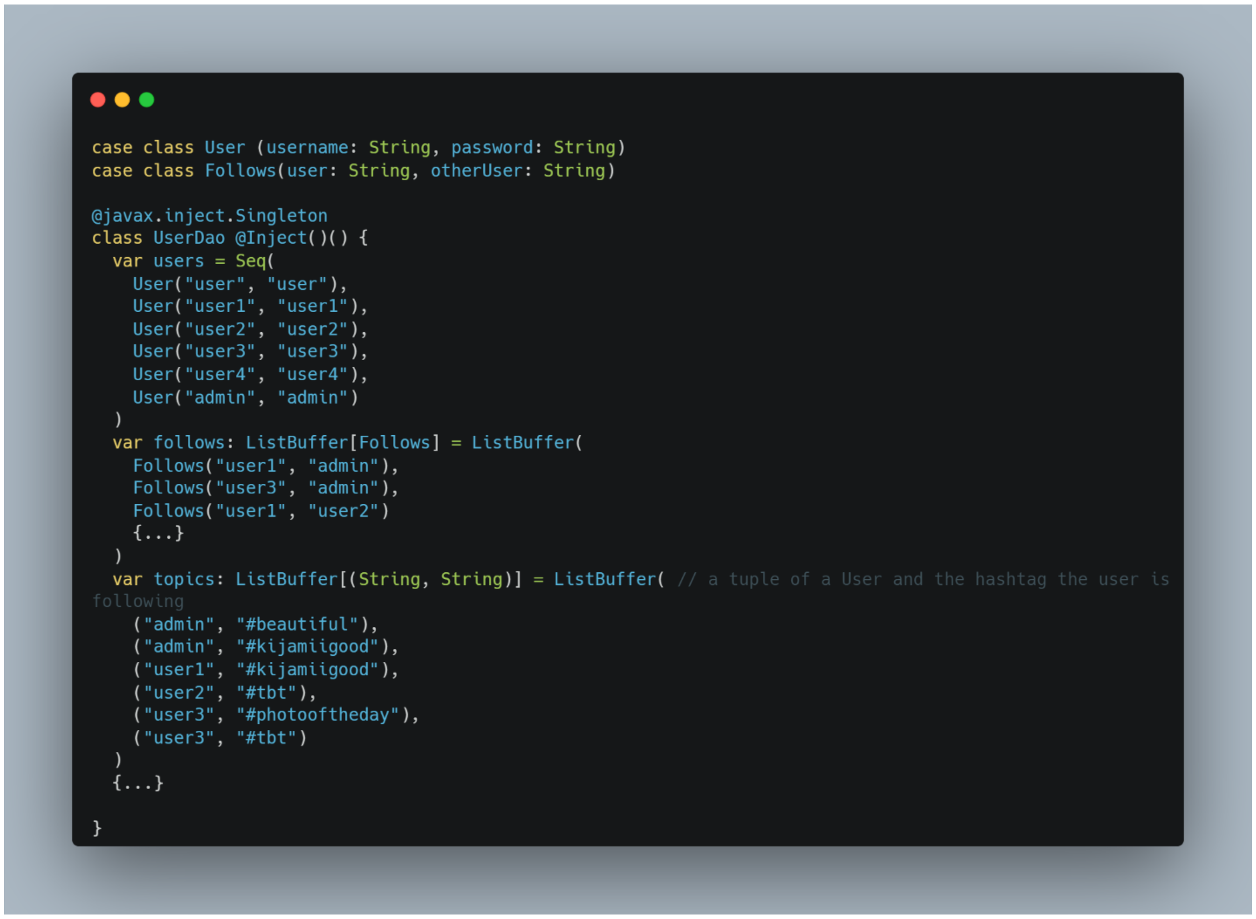
MVC

WITH SCALA PLAY

Amadou Sarjo Jallow
Vrije Universiteit Brussels
Software Architectures

CONTENT

userDao



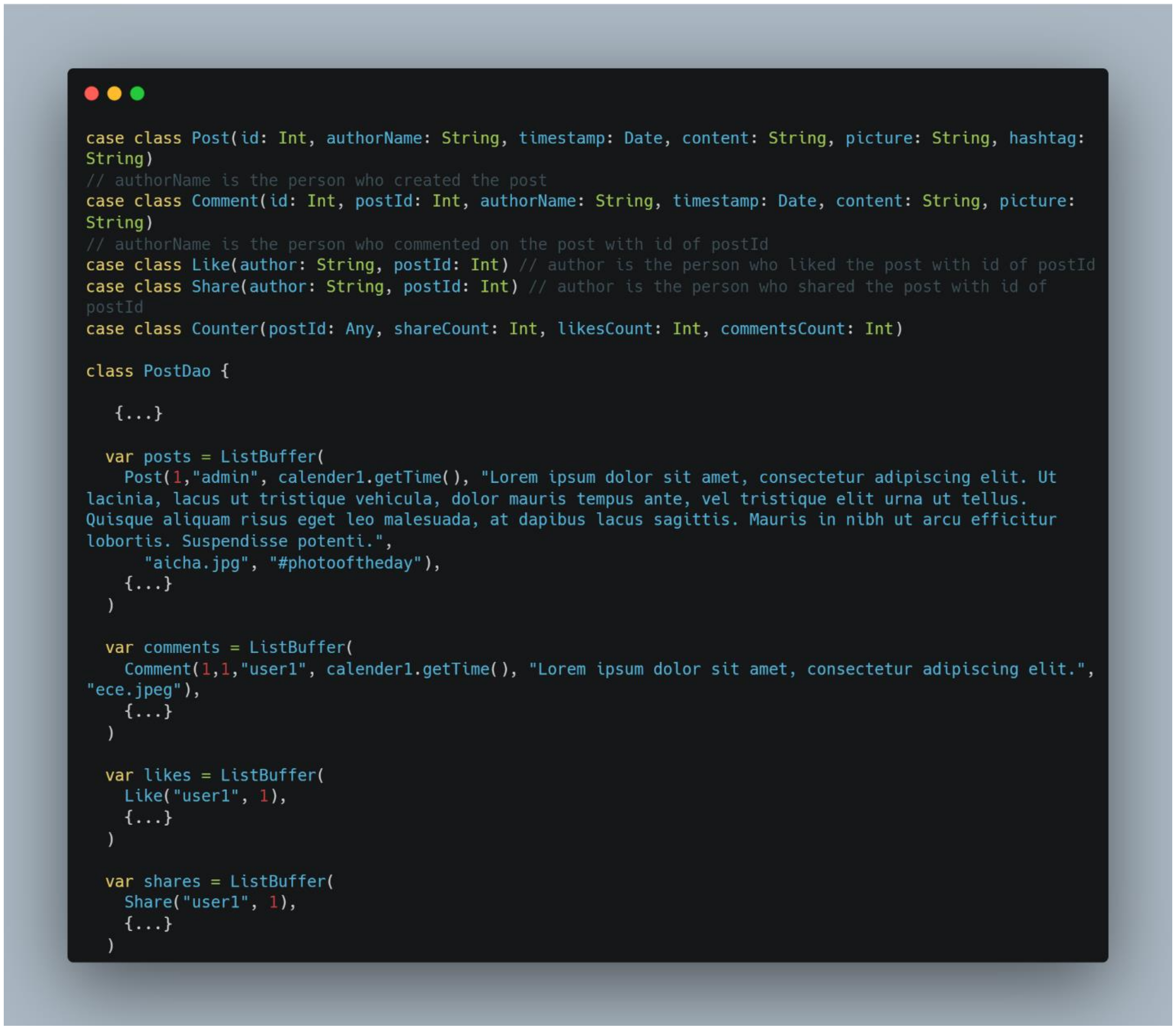
```
case class User (username: String, password: String)
case class Follows(user: String, otherUser: String)

@javax.inject.Singleton
class UserDao @Inject()() {
  var users = Seq(
    User("user", "user"),
    User("user1", "user1"),
    User("user2", "user2"),
    User("user3", "user3"),
    User("user4", "user4"),
    User("admin", "admin")
  )
  var follows: ListBuffer[Follows] = ListBuffer(
    Follows("user1", "admin"),
    Follows("user3", "admin"),
    Follows("user1", "user2")
    {...}
  )
  var topics: ListBuffer[(String, String)] = ListBuffer( // a tuple of a User and the hashtag the user is 
following
    ("admin", "#beautiful"),
    ("admin", "#kijamiigood"),
    ("user1", "#kijamiigood"),
    ("user2", "#tbt"),
    ("user3", "#photooftheday"),
    ("user3", "#tbt")
  )
  {...}

}
```

A UserDao object has variables, users, which is a sequence of User case classes , follows, which is a ListBuffer of Follows, were the first parameter(user) follows the second parameter(otherUser) and topics, a ListBuffer of tuple pairs (username, hashtag) where each tiple describes a user and the hashtag the user is subscribed to.

postDao



```scala
case class Post(id: Int, authorName: String, timestamp: Date, content: String, picture: String, hashtag:
String)
// authorName is the person who created the post
case class Comment(id: Int, postId: Int, authorName: String, timestamp: Date, content: String, picture:
String)
// authorName is the person who commented on the post with id of postId
case class Like(author: String, postId: Int) // author is the person who liked the post with id of postId
case class Share(author: String, postId: Int) // author is the person who shared the post with id of
postId
case class Counter(postId: Any, shareCount: Int, likesCount: Int, commentsCount: Int)

class PostDao {

   {...}

  var posts = ListBuffer(
    Post(1,"admin", calender1.getTime(), "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut
lacinia, lacus ut tristique vehicula, dolor mauris tempus ante, vel tristique elit urna ut tellus.
Quisque aliquam risus eget leo malesuada, at dapibus lacus sagittis. Mauris in nibh ut arcu efficitur
lobortis. Suspendisse potenti.",
      "aicha.jpg", "#photooftheday"),
    {...}
  )

  var comments = ListBuffer(
    Comment(1,1,"user1", calender1.getTime(), "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
"ece.jpeg"),
    {...}
  )

  var likes = ListBuffer(
    Like("user1", 1),
    {...}
  )

  var shares = ListBuffer(
    Share("user1", 1),
    {...}
  )
```

A PostDao object has variables:

posts, a ListBuffer of Post objects.

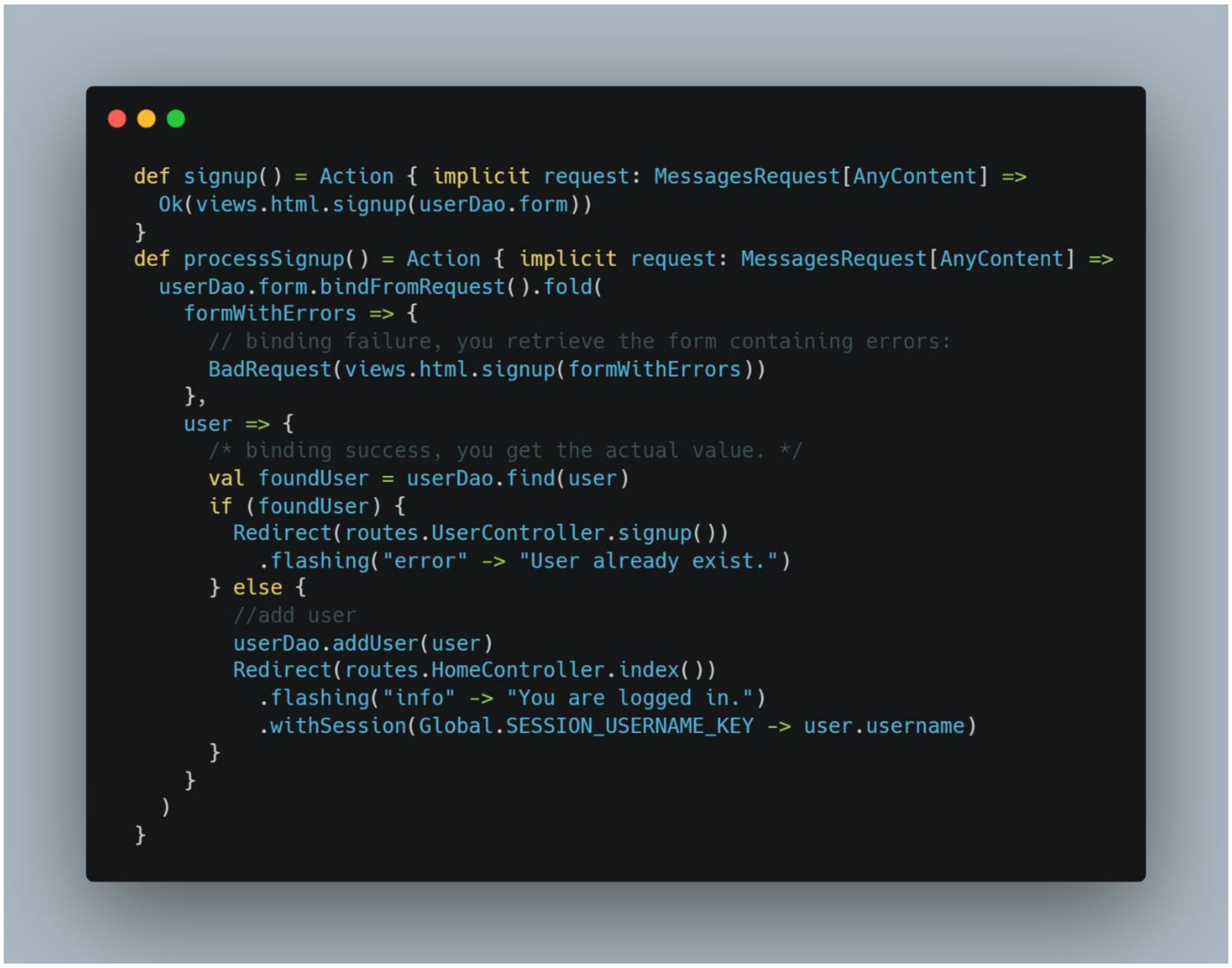
comments, a ListBuffer of Comment objects.

likes, a ListBuffer of Like objects, where each Like instance defines a user and the post liked.

Shares, a ListBuffer of Share objects, where an instance defines a user and the post shared by the user.

Controller components

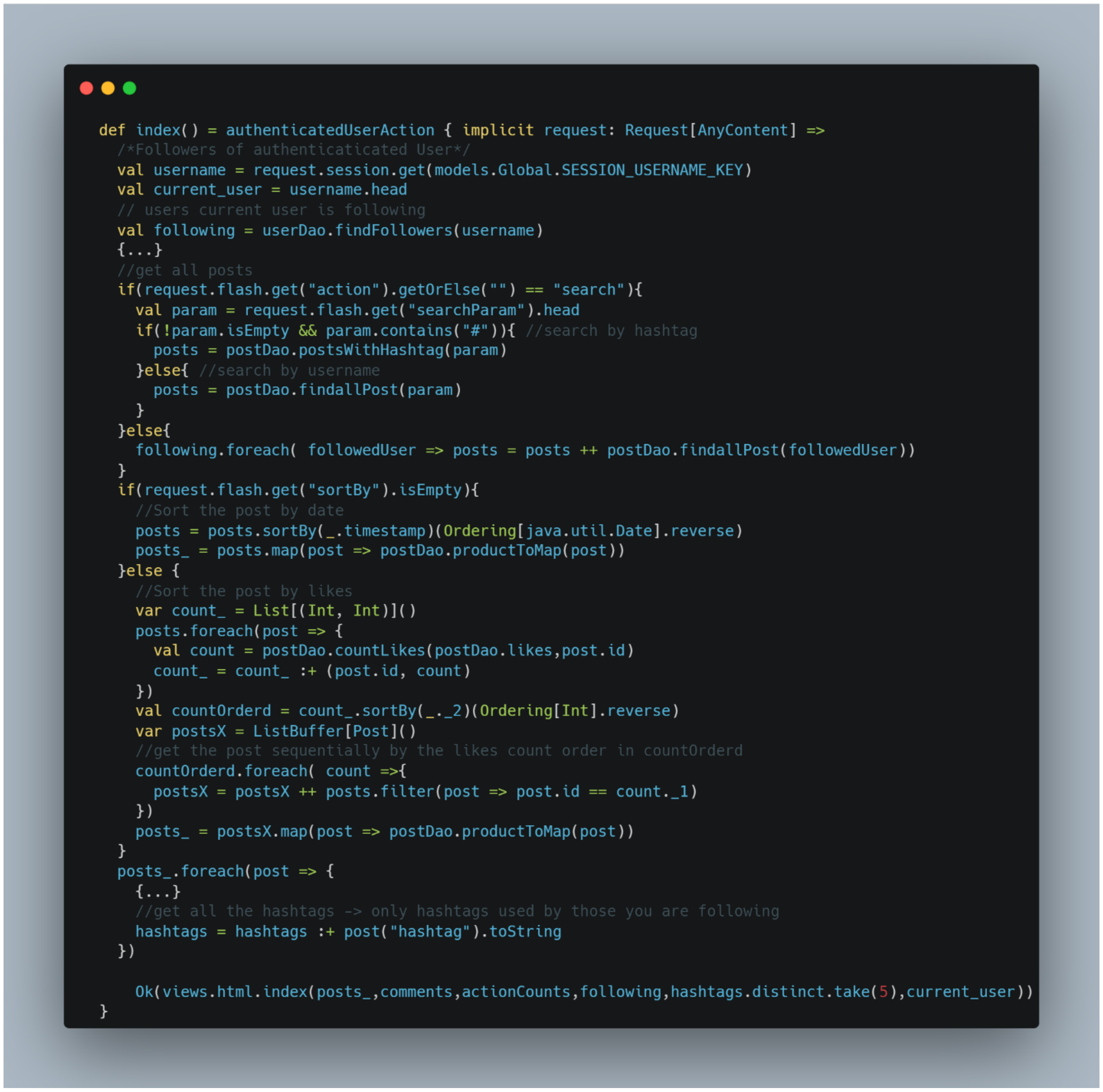
Signup



```
def signup() = Action { implicit request: MessagesRequest[AnyContent] =>
  Ok(views.html.signup(userDao.form))
}
def processSignup() = Action { implicit request: MessagesRequest[AnyContent] =>
  userDao.form.bindFromRequest().fold(
    formWithErrors => {
      // binding failure, you retrieve the form containing errors:
      BadRequest(views.html.signup(formWithErrors))
    },
    user => {
      /* binding success, you get the actual value. */
      val foundUser = userDao.find(user)
      if (foundUser) {
        Redirect(routes.UserController.signup())
          .flashing("error" -> "User already exist.")
      } else {
        //add user
        userDao.addUser(user)
        Redirect(routes.HomeController.index())
          .flashing("info" -> "You are logged in.")
          .withSession(Global.SESSION_USERNAME_KEY -> user.username)
      }
    }
  )
}
```

The signup method of the UserController calls the signup view passing a form validation tuple. When the user fills and submit the form, the processSignup method is called which checks if the user exist on the userDao.users sequence. If the user exists, it redirects to the signup view page again flashing an error message. If the user does not exist, then the new user is added to the userDao.users sequence and the user is redirected to the index(home) page of the HomeComtroller and a SESSION_USERNAME_KEY is started.

Index



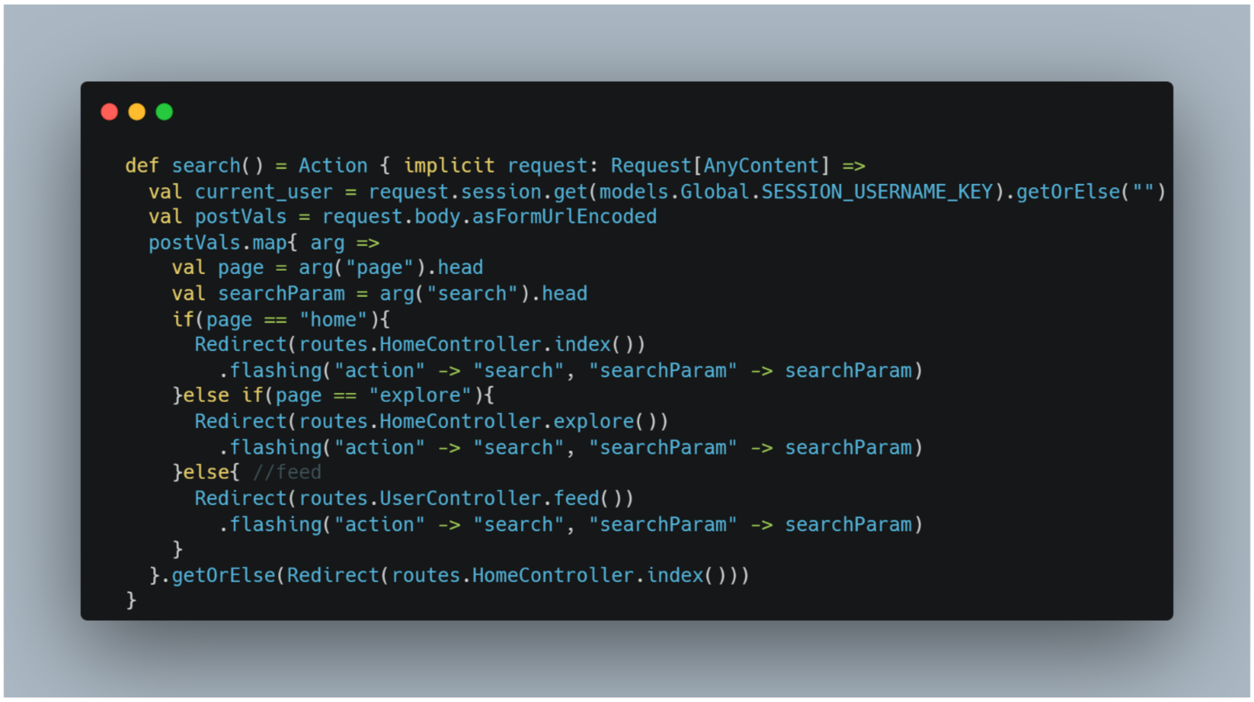
```
def index() = authenticatedUserAction { implicit request: Request[AnyContent] =>
  /*Followers of authenticaticated User*/
  val username = request.session.get(models.Global.SESSION_USERNAME_KEY)
  val current_user = username.head
  // users current user is following
  val following = userDao.findFollowers(username)
  {...}
  //get all posts
  if(request.flash.get("action").getOrElse("") == "search"){
    val param = request.flash.get("searchParam").head
    if(!param.isEmpty && param.contains("#")){ //search by hashtag
      posts = postDao.postsWithHashtag(param)
    }else{ //search by username
      posts = postDao.findallPost(param)
    }
  }else{
    following.foreach( followedUser => posts = posts ++ postDao.findallPost(followedUser))
  }
  if(request.flash.get("sortBy").isEmpty){
    //Sort the post by date
    posts = posts.sortBy(_.timestamp)(Ordering[java.util.Date].reverse)
    posts_ = posts.map(post => postDao.productToMap(post))
  }else {
    //Sort the post by likes
    var count_ = List[(Int, Int)]()
    posts.foreach(post => {
      val count = postDao.countLikes(postDao.likes,post.id)
      count_ = count_ :+ (post.id, count)
    })
    val countOrderd = count_.sortBy(_._2)(Ordering[Int].reverse)
    var postsX = ListBuffer[Post]()
    //get the post sequentially by the likes count order in countOrderd
    countOrderd.foreach( count =>{
      postsX = postsX ++ posts.filter(post => post.id == count._1)
    })
    posts_ = postsX.map(post => postDao.productToMap(post))
  }
  posts_.foreach(post => {
    {...}
    //get all the hashtags -> only hashtags used by those you are following
    hashtags = hashtags :+ post("hashtag").toString
  })

    Ok(views.html.index(posts_,comments,actionCounts,following,hashtags.distinct.take(5),current_user))
}
```

The index method of the HomeController is simply the method that calls the index or home view if a user is authenticated successfully. It first finds all the users that the current user is following. The “action” and “searchParam” flash values are only set when the request is a Redirect from the search() method, where in we find all post based on the “searchParam”(username or #hashtag). If that is not the case, then we find all the posts by users the current user is following. By default, the posts are sorted by date unless when the “sortBy” flash value is set, in which case, the posts are sorted by likes.The sortByLikes() method is the method which Redirects to the index() method and sets the “sortBy” flash key to value “likes”.

Looping through the posts, the associated comments, likes, shares and hashtags are retrieved. These results are sent to the index view to build the home page.

search

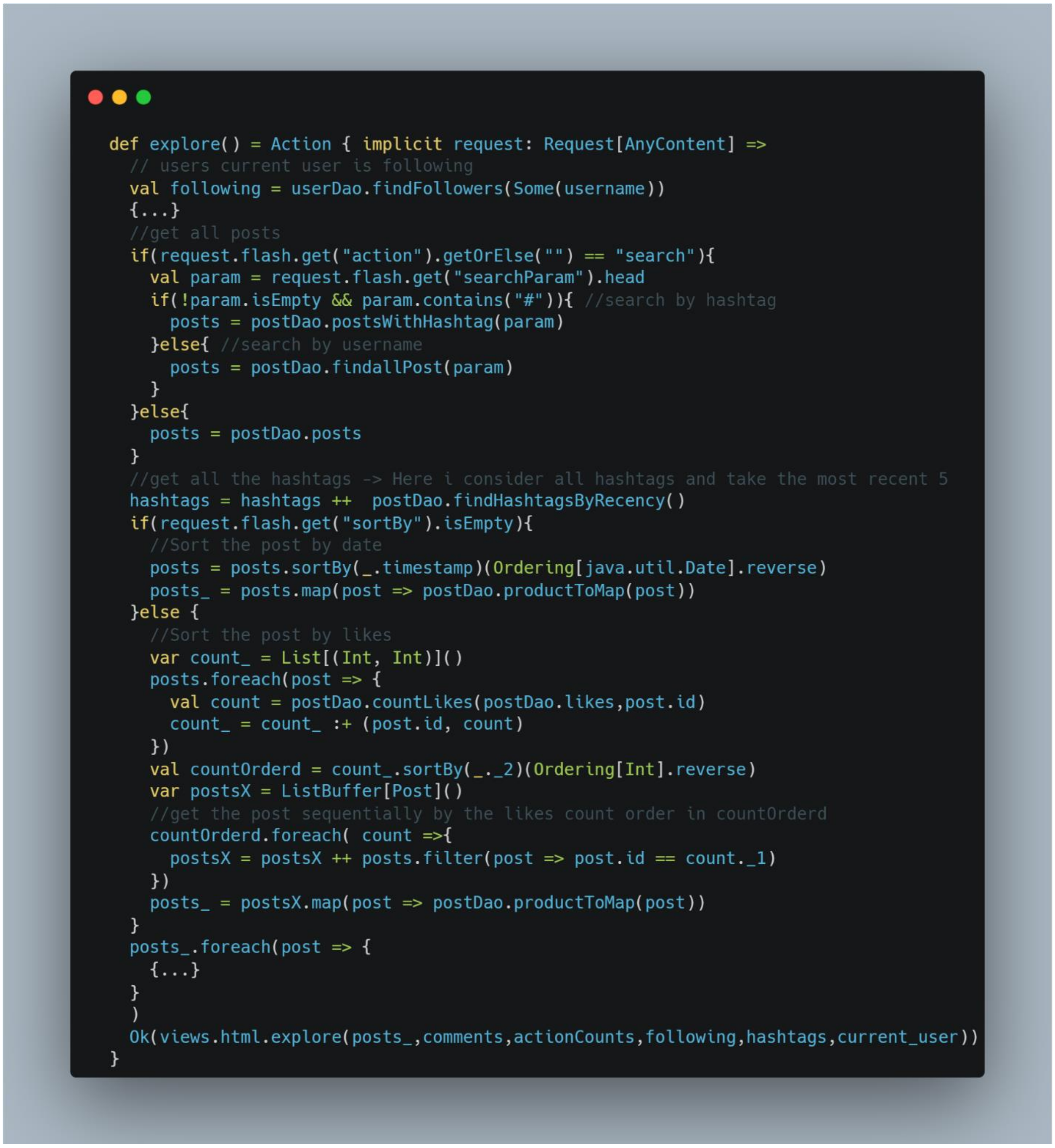


```scala
def search() = Action { implicit request: Request[AnyContent] =>
  val current_user = request.session.get(models.Global.SESSION_USERNAME_KEY).getOrElse("")
  val postVals = request.body.asFormUrlEncoded
  postVals.map{ arg =>
    val page = arg("page").head
    val searchParam = arg("search").head
    if(page == "home"){
      Redirect(routes.HomeController.index())
        .flashing("action" -> "search", "searchParam" -> searchParam)
    }else if(page == "explore"){
      Redirect(routes.HomeController.explore())
        .flashing("action" -> "search", "searchParam" -> searchParam)
    }else{ //feed
      Redirect(routes.UserController.feed())
        .flashing("action" -> "search", "searchParam" -> searchParam)
    }
  }.getOrElse(Redirect(routes.HomeController.index()))
}
```

This method of the HomeController is called in the home, explore and feeds pages. Each time a post request is made to the search() method, a form is submitted, which contains, the calling page name and the searchParam value. Based on these two, the appropriate Redirect is made, flashing “action” and “searchParam” values. The rest is handled by the called methods.

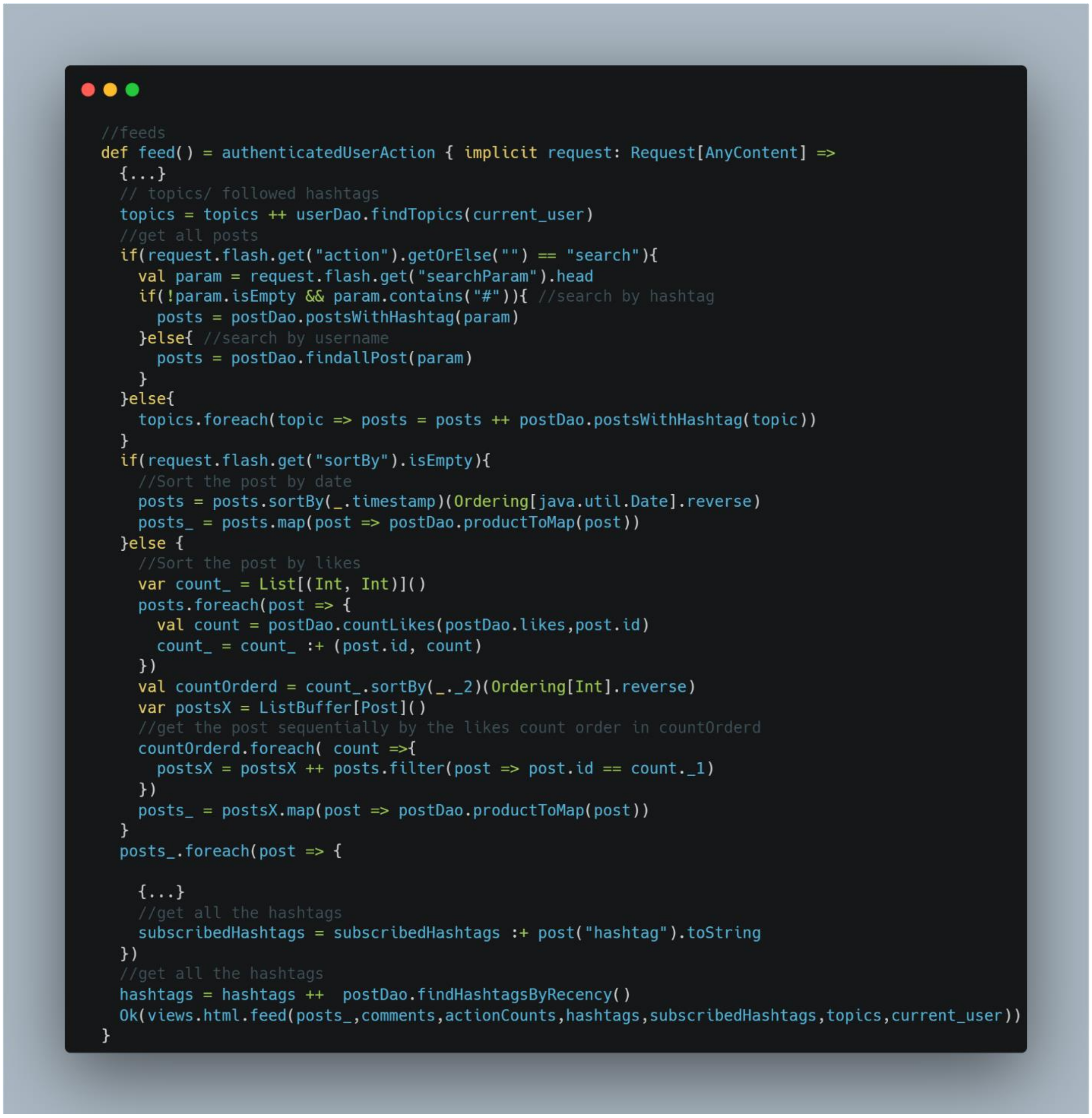
explore

The explore() method of the HomeController is simply the method that calls the explore view. This view retrieves all posts in the system. The “action” and “searchParam” flash values are only set when the request is a Redirect from the search() method, where in we find all post based on the “searchParam”(username or #hashtag). If that is not the case, then we simply find all posts. By default, the posts are sorted by date unless when the “sortBy” flash value is set, in which case, the posts are sorted by likes. Looping through the posts, the associated comments, likes, shares and hashtags are retrieved. This method is like the index method except that here, the hashtags are the ordered by recency and we return only 5 hashtags that are distinct. In short, the last 5 used hashtags. Where in , the index method only considers hashtags from the posts of the users you are following. These results are sent to the explore view to build the explore page.



```scala
def explore() = Action { implicit request: Request[AnyContent] =>
  // users current user is following
  val following = userDao.findFollowers(Some(username))
  {...}
  //get all posts
  if(request.flash.get("action").getOrElse("") == "search"){
    val param = request.flash.get("searchParam").head
    if(!param.isEmpty && param.contains("#")){ //search by hashtag
      posts = postDao.postsWithHashtag(param)
    }else{ //search by username
      posts = postDao.findallPost(param)
    }
  }else{
    posts = postDao.posts
  }
  //get all the hashtags -> Here i consider all hashtags and take the most recent 5
  hashtags = hashtags ++  postDao.findHashtagsByRecency()
  if(request.flash.get("sortBy").isEmpty){
    //Sort the post by date
    posts = posts.sortBy(_.timestamp)(Ordering[java.util.Date].reverse)
    posts_ = posts.map(post => postDao.productToMap(post))
  }else {
    //Sort the post by likes
    var count_ = List[(Int, Int)]()
    posts.foreach(post => {
      val count = postDao.countLikes(postDao.likes,post.id)
      count_ = count_ :+ (post.id, count)
    })
    val countOrderd = count_.sortBy(_._2)(Ordering[Int].reverse)
    var postsX = ListBuffer[Post]()
    //get the post sequentially by the likes count order in countOrderd
    countOrderd.foreach( count =>{
      postsX = postsX ++ posts.filter(post => post.id == count._1)
    })
    posts_ = postsX.map(post => postDao.productToMap(post))
  }
  posts_.foreach(post => {
    {...}
  }
  )
  Ok(views.html.explore(posts_,comments,actionCounts,following,hashtags,current_user))
}
```

feed

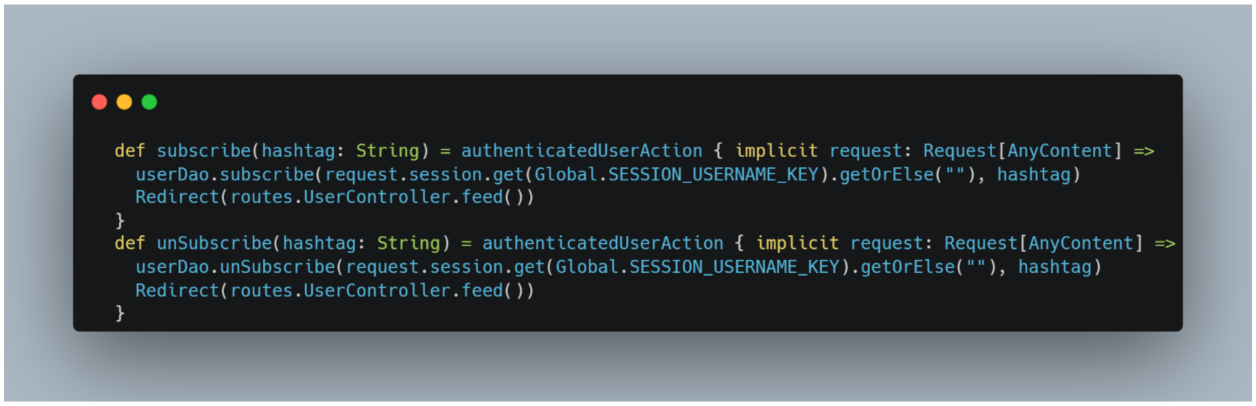


```
//feeds
def feed() = authenticatedUserAction { implicit request: Request[AnyContent] =>
  {...}
  // topics/ followed hashtags
  topics = topics ++ userDao.findTopics(current_user)
  //get all posts
  if(request.flash.get("action").getOrElse("") == "search"){
    val param = request.flash.get("searchParam").head
    if(!param.isEmpty && param.contains("#")){ //search by hashtag
      posts = postDao.postsWithHashtag(param)
    }else{ //search by username
      posts = postDao.findallPost(param)
    }
  }else{
    topics.foreach(topic => posts = posts ++ postDao.postsWithHashtag(topic))
  }
  if(request.flash.get("sortBy").isEmpty){
    //Sort the post by date
    posts = posts.sortBy(_.timestamp)(Ordering[java.util.Date].reverse)
    posts_ = posts.map(post => postDao.productToMap(post))
  }else {
    //Sort the post by likes
    var count_ = List[(Int, Int)]()
    posts.foreach(post => {
      val count = postDao.countLikes(postDao.likes,post.id)
      count_ = count_ :+ (post.id, count)
    })
    val countOrderd = count_.sortBy(_._2)(Ordering[Int].reverse)
    var postsX = ListBuffer[Post]()
    //get the post sequentially by the likes count order in countOrderd
    countOrderd.foreach( count =>{
      postsX = postsX ++ posts.filter(post => post.id == count._1)
    })
    posts_ = postsX.map(post => postDao.productToMap(post))
  }
  posts_.foreach(post => {

    {...}
    //get all the hashtags
    subscribedHashtags = subscribedHashtags :+ post("hashtag").toString
  })
  //get all the hashtags
  hashtags = hashtags ++  postDao.findHashtagsByRecency()
  Ok(views.html.feed(posts_,comments,actionCounts,hashtags,subscribedHashtags,topics,current_user))
}
```

The feed() method of the UserController is like the index() and explore() methods with a little twist. Here, we first retrieve all the topics or hashtags that the current user is following and then find all posts with those hashtags. We still do pass the hashtags along too so that a user can subscribe to other hashtags or unsubscribe to hashtags if needed.

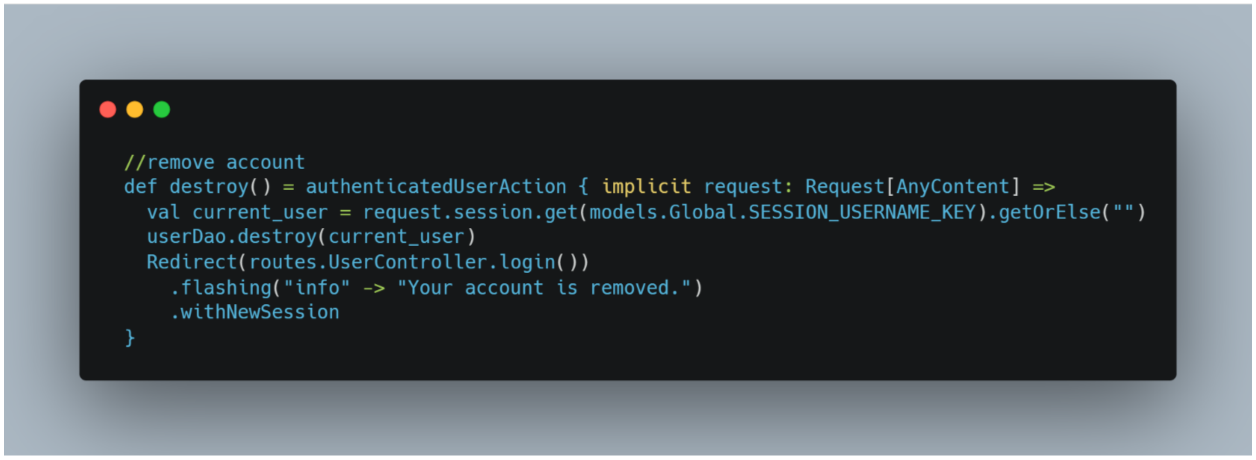
subscribe and unSubscribe



```scala
def subscribe(hashtag: String) = authenticatedUserAction { implicit request: Request[AnyContent] =>
  userDao.subscribe(request.session.get(Global.SESSION_USERNAME_KEY).getOrElse(""), hashtag)
  Redirect(routes.UserController.feed())
}
def unSubscribe(hashtag: String) = authenticatedUserAction { implicit request: Request[AnyContent] =>
  userDao.unSubscribe(request.session.get(Global.SESSION_USERNAME_KEY).getOrElse(""), hashtag)
  Redirect(routes.UserController.feed())
}
```

These two methods allow users to subscribe and unsubscribe to hashtags or topics in the feeds view.

destroy

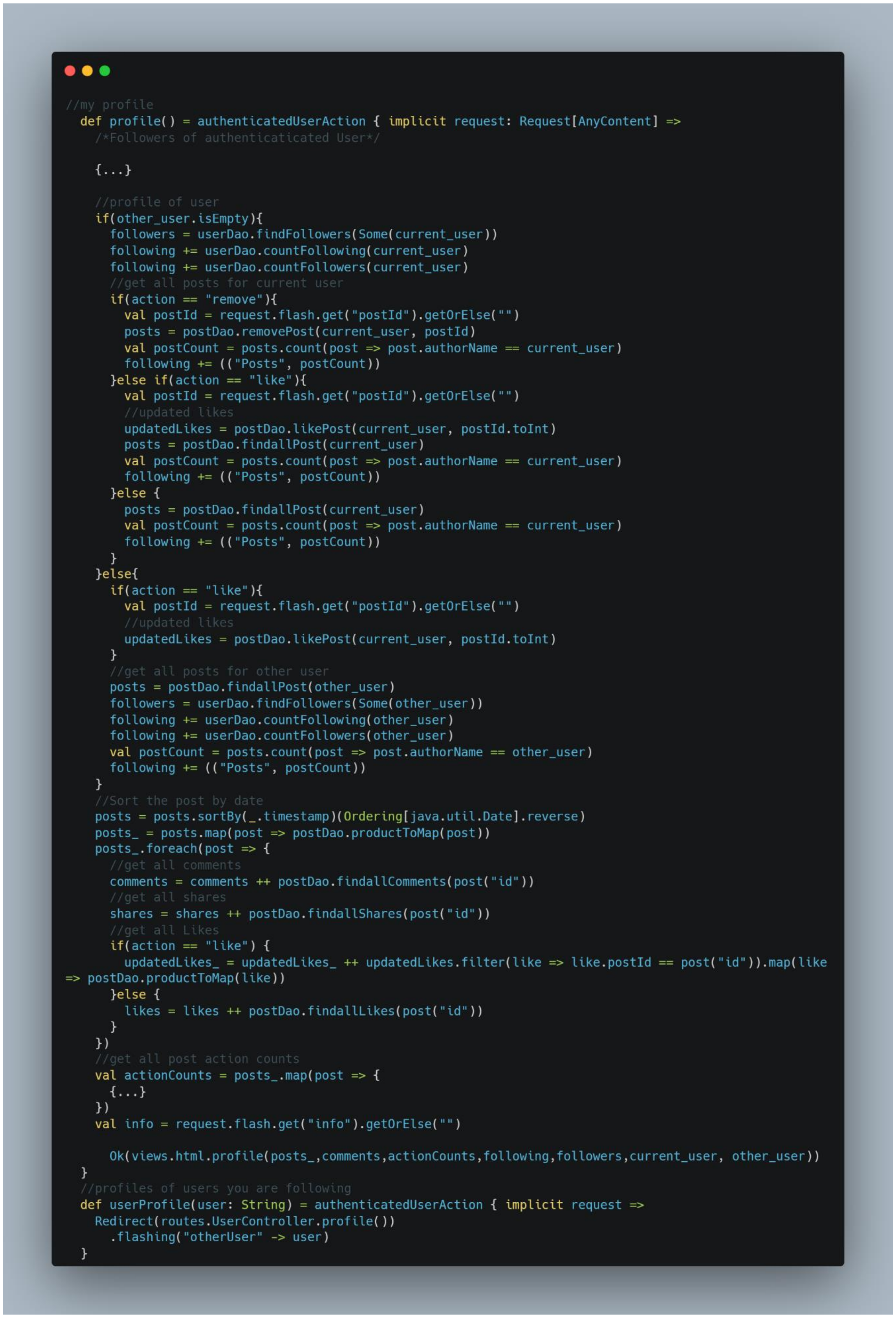


```scala
//remove account
def destroy() = authenticatedUserAction { implicit request: Request[AnyContent] =>
  val current_user = request.session.get(models.Global.SESSION_USERNAME_KEY).getOrElse("")
  userDao.destroy(current_user)
  Redirect(routes.UserController.login())
    .flashing("info" -> "Your account is removed.")
    .withNewSession
}
```

This method is called when the current user wants to delete their account. The user is removed and is then redirected to the login view. It can only be called in the profile view.

profile

This method is called to view the profile of both the current user and other users in the system which have little differences in their UI but basically similar. The current user can delete their own posts and delete their account whilst when viewing the profile of other users, you can only follow or unfollow their account and like, share or comment on their posts.



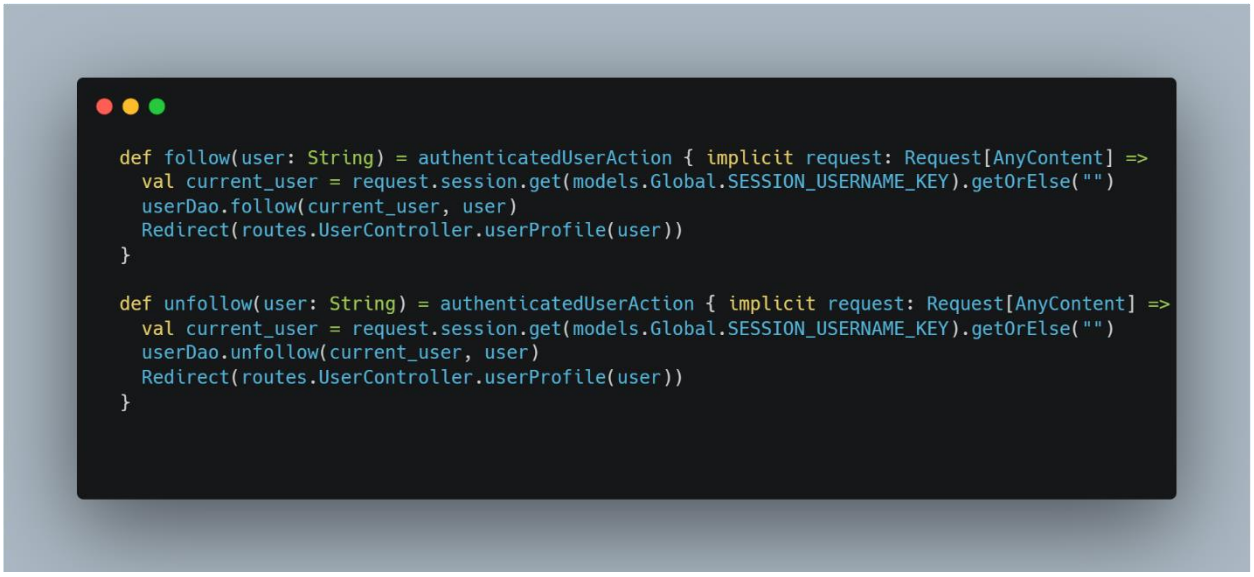
```scala
//my profile
  def profile() = authenticatedUserAction { implicit request: Request[AnyContent] =>
    /*Followers of authenticaticated User*/

    {...}

    //profile of user
    if(other_user.isEmpty){
      followers = userDao.findFollowers(Some(current_user))
      following += userDao.countFollowing(current_user)
      following += userDao.countFollowers(current_user)
      //get all posts for current user
      if(action == "remove"){
        val postId = request.flash.get("postId").getOrElse("")
        posts = postDao.removePost(current_user, postId)
        val postCount = posts.count(post => post.authorName == current_user)
        following += (("Posts", postCount))
      }else if(action == "like"){
        val postId = request.flash.get("postId").getOrElse("")
        //updated likes
        updatedLikes = postDao.likePost(current_user, postId.toInt)
        posts = postDao.findallPost(current_user)
        val postCount = posts.count(post => post.authorName == current_user)
        following += (("Posts", postCount))
      }else {
        posts = postDao.findallPost(current_user)
        val postCount = posts.count(post => post.authorName == current_user)
        following += (("Posts", postCount))
      }
    }else{
      if(action == "like"){
        val postId = request.flash.get("postId").getOrElse("")
        //updated likes
        updatedLikes = postDao.likePost(current_user, postId.toInt)
      }
      //get all posts for other user
      posts = postDao.findallPost(other_user)
      followers = userDao.findFollowers(Some(other_user))
      following += userDao.countFollowing(other_user)
      following += userDao.countFollowers(other_user)
      val postCount = posts.count(post => post.authorName == other_user)
      following += (("Posts", postCount))
    }
    //Sort the post by date
    posts = posts.sortBy(_.timestamp)(Ordering[java.util.Date].reverse)
    posts_ = posts.map(post => postDao.productToMap(post))
    posts_.foreach(post => {
      //get all comments
      comments = comments ++ postDao.findallComments(post("id"))
      //get all shares
      shares = shares ++ postDao.findallShares(post("id"))
      //get all Likes
      if(action == "like") {
        updatedLikes_ = updatedLikes_ ++ updatedLikes.filter(like => like.postId == post("id")).map(like
=> postDao.productToMap(like))
      }else {
        likes = likes ++ postDao.findallLikes(post("id"))
      }
    })
    //get all post action counts
    val actionCounts = posts_.map(post => {
      {...}
    })
    val info = request.flash.get("info").getOrElse("")

      Ok(views.html.profile(posts_,comments,actionCounts,following,followers,current_user, other_user))
  }
  //profiles of users you are following
  def userProfile(user: String) = authenticatedUserAction { implicit request =>
    Redirect(routes.UserController.profile())
      .flashing("otherUser" -> user)
  }
```

follow and unFollow

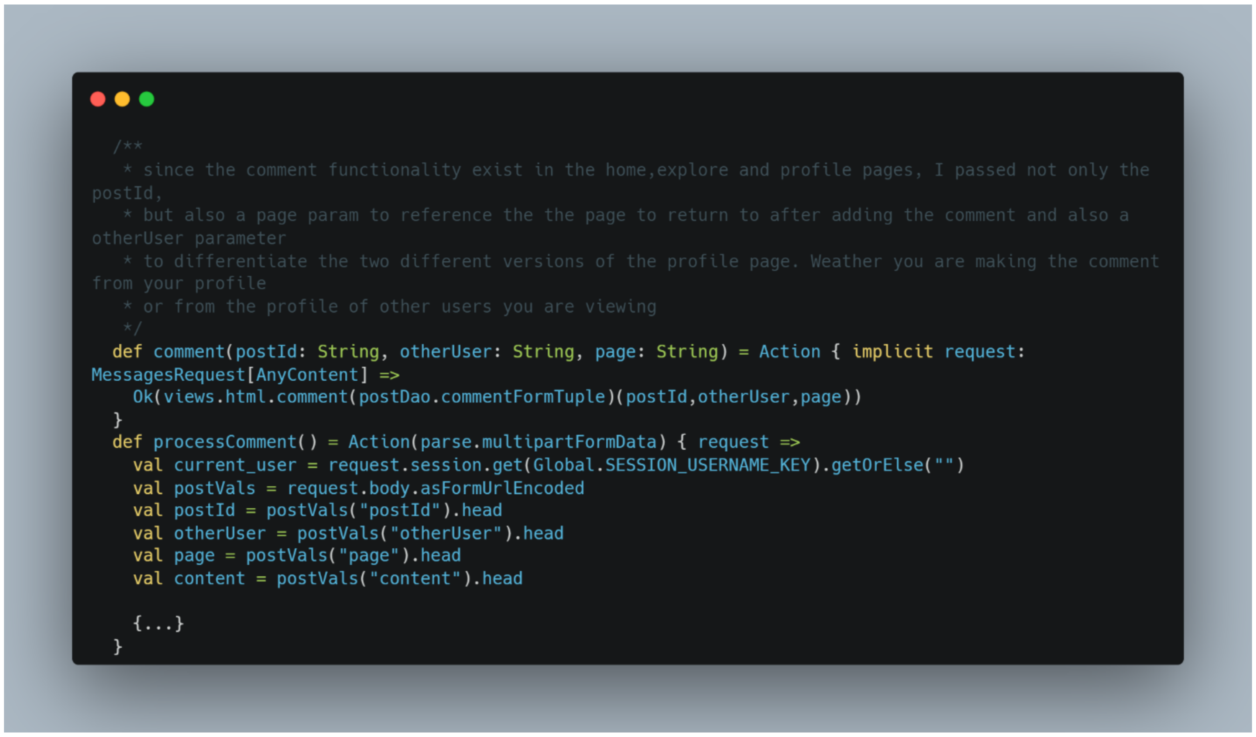


```scala
def follow(user: String) = authenticatedUserAction { implicit request: Request[AnyContent] =>
  val current_user = request.session.get(models.Global.SESSION_USERNAME_KEY).getOrElse("")
  userDao.follow(current_user, user)
  Redirect(routes.UserController.userProfile(user))
}

def unfollow(user: String) = authenticatedUserAction { implicit request: Request[AnyContent] =>
  val current_user = request.session.get(models.Global.SESSION_USERNAME_KEY).getOrElse("")
  userDao.unfollow(current_user, user)
  Redirect(routes.UserController.userProfile(user))
}
```

These methods are the actions performed when the current user follows or unfollows another user.

Comment and processComment

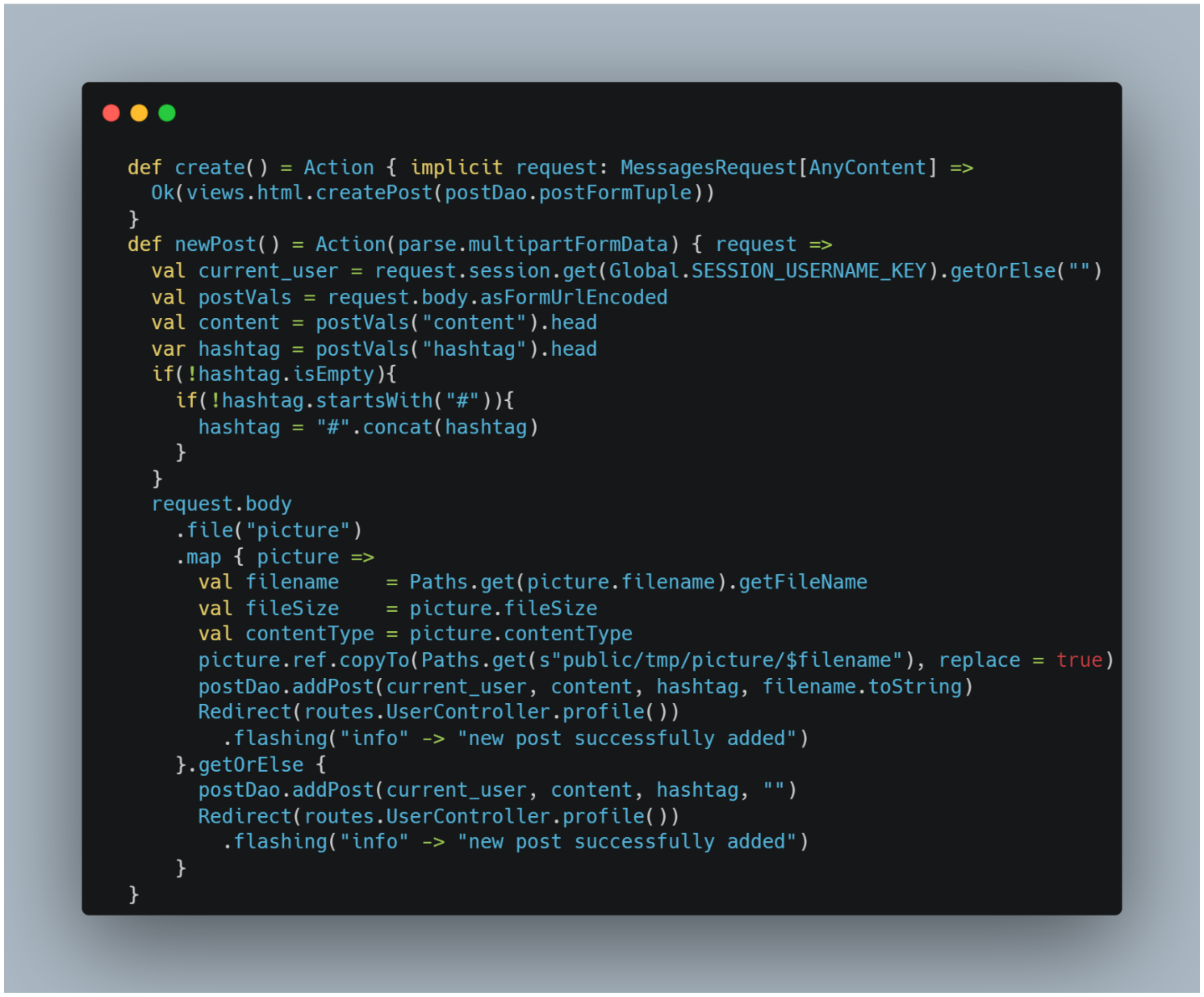


```scala
  /**
   * since the comment functionality exist in the home,explore and profile pages, I passed not only the
postId,
   * but also a page param to reference the the page to return to after adding the comment and also a
otherUser parameter
   * to differentiate the two different versions of the profile page. Weather you are making the comment
from your profile
   * or from the profile of other users you are viewing
   */
  def comment(postId: String, otherUser: String, page: String) = Action { implicit request:
MessagesRequest[AnyContent] =>
    Ok(views.html.comment(postDao.commentFormTuple)(postId,otherUser,page))
  }
  def processComment() = Action(parse.multipartFormData) { request =>
    val current_user = request.session.get(Global.SESSION_USERNAME_KEY).getOrElse("")
    val postVals = request.body.asFormUrlEncoded
    val postId = postVals("postId").head
    val otherUser = postVals("otherUser").head
    val page = postVals("page").head
    val content = postVals("content").head

    {...}
  }
```

These two methods are the actions performed to add a new comment to a post. The comment() methods takes the id of the post, the id of the otherUser(only in the profile view) and the calling page name. Once a new comment is added to a post, a redirect is made based on the calling page name. But in the case of the profile view, since the view could be of either the current user or other users, the otherUser parameter is set when a comment is made to a post in another user's profile else it is empty.

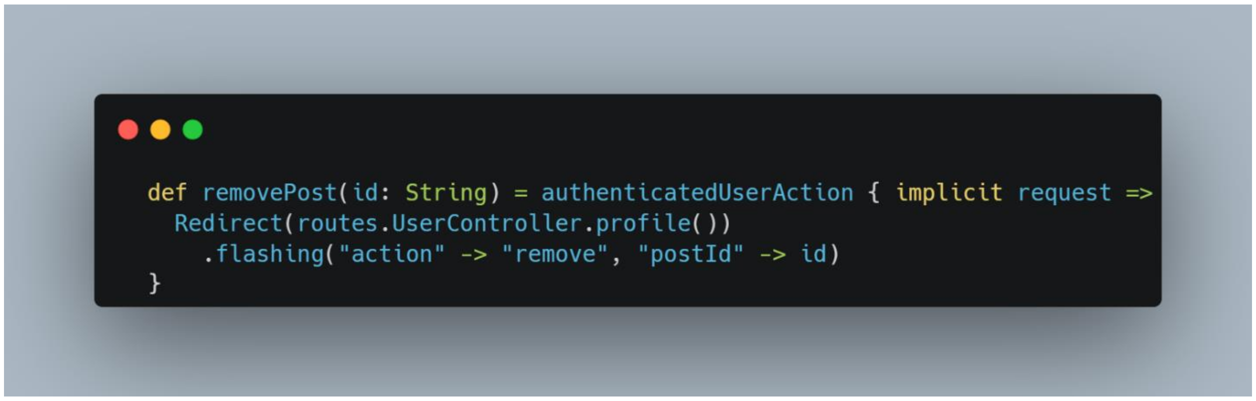
create and newPost



```scala
def create() = Action { implicit request: MessagesRequest[AnyContent] =>
  Ok(views.html.createPost(postDao.postFormTuple))
}
def newPost() = Action(parse.multipartFormData) { request =>
  val current_user = request.session.get(Global.SESSION_USERNAME_KEY).getOrElse("")
  val postVals = request.body.asFormUrlEncoded
  val content = postVals("content").head
  var hashtag = postVals("hashtag").head
  if(!hashtag.isEmpty){
    if(!hashtag.startsWith("#")){
      hashtag = "#".concat(hashtag)
    }
  }
  request.body
    .file("picture")
    .map { picture =>
      val filename    = Paths.get(picture.filename).getFileName
      val fileSize    = picture.fileSize
      val contentType = picture.contentType
      picture.ref.copyTo(Paths.get(s"public/tmp/picture/$filename"), replace = true)
      postDao.addPost(current_user, content, hashtag, filename.toString)
      Redirect(routes.UserController.profile())
        .flashing("info" -> "new post successfully added")
    }.getOrElse {
      postDao.addPost(current_user, content, hashtag, "")
      Redirect(routes.UserController.profile())
        .flashing("info" -> "new post successfully added")
    }
}
```

These two methods are the actions that get executed when the current user wants to add a new post. The create() method returns the form tuple against which validations are made and sends the user to the createPost view. When a new post is added, the newPost() method retrieves the post values and adds the post to the posts collection.

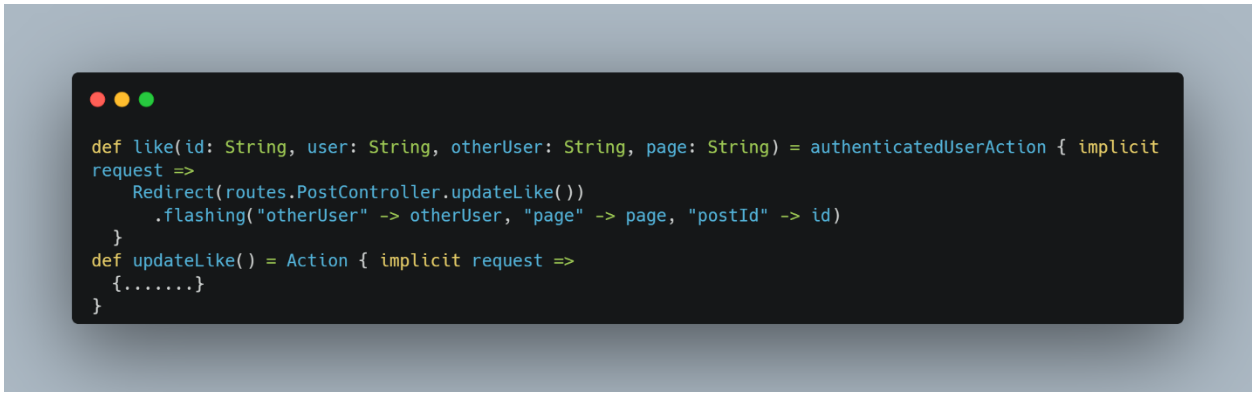
removePost



```
def removePost(id: String) = authenticatedUserAction { implicit request =>
  Redirect(routes.UserController.profile())
    .flashing("action" -> "remove", "postId" -> id)
}
```

This PostController method takes the id of the post to remove, redirects to the profile() method with flash "action" and "postId" of values "remove" and id parameter respectively. The post is then removed. Remember that you can only remove a post from the profile view.

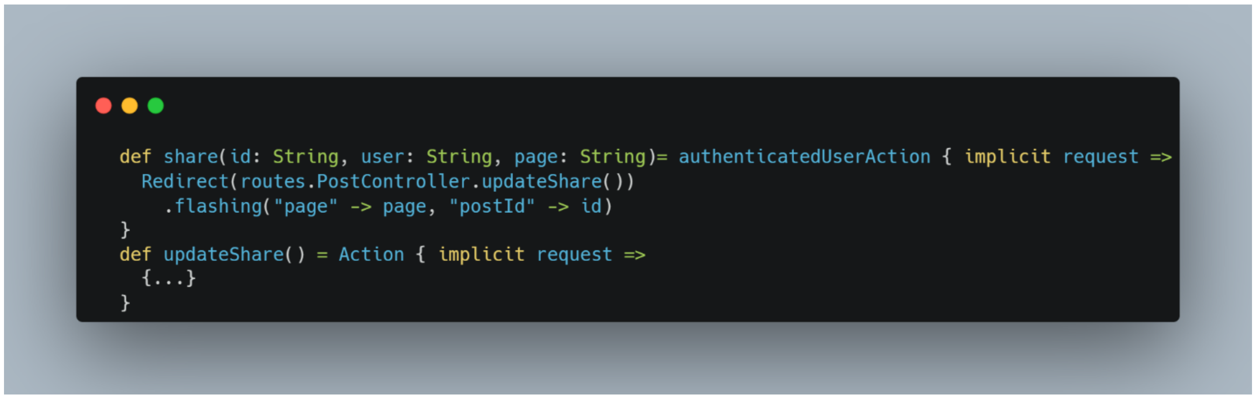
like and updateLike



```
def like(id: String, user: String, otherUser: String, page: String) = authenticatedUserAction { implicit
request =>
    Redirect(routes.PostController.updateLike())
      .flashing("otherUser" -> otherUser, "page" -> page, "postId" -> id)
  }
def updateLike() = Action { implicit request =>
  {.......}
}
```

Since a post can be liked from any view page, the like() method takes in parameters page and otherUser to know from which view the action is invoked and then calls the updateLike() method which then updates the like count of that post if that post has not already been liked by the current user and then redirects to the appropriate view based on the page value and the otherUser in the case of the profile view.

share and updateShare



```
def share(id: String, user: String, page: String)= authenticatedUserAction { implicit request =>
  Redirect(routes.PostController.updateShare())
    .flashing("page" -> page, "postId" -> id)
}
def updateShare() = Action { implicit request =>
  {...}
}
```

The actions of these methods are like the like() and updateLike() methods but here, it is the share count that is updated.