

**Scala**

**Mini-Projet**

***Scala et Play Framework***

Sommaire

[1 Technologies 2](#_Toc516998369)

[2 Choix 3](#_Toc516998370)

[3 Schéma 4](#_Toc516998371)

[4 Implémentation 4](#_Toc516998372)

[5 Problèmes rencontrés 11](#_Toc516998373)

[6 Conclusion 12](#_Toc516998374)

# Technologies

Le but de ce projet est d’utiliser le langage de programmation Scala, appliqué à un framework écrit lui-même en Scala, « Play ».

#### Play

C’est un framework web open-source, il est possible d’y utiliser Scala, mais aussi Java. Comme un bon nombre de framework, il est basé sur le patron de conception MVC (modèle –vue – contrôleur). Il est aussi possible d’y développer des backend API REST. Les processus de production et de déploiement sont gérés par SBT.

#### Base de données

En ce qui concerne la base de données, nous utilisons Mysql, qui nous permet de fonctionner avec une librairie Scala « Slick ».

#### Slick

C’est une librairie Scala permettant d’accéder à une base de données et de l’interroger en Scala. C’est un FRM (Functionnal-Relational Mapping), l’équivalent d’un ORM pour du langage fonctionnel. (Équivalent à Hibernate).

Pour travailler avec Slick, il faut ajouter différents parties à notre code. Des Modèles, des DAO et ensuite des contrôleurs. Ceci pour chaque table présente dans notre base de données. Un exemple sera fourni ultérieurement.

#### Javascript, HTML Css

Ces langages sont utilisés pour écrire des applications web frontend.

# Choix

Pour ce projet la seule contrainte était d’utiliser Scala avec le framework Play en backend. Nous avons choisi d’utiliser Play en simple API REST, et donc de créer un frontend totalement dissocié du backend. Nous n’utilisons donc pas Play en MVC classique.

Pour le frontend, afin d’avoir un visuel sympathique, et d’avoir une application totalement responsive (utilisation sur écran de smartphone ou tablette facile) nous avons décidé de nous baser sur un template bootstrap, et d’y apporter des modifications en HtML, CSS et JS.

Notre projet est une application web qui permet de soumettre une question à une communauté. Cette question contient :

* Une question posée, sous forme de texte uniquement
* Une première proposition de réponse qui contient soit un texte, soit une image ou les deux
* Une deuxième question semblable à la première

Il y a un système de connexion, qui nous redirige vers une page utilisateur, et c’est depuis cette page que l’ajout d’une nouvelle question est possible.

Sur cette même page, nous pouvons répondre à des questions posées par les autres utilisateurs. Un bouton est prévu à cet effet, et il ouvre une pop-up ou nous voyons une question posée à laquelle nous n’avons pas encore répondu, et nous avons le choix de sélectionner une des deux proposition de réponse. Lorsque nous avons répondu à la question des statistiques apparaissent. (Pourcentage de vote par tranches d’âge, par sexe etc).

En fin de page nous avons une liste de nos questions, que nous pouvons effacer, ou ouvrir les statistiques.

# Schéma

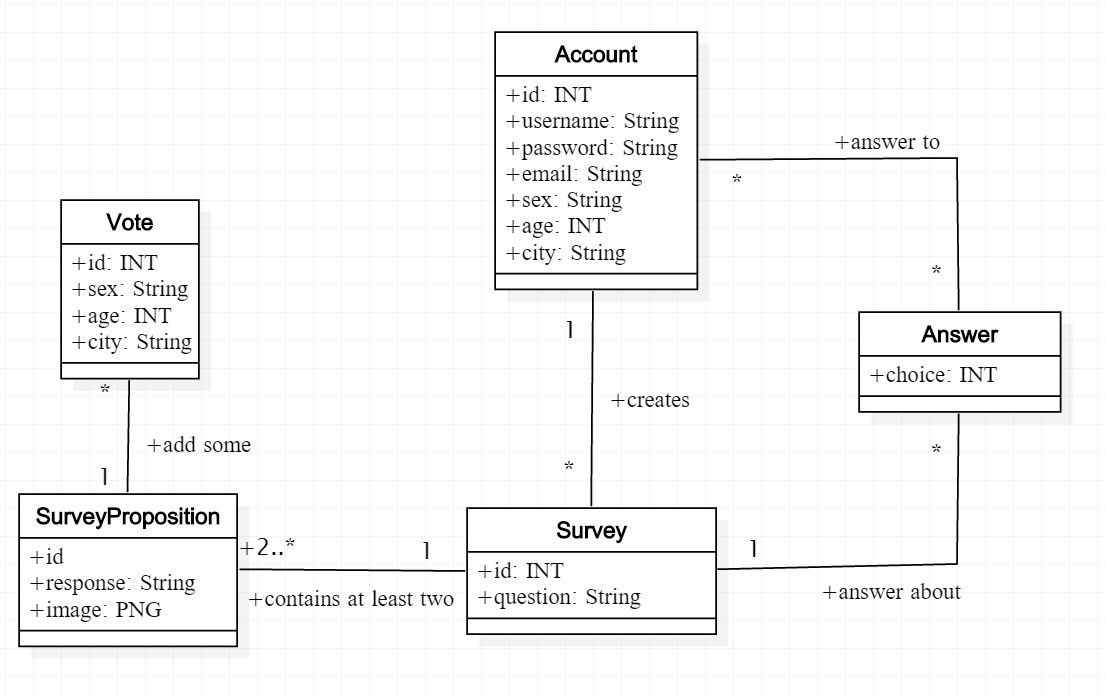


Figure 1: schéma de la bd

# Implémentation

#### Backend

Pour faciliter l’explication de l’implémentation, nous allons nous concentrer sur une table de la base de données et donner le code pour chacune des étapes. Nous choisissons la table « Account ».

Pour commencer nous avons ajouté un dossier « sql » dans le dossier « app ». Dans ce dossier « sql », nous avons créé un script, qui nous permet de créer notre base de données. Voici le code pour créer la table « Account ».

1. **DROP** **TABLE** IF EXISTS `accounts`;
2. **CREATE** **TABLE** `accounts` (
3. `id` **int**(11) NOT NULL AUTO\_INCREMENT,
4. `username` **varchar**(45)  **unique** NOT NULL,
5. `**password**` **varchar**(45) NOT NULL,
6. `email` **varchar**(45) **unique** NOT NULL,
7. `sex` **varchar**(45) NOT NULL,
8. `age` **int**(45) NOT NULL,
9. `city` **varchar**(45) NOT NULL,
10. **PRIMARY** **KEY** (`id`)
11. ) ENGINE=InnoDB **DEFAULT** CHARSET=latin1;

Nous pouvons voir que ceci correspond au schéma présent au-dessus.

Ensuite nous allons écrire le modèle « AccountModel ». Tous les modèles sont présents dans un dossier « models ». Le modèle décrit simplement la table sous forme de Scala.

1. **package** models
3. **case** **class** AccountModel(id: Option[Long], username: String, password: String, email: String,
4. sex: String, age: Int, city: String)

En Scala Option[..] permet de dire que ce paramètre n’est pas obligatoire.

Nous passons ensuite au lien entre ce modèle Scala, qui va pouvoir être utiliser dans notre code, et de la table correspondante. Ce lien va être fait par « AccountDAO ». Tous les DAO se trouvent dans le dossier DAO. Ce DAO mettra aussi à disposition des fonction permettant de récupérer, inserer, supprimer les Account dans la DB.

1. **import** models.AccountModel
2. **import** play.api.db.slick.HasDatabaseConfigProvider
4. **import** scala.concurrent.{ExecutionContext, Future}
5. **import** javax.inject.{Inject, Singleton}
6. **import** play.api.db.slick.DatabaseConfigProvider
7. **import** play.api.db.slick.HasDatabaseConfigProvider
8. **import** slick.jdbc.JdbcProfile
10. trait AccountComponent {
11. self: HasDatabaseConfigProvider[JdbcProfile] =>
13. **import** profile.api.\_
15. // This class convert the database's account table in a object-oriented entity: the Account model.
16. **class** AccountTable(tag: Tag) **extends** Table[AccountModel](tag, "ACCOUNTS")
17. {
18. def id        = column[Long]("ID", O.PrimaryKey, O.AutoInc) // Primary key, auto-incremented
19. def username = column[String]("USERNAME")
20. def password = column[String]("PASSWORD")
21. def email    = column[String]("EMAIL")
22. def sex      = column[String]("SEX")
23. def age      = column[Int]("AGE")
24. def city     = column[String]("CITY")

27. // Map the attributes with the model; the ID is optional.
28. def \* = (id.?, username, password, email, sex, age, city) <> (AccountModel.tupled, AccountModel.unapply)
29. }
31. }

34. @Singleton
35. **class** AccountDAO @Inject()(**protected** val dbConfigProvider: DatabaseConfigProvider)(implicit executionContext: ExecutionContext)
36. **extends** AccountComponent with HasDatabaseConfigProvider[JdbcProfile] {
37. **import** profile.api.\_
39. // Get the object-oriented list of courses directly from the query table.
40. val accounts = TableQuery[AccountTable]
42. /\*\* Retrieve the list of accounts sorted by username \*/
43. def list(): Future[Seq[AccountModel]] = {
44. val query = accounts.sortBy(s => s.username)
45. db.run(query.result)
46. }
48. /\*\* Retrieve an account from the username. \*/
49. def findByUsername(username: String): Future[Option[AccountModel]] =
50. db.run(accounts.filter(\_.username === username).result.headOption)
52. /\*\* Retrieve an account from the username. \*/
53. def findByEmail(email: String): Future[Option[AccountModel]] =
54. db.run(accounts.filter(\_.email === email).result.headOption)
56. /\*\* Retrieve an account from the id. \*/
57. def findById(id: Long): Future[Option[AccountModel]] =
58. db.run(accounts.filter(\_.id === id).result.headOption)
60. /\*\* Insert a new account, then return it. \*/
61. def insert(account: AccountModel): Future[AccountModel] = {
62. val insertQuery = accounts returning accounts.map(\_.id) into ((account, id) => account.copy(Some(id)))
63. db.run(insertQuery += account)
64. }
66. /\*\* Update a account, then return an integer that indicates if the account was found (1) or not (0). \*/
67. def update(id: Long, account: AccountModel): Future[Int] = {
68. val accountToUpdate: AccountModel = account.copy(Some(id))
69. db.run(accounts.filter(\_.id === id).update(accountToUpdate))
70. }
72. /\*\* Delete a account, then return an integer that indicates if the account was found (1) or not (0) \*/
73. def delete(id: Long): Future[Int] =
74. db.run(accounts.filter(\_.id === id).delete)
75. }
76. Int, city: String)

Dès que cela est fait nous pouvons passer au cœur du code, les contrôleurs. Nous avons créé un contrôleur pour chaque table, et un contrôleur en plus (ApiController) où nous avons mis les les fonctions qui utilisent plusieurs tables diffèrentes, ou qui gère la création du token. Pour AccountController voici notre code.

1. **package** controllers
3. **import** dao.AccountDAO
4. **import** javax.inject.{Inject, Singleton}
5. **import** models.AccountModel
6. **import** play.api.libs.functional.syntax.\_
7. **import** play.api.libs.json.Reads.\_
8. **import** play.api.libs.json.\_
9. **import** play.api.mvc.{AbstractController, ControllerComponents}
10. **import** utils.JWTUtils
12. **import** scala.concurrent.ExecutionContext.Implicits.global
14. @Singleton
15. **class** AccountController @Inject()(cc: ControllerComponents, accountDAO: AccountDAO) **extends** AbstractController(cc) {
17. implicit val accountToJson: Writes[AccountModel] = (
18. (JsPath \ "id").write[Option[Long]] and
19. (JsPath \ "username").write[String] and
20. (JsPath \ "password").write[String] and
21. (JsPath \ "email").write[String] and
22. (JsPath \ "sex").write[String] and
23. (JsPath \ "age").write[Int] and
24. (JsPath \ "city").write[String]
25. )(unlift(AccountModel.unapply))
27. implicit val jsonToAccount: Reads[AccountModel] = (
28. (JsPath \ "id").readNullable[Long] and
29. (JsPath \ "username").read[String] and
30. (JsPath \ "password").read[String] and
31. (JsPath \ "email").read[String] and
32. (JsPath \ "sex").read[String] and
33. (JsPath \ "age").read[Int] and
34. (JsPath \ "city").read[String]
35. )(AccountModel.apply \_)
37. def validateJson[A : Reads] = parse.json.validate(
38. \_.validate[A].asEither.left.map(e => BadRequest(JsError.toJson(e)))
39. )
41. def postAccount = Action.async(validateJson[AccountModel]) { request =>
42. val account = request.body
43. val createdAccount = accountDAO.insert(account)
44. createdAccount.map(c =>
45. Ok(
46. Json.obj(
47. "status" -> "OK",
48. "id" -> c.id,
49. "message" -> ("Account '" + c.username + "' saved.")
50. )
51. )
52. )
53. }
55. def getAccount(username: String) = Action.async { request =>
56. accountDAO.findByUsername(username).map {
57. **case** Some(c)  =>
58. val token = request.headers.get("token").fold("")(\_.toString)
59. val payload = JWTUtils.decodePayload(token).get
60. **if**(payload.equals(username)){
61. Ok(Json.toJson(c))
62. }
63. **else** {
64. NotFound(Json.obj(
65. "status" -> "Not Allowed",
66. "message" -> ("your token is not good or you don't have token in headers")
67. ))}
69. **case** None =>
70. NotFound(Json.obj(
71. "status" -> "Not Found",
72. "message" -> ("Account with username: " + username + " not found.")
73. ))
74. }
75. }
76. }

Le code se compose d’une première partie qui permet de vérifier qu’un Json correspond à la table pour bien pouvoir l’enregistrer dans la bd, et un autre qui transforme les informations d’une table en Json prêt à être envoyer au frontend.

La deuxième partie contient les fonctions utiles à la table Account. Ici nous avons postAccount et getAccount. Chaque fonction présente dans les contrôleurs, est une fonction « utilisée » par le frontend. Pour cela Il faut définir des routes (endpoints) de notre API REST. Ceci est définit dans le fichier « routes ».

1. # Routes
2. # This file defines all application routes (Higher priority routes first)
4. # Get the account with given username
5. GET     /api/accounts/:username     controllers.AccountController.getAccount(username)
7. # Post a **new** account
8. POST     /api/accounts     controllers.AccountController.postAccount()

11. # Get list of surveys created by the user
12. GET     /api/accounts/:username/surveys     controllers.SurveyController.getSurveys(username)
14. # Get survey by id
15. GET     /api/surveys/:id     controllers.SurveyController.getSurveyById(id)
17. # Post **new** survey **for** **this** user
18. POST     /api/accounts/:username/surveys     controllers.SurveyController.postSurvey(username)
20. # Delete that survey (and the associated propositions, and answers about it)
21. DELETE     /api/accounts/:username/surveys/:id     controllers.SurveyController.deleteSurvey(username, id)

24. # Get list of surveyPropositions of given survey
25. GET     /api/surveys/:id/surveyPropositions     controllers.SurveyPropositionController.getSurveyPropositions(id)
27. # Post surveyProposition **for** given survey
28. POST     /api/surveys/:id/surveyPropositions     controllers.SurveyPropositionController.postSurveyProposition(id)
30. # Post answer from user username
31. POST     /api/accounts/:username/answers    controllers.AnswerController.postAnswer(username)

34. # Get list statistics **for** asked survey
35. GET     /api/surveys/:id/stats     controllers.ApiController.getStats(id)
37. # Get the token of account with given username
38. GET     /api/token     controllers.ApiController.getToken(username: Option[String], password: Option[String])
40. # Get a question that the user didn't have responded yet randomly
41. GET     /api/accounts/:username/newquestion     controllers.ApiController.getNewQuestion(username)

Le dernier point important de l’implémentation du backend est celui du Token. Pour lors de la connexion à l’application l’utilisateur va faire appel à la méthode getToken, à l’aide du endpoint « /api/token ». ce endpoint requiert des query paramètres qui sont l’username, et le password.

1. def getToken(username: Option[String], password: Option[String]) = Action.async {
2. accountDAO.findByUsername(username.fold("")(\_.toString)).map {
3. **case** Some(c) =>
4. **if** (c.password.equals(password.fold("")(\_.toString))) {
5. Ok(Json.toJson(JWTUtils.createToken(c.username)))
6. }
7. **else** {
8. NotFound(Json.obj(
9. "status" -> "Not Allowed",
10. "message" -> ("your your username doesn't match with your password, or you forget one of those")
11. ))
12. }
14. **case** None =>
15. NotFound(Json.obj(
16. "status" -> "Not Found",
17. "message" -> ("You don't have query parameters username and password.")
18. ))
19. }
20. }

Cette fonction vérifie que le password et le username donnés sont correspondant, puis on utilise la fonction createToken présent dans JWTUtils). Cette fonction prend en paramètre une string et renvoie un token. Ce token envoyé au frontend sera stocké en SessionStorage, et utilisé en Header des appels à l’API qui demandent un token.

Dans les fonctions demandant un token, ce token va être décodé :

val payload = JWTUtils.decodePayload(token).get

et on vérifiera que le payload (qui sera le username) correspond bien au username.

#### Frontend

Pour le frontend nous avons donc commencé par récupérer un template bootstrap, sur <https://startbootstrap.com/>. Ensuite un gros travail de nettoyage a eu lieu. Il aura fallu opérer à des modifications et des ajouts de html, css et js. Le plus important pour ce frontend, est le script apiCallController.js. C’est de ce script que nous appelons l’API à l’aide de XMLHttpRequest(). Voici un exemple de code d’appel à l’API :

1. **function** postAccount() {
2. **var** username = document.getElementById("usernameRegister").value;
3. **var** password = document.getElementById("passwordRegister").value;
4. **var** email = document.getElementById("email").value;
5. **var** age = document.getElementById("age").value;
6. **var** city = document.getElementById("city").value;
7. **var** sex = "male";
8. **if**(document.getElementById("radio2").checked) {
9. sex = "female";
10. }
12. **var** accountJson = '{ "username":"' + username + '",' +
13. '"password":"' + password + '",'+
14. '"email":"' + email + '",'+
15. '"sex":"' + sex + '",'+
16. '"age":' + age + ','+
17. '"city":"' + city + '"'+
18. '}';
19. **var** xhttp = **new** XMLHttpRequest();
20. xhttp.onprogress = waitForApiCallResponse(event);
21. xhttp.open("POST", urlApi + "/accounts", **true**);
22. xhttp.setRequestHeader("Content-type", "application/json");
23. xhttp.setRequestHeader("Access-Control-Allow-Origin", "\*");
24. xhttp.send(accountJson);
25. xhttp.onreadystatechange = **function**() {
26. **if** (xhttp.readyState == XMLHttpRequest.DONE) {
27. **if**(xhttp.status != 200){
28. alert("This account already exists!");
29. }
30. **else** {
31. alert("Account created with success!");
32. // Get the modal
33. **var** modalRegister = document.getElementById("myModalRegister");
34. modalRegister.style.display = "none";
36. }
37. }
38. }
40. **return** **false**;
41. }

Cette fonction est appelée lors de l’ajout d’un compte. Elle utilise le endpoint « POST /api/accounts ». Ce endpoint a besoin d’un body qui correspond à un Account. Nous le créons « accountJson », et l’envoyant a l’aide de xhttp.send(accountJson). Nous attendons ensuite la réponse du backend pour réagir en conséquence (à l’aide de xhhtp.onreadystatechange).

# Problèmes rencontrés

Ce projet était sujet à un grand nombre de problèmes pour trois raisons principales :

* Ne n’avions pas une grande expérience en Scala. C’est un langage exigeant, qui facilite grandement l’écriture de code lorsque nous avons de bonnes bases, mais pour des novices c’est un langage qui ajoute certaines difficultés.
* Play, est un framework relativement accessible, et qui ressemble un petit peut à springboot (framework déjà utilisé dans un autre cours), mais apprendre un framework pour un petit projet, et tout cela en 1 ou 2 semaines maximum est une tâche ardue, et nous passons à côté de plusieurs points spécifiques qui nous font nous perdre parfois.
* Nous ne sommes pas des spécialistes web, (HTML, css et JS).

Les problèmes rencontrés ont été :

* Le temps de développement du frontend a été bien plus long que prévu. Nous pensions qu’en prenant un template bootstrap et en le modifiant, le temps de développement serait réduit. Malheureusement la modification d’un modèle bootstrap amène des soucis assez rapidement. Les éléments html ne sont plus du tout au bon endroit, et l’ajout de nouveaux éléments demande un temps considérable, pour analyser les très longues feuilles de style (CSS) pour y trouver ce dont on a besoin.

L’objectif que nous nous étions fixé d’avoir une application adaptative, a été difficile à mettre en place, et a amené des soucis visuels à l’application.

* Pour le backend le souci principal a été la mauvaise compréhension de « map ». Lorsque nous voulions imbriquer plusieurs appels à la base de données, nous devions faire appel à map, à l’intérieur d’autre map. Ceci amenait des soucis, et nous avons mis tu temps à nous rendre compte qu’il suffisait d’appeler des flatMap.
* L’utilisation du token a pris un temps non-négligeable à mettre en place. Les librairies en Scala pour opérer cela était des petits projets open-source mal documentés, et parfois bugés. De plus il n’y en avait que 2 ou 3 différents. Pour faire fonctionner la librairie trouvée (Authentikate) nous avons dû modifier en partie le code.
* Un autre souci rencontré, a été pour stocker les images. Nous sommes partie sur le stockage en base de données de l’image sous forme de Array[Byte]. Les requêtes depuis le frontend contenaient, donc, un body très long et volumineux. Ce volume était trop important et il était bloqué par le backend. Nous n’avons pas tout de suite trouvé l’înformation, qui disait que ce problème se réglait en ajoutant : « play.http.parser.maxMemoryBuffer= \_\_\_\_\_\_ » à « application.conf »
* Au final, le principal problème a été que ce projet était trop ambitieux. La charge de travail qui devait y être allouée a été fortement dépassée. La méconnaissance des technologies n’a pas été prise en compte lors de l’évaluation du temps nécessaire.

# Conclusion

Ce projet a été mené à terme. C’était un moyen très efficace pour appliquer Scala dans un framework utilisé dans le monde professionnel. Play framework est un outil puissant, et au final relativement facile à prendre en main. Bien que trop ambitieux le projet est terminé, la phase de tests a par contre été abandonnée par manque de temps.

L’utilisation de Play peut être une très bonne alternative à un backend en Node.js par exemple. Avec Play notre code backend est plus ordonné, lisible et évolutif. Son utilisation en temps qu’API REST permet une liberté de choix du frontend appréciable. Si le temps l’avait permis, le développement d’un frontend à l’aide d’une technologie récente telle que React.js ou Angular.js, aurait donné une application Web propre et facile à maintenir.