

SCALA: Slick

Contents

1	Slick	2
1.1	Application d'exemple	2
1.2	Exemple de données	2
1.3	Premier exemple de requête dans Slick	3
1.4	Intégration de Slick dans un projet Play Framework	4
1.5	Les Modèles	4
1.6	Les DAO	5
1.7	Utilisation des DAO	7
2	Références	8

1 Slick

Slick est une librairie Scala permettant d'accéder à une base de données et de l'interroger en Scala. Elle permet de travailler avec des données stockées en interne, permettant ainsi d'utiliser des collections Scala et donc d'utiliser les paradigmes fonctionnels offerts par le langage. Il s'agit donc d'un FRM (Functional-Relational Mapping), la version fonctionnelle d'un ORM (Object-Relational Mapping) comme pourrait l'être Hibernate, par exemple.

Les avantages de cette technologie résident dans le fait que le code est écrit en Scala au lieu d'être directement rédigé en SQL, permettant ainsi de générer du code SQL différent selon la DB utilisée (MySQL, SQLite, PostgreSQL, SQLServer, Oracle, H2, etc.), et offrant une sécurité accrue grâce au typage fort et grâce aux contrôles effectuée lors de la compilation.

Dans ce tutoriel, nous utiliserons MySQL ; veuillez donc à ce que ce dernier soit installé sur votre machine de développement, si vous souhaitez pouvoir explorer complètement l'application d'exemple.

1.1 Application d'exemple

Vous trouverez le code complet de l'application décrite tout au long des exemples de ce document sur [ce repo GitHub](#), dans la branche *fb-play-slick-app*. Cette application se base sur l'application REST présentée dans le tutoriel d'explication de Play Framework.

Pour faire fonctionner l'application, il faut créer une base de données "scala_sql_example" dans MySQL et y importer le script SQL contenu dans le fichier `/sql/script.sql`. Si l'IDE ne reconnaît pas les différents mots-clés et génère plein d'erreurs, il faut ajouter le support du framework Play au projet (**clic-droit sur le projet | Add Framework Support... | cocher Play 2.x**).

A noter que vous pouvez trouver d'autres exemples d'applications Slick basiques [ici](#).

1.2 Exemple de données

Afin que vous puissiez mieux comprendre les explications qui suivent, nous allons utiliser une base de données MySQL assez basique qui gère une liste d'étudiants et de cours ; voici une ébauche du diagramme :

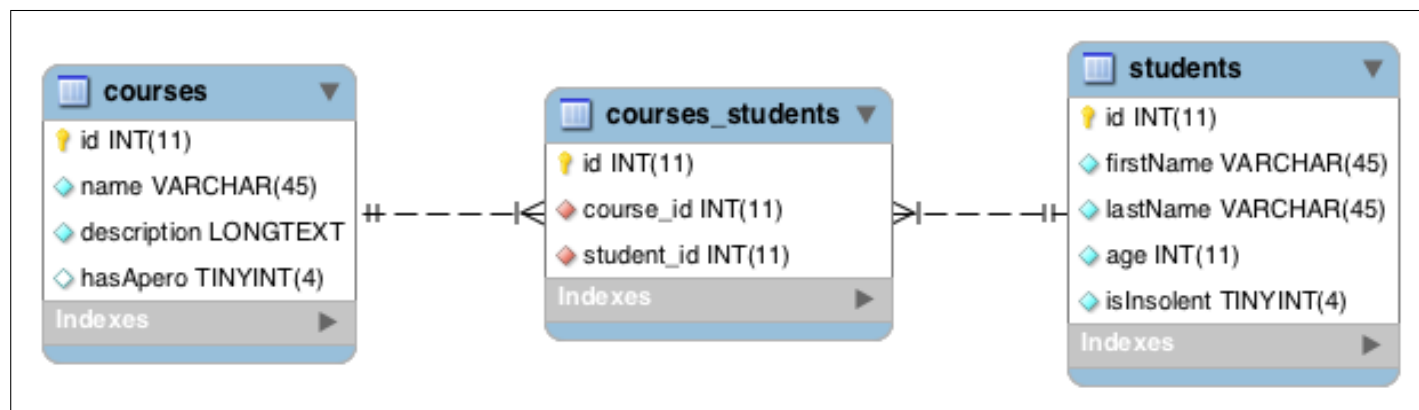


Figure 1: diagramme de la base de données qui sera utilisée dans ce tutoriel

La table « students » contient une liste des étudiants, avec pour chaque entrée le prénom et le nom de l'étudiant, ainsi qu'un champ de type booléen qui indique si l'élève est insolent ou non envers le pauvre assistant. Cette table contient les données suivantes :

id	firstName	lastName	age	isInsolent
1	Valentin	Niffi	23	1
2	Céline	Jacquart	21	0
3	Guillaume	Jurapu	18	1
4	Michel	Sainte-Marie	24	0

La table « courses » contient une liste de cours, avec pour chaque entrée un nom de cours, sa description, ainsi qu'un champ facultatif de type booléen qui indique si un apéro est organisé ou non par le professeur à la fin du semestre. Exemple :

id	name	description	hasApero
1	GEN	Du génie, des logiciels, et des ninjas.	1
2	TWEB	Apprendre aux étudiants que JavaScript, c'est cool.	1
3	SCALA	Apprendre aux étudiants que Scala aussi, c'est cool.	NULL
4	PRR	Cours d'ADA et de programmation répartie.	0

Une table de liaison « courses_students » permet de lier un ou plusieurs étudiants à un ou plusieurs cours. Valentin et Guillaume suivent tous les cours, Céline suit uniquement le cours de TWEB, et Michel suit tous les cours, sauf PRR.

1.3 Premier exemple de requête dans Slick

Ok, ok, vous voilà donc tout excités à l'idée d'apprendre une nouvelle technologie ! Ceci dit, avant de passer à l'installation même de la librairie et de foncer tête baissée dans la marée noire, voici un premier exemple de requête qui vous permettra de faire le lien entre du SQL et les requêtes que l'on utilisera avec Slick.

Pour réaliser la sélection du prénom et du nom des étudiants dont l'âge est inférieur à 20 dans la table "students", une requête SQL ressemblerait à ceci :

```
SELECT firstName, lastName FROM students WHERE age < 20;
```

La version équivalente à cette sélection en Scala avec Slick s'écrit ainsi :

```
(for (s <- students; if s.age < 20) yield (s.firstName, s.lastName)).result
```

Il reste ensuite *simplement* à déterminer comment récupérer l'object "students" contenant la liste des étudiants à partir de la base de données, mais patience, nous verrons tout cela plus tard dans le document.

1.4 Intégration de Slick dans un projet Play Framework

Passons tout de suite sur du concret ! Pour pouvoir utiliser Slick dans un projet Play, il suffit simplement d'ajouter deux nouvelles dépendances (Slick, ainsi que le connecteur à la base de données) dans le fichier **build.sbt**, puis de rafraîchir le projet et de laisser SBT télécharger ces dépendances :

```
libraryDependencies += "com.typesafe.play" %% "play-slick" % "3.0.0" // Slick
libraryDependencies += "mysql" % "mysql-connector-java" % "5.1.24" // Connecteur MySQL
```

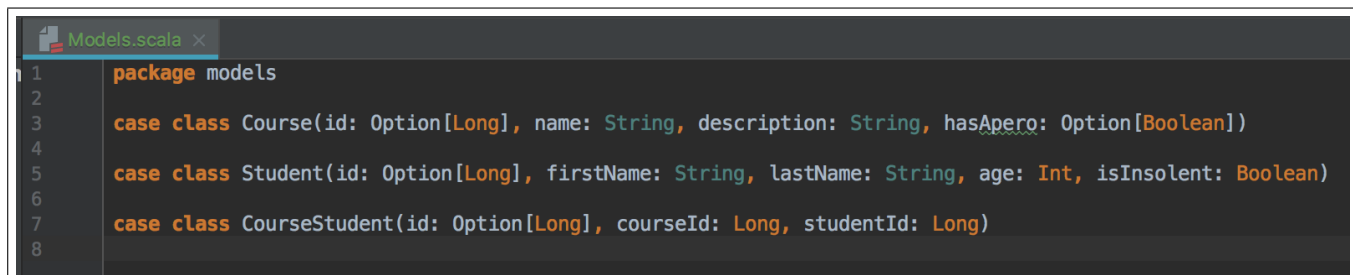
D'une fois les dépendances téléchargées, il faut configurer l'accès à la base de données depuis le serveur. Pour ce faire, ajouter les lignes suivantes tout à la fin du fichier de configuration de l'application (*/conf/application.conf*) en changeant les variables *[DATABASE_NAME]*, *[USER]* et *[PASSWORD]*, qui correspondent comme leur nom l'indique au nom de la base de données sur le serveur, ainsi qu'à l'utilisateur et au mot de passe utilisés pour s'y connecter :

```
slick.dbs.default.profile = "slick.jdbc.MySQLProfile$"
slick.dbs.default.db.driver = "com.mysql.jdbc.Driver"
slick.dbs.default.db.url = "jdbc:mysql://localhost/[DATABASE_NAME]"
slick.dbs.default.db.user = "[USER]"
slick.dbs.default.db.password = "[PASSWORD]"
```

Par convention, la base de données utilisée par défaut avec Slick est nommée "default". S'il est nécessaire d'avoir plusieurs DB dans une même application, il est possible d'ajouter plusieurs fois ces lignes de configuration en nommant les bases de données différemment (par exemple `slick.dbs.beer.profile = "slick.jdbc.MySQLProfile$"` pour une base de données qui s'appellerait "beer" dans l'application).

1.5 Les Modèles

Un modèle est la représentation objet d'une entrée d'une table de la base de données (donc par exemple la représentation d'un étudiant), déclarée sous forme de `case class`. Ces classes contiennent généralement un attribut pour chaque champ de la table liée, dans la base de données. Nous avons donc besoin de 3 modèles dans notre application : `Course`, `Student` et `CourseStudent`. Afin de rester propres dans nos démarches, nous avons ici créé un nouveau package "models" dans le dossier "app", contenant un fichier "Models.scala" qui contient lui-même la liste des modèles.



```
1 package models
2
3 case class Course(id: Option[Long], name: String, description: String, hasApero: Option[Boolean])
4
5 case class Student(id: Option[Long], firstName: String, lastName: String, age: Int, isInsolent: Boolean)
6
7 case class CourseStudent(id: Option[Long], courseId: Long, studentId: Long)
8
```

Figure 2: modèles représentant les entités de la base de données dans notre application

Les IDs sont ici facultatifs, car nous ne les utiliserons pas forcément tout le temps. En effet, lors de la création d'un nouvel étudiant, nous ne souhaitons pas avoir à indiquer manuellement un ID et préférons laisser le serveur MySQL régler ce problème, contrairement à la modification de l'étudiant, où nous souhaitons pouvoir passer l'ID afin de localiser l'élève qui verra son identité changer à jamais.

1.6 Les DAO

Aaah, les DAO ! Ce mot à lui seul devrait vous faire TILT, et si ça n'est pas le cas, je vais le faire pour vous, comme disait l'autre : ***TILT*** !

Ok, rappel, les DAO c'est quoi ? Définition : un DAO (Data Access Object) est une classe qui regroupe tous les accès aux données persistantes, en général pour une table de la base de données. En plus de contenir les instructions qui permettent représenter une table sous forme de code orienté-objet (dans notre cas) et de permettre ainsi aux développeurs de ne pas perdre leurs repères, le DAO contient différentes méthodes d'accès aux données (par exemple la récupération de la liste des étudiants, la récupération d'un étudiant, l'édition, la suppression, l'ajout, etc.). Les DAO sont contenus dans l'exemple dans le package "dao".

Dans Slick notamment, une table de requête (query table) est utilisée pour faire le lien entre la base de données et un modèle :

```
// This class convert the database's courses table in a object-oriented entity: the Course model.
class CoursesTable(tag: Tag) extends Table[Course](tag, "COURSES") {
  def id = column[Long]("ID", 0.PrimaryKey, 0.AutoInc) // Primary key, auto-incremented
  def name = column[String]("NAME")
  def description = column[String]("DESCRIPTION")
  def hasApero = column[Option[Boolean]]("HASAPER0") // Optional field

  // Map the attributes with the model; the ID is optional.
  def * = (id.?, name, description, hasApero) <> (Course.tupled, Course.unapply)
}
```

Figure 3: table de requête des cours

Cette table de requête lie chaque colonne de la table à un attribut. Nous indiquons à chaque fois le type de la colonne, suivi de son nom ainsi que d'éventuels autres paramètres. Pour une question de respect des normes, le nom de la table et des champs sont écrits en majuscule, même s'ils sont en minuscules dans la base de données.

Le DAO utilise ensuite cette classe pour obtenir la liste de toutes les entrées de la table, et pour pouvoir travailler dessus.

```
// Get the object-oriented list of courses directly from the query table.
val courses = TableQuery[CoursesTable]

/** Retrieve the list of courses sorted by name */
def list(): Future[Seq[Course]] = {
  val query = courses.sortBy(s => s.name)
  db.run(query.result)
}
```

Figure 4: récupération de la liste des cours, à l'aide de la table de requête, dans le DAO

Dans l'exemple ci-dessus, le DAO permet de récupérer la liste des cours sous forme de tableau contenant une liste de modèles de cours (`Future[Seq[Course]]`).

D'ailleurs, quel est le rôle du type `Future[...]` ? Celui-ci agit comme une promesse et représente le futur résultat d'une requête (ce qui signifie que la valeur de retour de la méthode ne contient pas forcément tout de suite une valeur à proprement parler, et que celle-ci sera injectée dans la variable lorsque le résultat aura été obtenu), ce qui rend le processus asynchrones ("Hello, old friend."). Afin de récupérer la valeur d'une promesse lorsque celle-ci aura changé de statut, il faudra effectuer un `map` sur la variable (voir le chapitre "Utilisation des DAO" pour plus de détails).

Mais revenons à nos DAO ! Il est désormais possible d'accéder aux données et de travailler dessus. Voici 4 méthodes d'exemple, qui permettent de récupérer un cours selon son ID, d'en insérer un nouveau, d'en modifier et d'en supprimer un existant :

```
/** Retrieve a course from the id. */
def findById(id: Long): Future[Option[Course]] =
  db.run(courses.filter(_id == id).result.headOption)

/** Insert a new course, then return it. */
def insert(course: Course): Future[Course] = {
  val insertQuery = courses returning courses.map(_id) into ((course, id) => course.copy(Some(id)))
  db.run(insertQuery += course)
}

/** Update a course, then return an integer that indicates if the course was found (1) or not (0). */
def update(id: Long, course: Course): Future[Int] = {
  val courseToUpdate: Course = course.copy(Some(id))
  db.run(courses.filter(_id == id).update(courseToUpdate))
}

/** Delete a course, then return an integer that indicates if the course was found (1) or not (0) */
def delete(id: Long): Future[Int] =
  db.run(courses.filter(_id == id).delete)
```

Figure 5: méthodes d'accès aux données depuis le DAO

Il est bien-entendu possible d'écrire des requêtes plus complexes dans le DAO. Voici par exemple une requête qui permet de récupérer le prénom, le nom et l'âge de tous les étudiants dont l'âge est inférieur au paramètre donné, puis de trier le tout par nom et par prénom :

```
/** Retrieve the names (first and last names) and the age of the students, whose age is inferior of the given one,
 * then sort the results by last name, then first name */
def findIfAgeIsInferior(age: Int): Future[Seq[(String, String, Int)]] = {
  val query = (for {
    student <- students
    if student.age < age
  } yield (student.firstName, student.lastName, student.age)).sortBy(s => (s._2, s._1))
  db.run(query.result)
}
```

Figure 6: récupération du prénom et nom des étudiants dans le DAO

Finalement, il est possible d'exécuter des jointures en utilisant les tables de requête des autres DAO, par exemple ici en listant tous les étudiants d'un cours :

```
@Singleton
class CoursesDAO @Inject()(protected val dbConfigProvider: DatabaseConfigProvider)(implicit executionContext: ExecutionContext)
  extends CoursesComponent with StudentsComponent with CoursesStudentsComponent with HasDatabaseConfigProvider[JdbcProfile] {

  import profile.api._

  // Get the object-oriented list of courses directly from the query table.
  val courses = TableQuery[CoursesTable]
  val students = TableQuery[StudentsTable]
  val coursesStudents = TableQuery[CoursesStudentsTable]

  /** Get the students associated with the given course's ID. */
  def getStudentsOfCourse(id: Long): Future[Seq[Student]] = {
    val query = for {
      courseStudent <- coursesStudents
      student <- students if courseStudent.studentId === student.id
    } yield student

    db.run(query.result)
  }
}
```

Figure 7: méthodes d'accès aux données depuis le DAO

1.7 Utilisation des DAO

Finalement, lorsque les modèles et les DAO ont été créés, il suffit d'injecter le ou les DAO dans un contrôleur pour pouvoir utiliser leurs méthodes. Facile, non ?

```
@Singleton
class StudentsController @Inject()(cc: ControllerComponents, studentDAO: StudentsDAO) extends AbstractController(cc) {

  implicit val studentToJson: Writes[Student] = ??? // See full example for the serialization code.
  implicit val jsonToStudent: Reads[Student] = ??? // See full example for the serialization code.

  /**
   * Get the list of all existing students, then return it.
   * The Action.async is used because the request is asynchronous.
   */
  def getStudents = Action.async {
    val studentsList = studentDAO.list()
    studentsList map (s => Ok(Json.toJson(s)))
  }
}
```

Figure 8: utilisation du DAO des étudiants dans le contrôleur des étudiants

Comme vu précédemment, les méthodes offertes par le DAO renvoient des promesses, ce qui rend tout le fil d'exécution asynchrone. Afin de rendre les actions du contrôleur compatible avec cet asynchronisme, il faut que celles-ci soient déclarées comme étant des `Action.async` au lieu de simples `Action`.

2 Références

- <https://www.playframework.com/documentation/fr/2.4.x/PlaySlick>
- <https://github.com/playframework/play-slick/tree/master/samples>