Personne VALUES (?, ?, ?)";
        return jdbcTemplate.update(request, personne.getNum(), personne
           .getNom(), personne.getPrenom());
        }
}
```

Spring Boot

Dans `PersonneController`**, commençons par injecter** `PersonneDao`

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.dao.PersonneDao;
import com.example.demo.model.Personne;

@RestController
public class PersonneController {

        @Autowired
        private PersonneDao personneDao;

}
```

Spring Boot

Implémentons une méthode qui intercepte les requêtes HTTP de type `POST`

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.dao.PersonneDao;
import com.example.demo.model.Personne;

@RestController
public class PersonneController {

        @Autowired
        private PersonneDao personneDao;

        @ResponseStatus(HttpStatus.CREATED)
        @PostMapping("/personne")
        public int addPersonne(@RequestBody Personne personne) {
                return personneDao.save(personne);
        }
}
```

Spring Boot

Exemple d'objet JSON à envoyer dans la requête POST

```
{
    "num": 3,
    "nom": "Baggio",
    "prenom": "Roberto"
}
```

Spring Boot

Exemple d'objet JSON à envoyer dans la requête POST

```
{
    "num": 3,
    "nom": "Baggio",
    "prenom": "Roberto"
}
```


Réponse

```
1
```

Spring Boot

Utilisons `NamedParameterJdbcTemplate` dans la méthode `save` de la classe `PersonneDao` pour écrire une requête paramétrée

```
package com.example.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Personne;

@Repository
public class PersonneDao {

        @Autowired
        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public int save(Personne personne) {
                String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
                MapSqlParameterSource params = new MapSqlParameterSource();
                params.addValue("num", personne.getNum());
                params.addValue("nom", personne.getNom());
                params.addValue("prenom", personne.getPrenom());
                return namedParameterJdbcTemplate.update(request, params);
        }
}
```

Spring Boot

Nous pouvons aussi enchaîner `addValue()`

```
package com.example.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Personne;

@Repository
public class PersonneDao {

        @Autowired
        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public int save(Personne personne) {
                String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
                MapSqlParameterSource params = new MapSqlParameterSource();
                params
                .addValue("num", personne.getNum())
                .addValue("nom", personne.getNom())
                .addValue("prenom", personne.getPrenom());
                return namedParameterJdbcTemplate.update(request, params);
        }
}
```

Spring Boot

Rien à modifier dans `PersonneController`

```
package com.example.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import com.example.demo.dao.PersonneDao;
import com.example.demo.model.Personne;

@RestController
public class PersonneController {

        @Autowired
        private PersonneDao personneDao;

        @ResponseStatus(HttpStatus.CREATED)
        @PostMapping("/personne")
        public int addPersonne(@RequestBody Personne personne) {
                return personneDao.save(personne);
        }
}
```

Spring Boot

On peut simplifier l'affectation de paramètres en associant un JavaBean à nos paramètres

```
package com.example.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Personne;

@Repository
public class PersonneDao {

        @Autowired
        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public int save(Personne personne) {
                String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
                BeanPropertySqlParameterSource params = new BeanPropertySqlParameterSource(
                  personne);
                return namedParameterJdbcTemplate.update(request, params);
        }
}
```

Spring Boot

Pour récupérer la valeur attribuée à la clé primaire

```
package com.example.demo.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Personne;

@Repository
public class PersonneDao {

        @Autowired
        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public long save(Personne personne) {
                KeyHolder keyHolder = new GeneratedKeyHolder();
                String request = "INSERT INTO Personne VALUES (:num, :nom, :prenom)";
                BeanPropertySqlParameterSource params = new BeanPropertySqlParameterSource(
                  personne);
                namedParameterJdbcTemplate.update(request, params, keyHolder);
                return keyHolder.getKey().longValue();
        }
}
```

Spring Boot

Exemple d'objet JSON à envoyer dans la requête `POST`

```
{
    "nom": "Baggio",
    "prenom": "Roberto"
}
```

Spring Boot

Explication

- Par défaut, le résultat d'une requête de lecture `SELECT FROM` est de type `ResultSet`.
- Pour transformer le `ResultSet` en un objet d'une classe `Model`, on utilise un `mapper`.
- Un mapper : classe implémentant l'interface `RowMapper<>` et sa méthode `mapRow`

Pour mapper le résultat d'une requête en un objet `Personne`, on utilise l'interface `RowMapper`

```
package com.example.demo.mapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import com.example.demo.model.Personne;

public class PersonneRowMapper implements RowMapper<Personne> {

    @Override
    public Personne mapRow(ResultSet rs, int rowNum) throws
       SQLException {
        Personne personne = new Personne();
        personne.setNum(rs.getLong("num"));
        personne.setNom(rs.getString("nom"));
        personne.setPrenom(rs.getString("prenom"));
        return personne;
    }
}
```

Spring Boot

Ajoutons une méthode `findById()` **dans** `PersonneDao` **qui utilise** `PersonneRowMapper`

```
public Personne findById(long id) {
    String request = "SELECT * FROM Personne WHERE num = :num";
    MapSqlParameterSource params = new MapSqlParameterSource();
    params.addValue("num", id);
    return namedParameterJdbcTemplate.queryForObject(request, params,
       new   PersonneRowMapper());
}
```

Spring Boot

Pour tester dans `PersonneController`

```
@GetMapping("/personne/{id}")
public Personne getOnePersonne(@PathVariable int id) {
    Personne personne;
    try {
        personne = personneDao.findById(id);
    } catch (Exception e) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND);
    }
    return personne;
}
```

Spring Boot

Deuxième version avec `JdbcTemplate`

```
public Personne findById(long id) {
    String request = "SELECT * FROM Personne WHERE num = ?";
    return jdbcTemplate.queryForObject(request, new PersonneRowMapper()
       , id);
}
```

Spring Boot

Ajoutons une méthode `findAll()` dans `PersonneDao` qui utilise `PersonneRowMapper`

```
public List<Personne> findAll() {
    String request = "SELECT * FROM Personne";
    return namedParameterJdbcTemplate.query(request, new
      PersonneRowMapper());
}
```

Spring Boot

Pour tester dans `PersonneController`

```
@GetMapping("/personne")
public List<Personne> getAllPersonnes() {
    return personneDao.findAll();
}
```

Spring Boot

Deuxième version avec `JdbcTemplate`

```
public List<Personne> findAll() {
    String request = "SELECT * FROM Personne";
    return jdbcTemplate.query(request, new PersonneRowMapper());
}
```

Spring Boot

Exercice

- Ajoutez deux méthodes et qui permettent respectivement la suppression et la modification. (Les signatures de ces méthodes sont données ci-dessous.)
- Proposez deux versions de ces méthodes : une utilisant `JdbcTemplate` et l'autre `NamedParameterJdbcTemplate`.
- Utilisez les deux méthodes dans deux actions différentes du contrôleur `PersonneController`.


```
public int delete(int id) {
}

public int update(Personne personne) {
}
```

Spring Boot

Explication

- Les méthodes définies dans `PersonneDao` sont des méthodes souvent présentes dans toutes les classes **DAO**.
- Pour une meilleure structuration du projet, définissons une interface `Dao<C, T>` contenant la signature des cinq méthodes les plus utilisées.
  - `C` : type générique à remplacer par une classe `model`
  - `T` : type objet de l'identifiant de la classe `model`
- Toute classe **DAO** implémentera `Dao<C, T>` et redéfinira donc toutes ses méthodes abstraites.

Spring Boot

Contenu de `Dao<C, T>`

```
package com.example.demo.model;

import java.util.List;

public interface Dao<C, T> {

    public T save(C obj);

    public C findById(T id);

    public List<C> findAll();

    public int delete(T id);

    public int update(C obj);
}
```

Spring Boot

Considérons la classe `Adresse` suivante

```
package com.example.demo.model;

public class Adresse {
  private Long id;
  private String rue;
  private String codePostal;
  private String ville;

  public Adresse() {
  }

  public Adresse(Long id, String rue, String codePostal, String ville)
     {
    this.id = id;
    this.rue = rue;
    this.codePostal = codePostal;
    this.ville = ville;
  }

  // + getters / setters / toString
```

Spring Boot

Dans `Personne`, définissons un nouvel attribut `adresses`

```
public class Personne {

  private Long num;
  private String nom;
  private String prenom;

  private List<Adresse> adresses;

  // + getter / setter / toString

}
```

Exécutons le script suivant pour mettre à jour la base de données avec les nouvelles tables

```
DROP DATABASE cours_spring_jdbc;
CREATE DATABASE cours_spring_jdbc;
USE cours_spring_jdbc;

CREATE TABLE personne(
num INT PRIMARY KEY AUTO_INCREMENT,
nom VARCHAR(30),
prenom VARCHAR(30)
)ENGINE=InnoDB;

INSERT INTO personne (nom, prenom) VALUES ("Wick", "John"), ("Dalton", "Jack");

CREATE TABLE adresse(
id INT PRIMARY KEY AUTO_INCREMENT,
rue VARCHAR(30),
code_postal VARCHAR(30),
ville VARCHAR(30)
)ENGINE=InnoDB;

INSERT INTO adresse (rue, code_postal, ville) VALUES
("paradis", "13006", "Marseille"),
("plantes", "75014", "Paris");

CREATE TABLE personne_adresse(
id INT PRIMARY KEY AUTO_INCREMENT,
num_personne INT,
id_adresse INT,
FOREIGN  KEY (num_personne) REFERENCES personne (num),
FOREIGN  KEY (id_adresse) REFERENCES adresse (id)
)ENGINE=InnoDB;

INSERT INTO personne_adresse (num_personne, id_adresse) VALUES (1, 1), (1, 2), (2, 2);
```

Spring Boot

Exercice 1

- Créez une classe `AdresseDao` qui implémente `Dao`
- Implémentez les méthodes de `Dao`
- Préparez le service et le contrôleur qui permettrons d'interroger `AdresseController` comme une ressource **REST**.

Exercice 2

Ajoutes les méthodes nécessaires dans les classes **DAO**, service et contrôleur afin de permettre de tester les requêtes suivantes

- `GET` (Route : `/personne/{id}`) qui permettra de retourner une personne selon son identifiant
- `GET` (Route : `/personne/{id}/adresse`) qui permettra de retourner les adresses d'une personne selon son identifiant
- `GET` (Route : `/personne/{idPersonne}/adresse/{idAdresse}`) qui permettra de retourner l'adresse d'une personne selon leurs identifiants

Pensez à implémenter les méthodes suivantes

```
public List<Adresse> findAdressesByPersonneId(int id) {
}

public Adresse findAdresseById(int idPersonne, int idAdresse) {
}
```

Spring Boot

Exercice 3

Ajoutes les méthodes nécessaires dans les classes **DAO**, service et contrôleur afin de permettre de tester les requêtes suivantes

- `POST` (Route : `/personne/{idPersonne}/adresse/{idAdresse}`) qui permettra d'associer la personne (ayant la clé `idPersonne`) et (ayant la clé `idAdresse`), c'est-à-dire ajouter un tuple dans la table de jointure (`personne_adresse`).
- `DELETE` (Route : `/personne/{idPersonne}/adresse/{idAdresse}`) qui permettra de supprimer l'association entre la personne (ayant la clé `idPersonne`) et (ayant la clé `idAdresse`) dans la table de jointure (`personne_adresse`).

Pensez à implémenter les méthodes suivantes dans une classe DAO

```
public int mapPersonneAdresse(Integer idPersonne, Integer idAdresse) {
}

public int unmapPersonneAdresse(Integer idPersonne, Integer idAdresse) {
}
```

Spring Boot

Exercice 4

- Modifiez la méthode de suppression une personne ayant des adresses. Ne supprimez pas les adresses. Supprimer uniquement les liens entre personne et adresses dans la table de jointure (`personne_adresse`).
- Modifiez la méthode d'ajout pour qu'elle permette d'ajouter une personne avec des adresses. Si l'adresse n'a pas d'identifiant elle sera insérée dans la table `adresse`. Dans tous les cas, ajoutes les liens entre personne et adresses dans la table de jointure (`personne_adresse`).

Spring Boot

Exercice 5

Créer une application **Angular**, **React**, **Vue.js** ou pure **JavaScript** qui permet à un utilisateur, via des interfaces graphiques, la gestion de personnes (ajout, modification, suppression, consultation et recherche) en utilisant les web services définis par **Spring**.

Spring Boot

Exercice 5

Créer une application **Angular**, **React**, **Vue.js** ou pure **JavaScript** qui permet à un utilisateur, via des interfaces graphiques, la gestion de personnes (ajout, modification, suppression, consultation et recherche) en utilisant les web services définis par **Spring**.

Pour régler le problème **CORS**

Il faut ajouter l'annotation `@CrossOrigin` au contrôleur.