ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Департамент программной инженерии

СОГЛАСОВАНО
Профессор департамента программной инженерии факультета компьютерных наук, к.т.н

_____ Е. М. Гринкруг
«_____» _____ 2020 г.

УТВЕРЖДАЮ
Академический руководитель образовательной программы «Программная инженерия» профессор департамента программной инженерии, канд. техн. наук

_____ В. В. Шилов
«_____» _____ 2020 г.

Инв. № подл	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

СИСТЕМА УПРАВЛЕНИЯ ЗАДАНИЯМИ ПО АВТОМАТИЧЕСКОМУ СБОРУ ДАННЫХ ИЗ СЕТИ ИНТЕРНЕТ

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.04.13-01 12 01-1

Исполнитель: студент группы БПИ 174
_____ Д. Ю. Редникина
«_____» _____ 2020 г.

Москва 2020

УТВЕРЖДЕН
RU.17701729.04.13-01 12 01-1

Инв. № подл	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

СИСТЕМА УПРАВЛЕНИЯ ЗАДАНИЯМИ ПО АВТОМАТИЧЕСКОМУ СБОРУ ДАННЫХ ИЗ СЕТИ ИНТЕРНЕТ

Текст программы

RU.17701729.04.13-01 12 01-1

Листов 140

Москва 2020

Содержание

1 App **5**
1.1 controllers/ApplicationController.scala 5
1.2 controllers/CrawlersController.scala 6
1.3 controllers/JobsController.scala 9
1.4 controllers/MembershipController.scala 13
1.5 controllers/PeriodicJobsController.scala 15
1.6 controllers/ProjectsController.scala 22
1.7 controllers/SignInController.scala 27
1.8 controllers/SignUpController.scala 29
1.9 forms/OnetimeJobForm.scala 30
1.10 forms/periodic/PeriodicJobChangeForm.scala 31
1.11 forms/periodic/PeriodicJobCreateForm.scala 31
1.12 forms/project/ProjectChangeForm.scala 32
1.13 forms/project/ProjectDeployForm.scala 32
1.14 forms/project/ProjectForm.scala 33
1.15 forms/Request.scala 33
1.16 forms/SignIn.scala 33
1.17 forms/SignUp.scala 34
1.18 forms/SpiderChangeForm.scala 34
1.19 models/actors/JobSchedulerActor.scala 35
1.20 models/common/DBCreator.scala 37
1.21 models/common/enums/JobExecutionStatus.scala 39
1.22 models/common/enums/JobPriority.scala 40
1.23 models/common/enums/MembershipAccessRight.scala 40
1.24 models/common/enums/RunningStatus.scala 41
1.25 models/common/enums/RunningType.scala 41
1.26 models/common/extensions.scala 41
1.27 models/common/PGProfile.scala 42
1.28 models/common/settings/ScrapydSettings.scala 43
1.29 models/common/settings/SettingsFromDB.scala 43
1.30 models/common/settings/SettingsMerger.scala 44
1.31 models/Cookie.scala 44
1.32 models/daos/CrawlersDAO.scala 45
1.33 models/daos/JobDAO.scala 46
1.34 models/daos/MembershipDAO.scala 52
1.35 models/daos/PasswordDAO.scala 54
1.36 models/daos/ProjectDAO.scala 55
1.37 models/responses/Job.scala 57
1.38 models/responses/Member.scala 58
1.39 models/responses/SimpleJob.scala 59
1.40 models/scrapyd_response/AddVersionResponse.scala 59
1.41 models/scrapyd_response/CancelJobResponse.scala 59
1.42 models/scrapyd_response/GeneralResponse.scala 60
1.43 models/scrapyd_response/JobScrapyd.scala 60
1.44 models/scrapyd_response/ListJobsResponse.scala 61
1.45 models/scrapyd_response/ListSpidersResponse.scala 61

1.46 models/scrapyd_response/ScheduleResponse.scala . . . 61
1.47 models/services/JobService.scala . . . 62
1.48 models/services/MembershipService.scala . . . 71
1.49 models/services/ProjectService.scala . . . 72
1.50 models/services/ScrapydService.scala . . . 76
1.51 models/services/SecurityService.scala . . . 80
1.52 models/services/UpdaterService.scala . . . 86
1.53 models/services/UserService.scala . . . 88
1.54 models/tables/Crawler.scala . . . 90
1.55 models/tables/JobExecution.scala . . . 91
1.56 models/tables/JobInstance.scala . . . 92
1.57 models/tables/Membership.scala . . . 93
1.58 models/tables/Password.scala . . . 94
1.59 models/tables/Project.scala . . . 94
1.60 models/tables/User.scala . . . 95
1.61 modules/ActorModule.scala . . . 96
1.62 modules/SilhouetteModule.scala . . . 96
1.63 utils/DefaultEnv.scala . . . 100
1.64 views/index.scala.html . . . 100
1.65 views/main.scala.html . . . 100

2 Test **101**
2.1 controllers/ApplicationSpec.scala . . . 101
2.2 controllers/AuthorizationSpec.scala . . . 102
2.3 controllers/CrawlerSpec.scala . . . 103
2.4 controllers/jobs/JobTestCase.scala . . . 105
2.5 controllers/jobs/specs/JobExecutionSpec.scala . . . 106
2.6 controllers/jobs/specs/PeriodicJobSpec.scala . . . 111
2.7 controllers/MembershipSpec.scala . . . 119
2.8 controllers/ProjectSpec.scala . . . 121
2.9 unit/EmailValidatorTest.scala . . . 126
2.10 unit/SettingsMergerTest.scala . . . 126
2.11 utils/AuthSpecification.scala . . . 128
2.12 utils/BaseSpecification.scala . . . 129
2.13 utils/data/CrawlerData.scala . . . 130
2.14 utils/data/JobTestData.scala . . . 131
2.15 utils/data/ProjectTestData.scala . . . 132
2.16 utils/data/UserData.scala . . . 132
2.17 utils/DatabaseCleaner.scala . . . 133
2.18 utils/DBTestFiller.scala . . . 134
2.19 utils/TestHelper.scala . . . 137

Лист регистрации изменений **140**

1 App

1.1 controllers/ApplicationController.scala

```
package controllers

import com.mohiva.play.silhouette.api.{LogoutEvent, Silhouette}
import io.swagger.annotations.{Api, ApiOperation, ApiResponse,
   ApiResponses}
import javax.inject.Inject
import play.api.libs.json.Json
import play.api.mvc.{Action, AnyContent, BaseController,
   ControllerComponents}
import utils.DefaultEnv

import scala.concurrent.Future

@Api(value = "Logout")
class ApplicationController @Inject() (val controllerComponents
   : ControllerComponents,
                                       silhouette: Silhouette[
                                          DefaultEnv]) extends
                                          BaseController{

  @ApiOperation(value = "", hidden = true)
  def index: Action[AnyContent] = silhouette.UnsecuredAction.
     async { implicit request =>
    Future.successful(Ok(Json.toJson("Hello␣unauthorized!")))
  }

  @ApiOperation(value = "", hidden = true)
  def redirectDocs: Action[AnyContent] = Action { implicit
     request =>
    Redirect(
      url = "/assets/lib/swagger-ui/index.html",
      queryString = Map("url" -> Seq("http://" + request.host +
          "/swagger.json"))
    )
  }

  @ApiOperation(value = "Logout")
  @ApiResponses(Array(
    new ApiResponse(code = 401, message = "Unauthorized")))
  def logout: Action[AnyContent] = silhouette.SecuredAction.
     async { implicit request =>
    val result = Redirect(routes.ApplicationController.index)
    silhouette.env.eventBus.publish(LogoutEvent(request.
       identity, request))
    silhouette.env.authenticatorService.discard(request.
```

```
        authenticator, result)
  }
}
```

1.2 controllers/CrawlersController.scala

```
package controllers

import java.util.{Optional, UUID}

import com.mohiva.play.silhouette.api.Silhouette
import forms.SpiderChangeForm
import io.swagger.annotations.{Api, ApiImplicitParam,
   ApiImplicitParams, ApiOperation, ApiParam, ApiResponse,
   ApiResponses}
import javax.inject.Inject
import models.common.enums.MembershipAccessRight
import models.daos.CrawlersDAO
import models.services.SecurityService
import models.tables.{Crawler, Membership}
import play.api.libs.json.{JsError, JsSuccess, Json, OFormat}
import play.api.mvc.{Action, AnyContent, BaseController,
   ControllerComponents, Result}
import utils.DefaultEnv

import scala.concurrent.{ExecutionContext, Future}

@Api(value = "Crawlers")
class CrawlersController @Inject()(val controllerComponents:
   ControllerComponents,
                                   crawlersDAO: CrawlersDAO,
                                   securityService:
                                      SecurityService,
                                   silhouette: Silhouette[
                                      DefaultEnv])(implicit ex:
                                       ExecutionContext)
                                      extends BaseController {

  // MARK: - Formats

  implicit val crawlerFormat: OFormat[Crawler] = Json.format[
     Crawler]

  // MARK: - Get

  @ApiOperation(
    value = "List␣spiders",
    notes = "List␣spiders␣of␣project␣without␣pagination",
    response = classOf[Crawler],
    responseContainer = "Set")
```

```
@ApiResponses(Array(
  new ApiResponse(code = 403, message = "Couldn't get project
     's spiders due to access rights permission"),
  new ApiResponse(code = 401, message = "Unauthorized"),
  new ApiResponse(code = 422, message = "Coulnd't get spiders
     from DB")))
def listSpiders(@ApiParam(value = "Project ID") projectId:
   Long,
                @ApiParam(value = "Version of the project")
                   version: Option[String] = None): Action[
                   AnyContent] = silhouette.SecuredAction.
                   async { implicit request =>
  val userId = request.identity.id

  val getListSpiders: (Option[Membership]) => (Future[Result
     ]) = {
    case Some(_) => listSpidersFromDB(projectId, userId,
       version)
    case None => Future(Forbidden)
  }

  securityService
    .checkUserPermission(projectId, userId)
    .flatMap(getListSpiders)
    .recoverWith {
      case e: Exception => Future(Forbidden(e.getMessage))
    }
}

// MARK: - Update

@ApiOperation(
  value = "Update spider's settings",
  notes = "Updates only spider's settings. `projectId` should
     match `spiderId`")
@ApiImplicitParams(Array(
  new ApiImplicitParam(
    value = "Form with new settings",
    required = true,
    dataType = "forms.SpiderChangeForm",
    paramType = "body"
  )
))
@ApiResponses(Array(
  new ApiResponse(code = 400, message = "Bad format
     SpiderChangeForm"),
  new ApiResponse(code = 403, message = "Can't change spider'
     s data due to access right permission"),
  new ApiResponse(code = 401, message = "Unauthorized")))
```

```
def updateSpider(projectId: Long, crawlerId: Long): Action[
   SpiderChangeForm] = silhouette.SecuredAction.async(parse.
   json[SpiderChangeForm]) { implicit request =>

  val userId = request.identity.id
  val settings = Some(Json.toJson(request.body.settings))
  val args = Some(Json.toJson(request.body.args))

  val updateCrawlerAction: Option[Crawler] => Future[Result]
     = {
    case Some(_) =>
      crawlersDAO
        .update(projectId, crawlerId, settings, args, userId)
        .flatMap {
          _ => Future(Ok)
        }
    case None => Future(Forbidden)
  }

  securityService
    .checkUserAndCrawler(userId, projectId, crawlerId,
       MembershipAccessRight.ReadAndWrite)
    .flatMap(updateCrawlerAction)
    .recoverWith {
      case e: RuntimeException => Future(Forbidden(e.
         getMessage))
    }
}

// MARK: - Private

private def listSpidersFromDB(projectId: Long, userId: UUID,
   version: Option[String]): Future[Result] = {

  val getCrawlersFromProject: Option[Membership] => Future[
     Result] = {
    case Some(_) =>
      crawlersDAO
        .get(projectId)
        .flatMap { crawlers => Future(Ok(Json.toJson(crawlers
           ))) }
    case None =>
      Future(Forbidden)
  }

  securityService
    .checkUserPermission(projectId, userId)
    .flatMap(getCrawlersFromProject)
}
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
}
```

1.3 controllers/JobsController.scala

```
package controllers

import java.util.UUID

import com.mohiva.play.silhouette.api.Silhouette
import forms.{OnetimeJobForm, Request}
import io.swagger.annotations.{Api, ApiImplicitParam,
   ApiImplicitParams, ApiOperation, ApiParam, ApiResponse,
   ApiResponses}
import javax.inject.Inject
import models.common.enums.JobExecutionStatus.
   JobExecutionStatus
import models.common.enums.{JobExecutionStatus,
   MembershipAccessRight}
import models.responses.Job
import models.services.{JobService, SecurityService}
import models.tables.{Crawler, JobExecution, JobInstance}
import play.api.libs.json.{Json, OFormat}
import play.api.mvc.{Action, AnyContent, BaseController,
   ControllerComponents, PathBindable, Result, Results}
import utils.DefaultEnv

import scala.concurrent.{ExecutionContext, Future}

@Api(value = "Onetime␣jobs")
class JobsController @Inject()(val controllerComponents:
   ControllerComponents,
                               silhouette: Silhouette[
                                  DefaultEnv],
                               securityService: SecurityService
                                  ,
                               jobService: JobService) (
                                  implicit ex: ExecutionContext
                                  ) extends BaseController {


  implicit val rFormat: OFormat[JobInstance] = Json.format[
     JobInstance]
  implicit val r1Format: OFormat[JobExecution] = Json.format[
     JobExecution]

  private def formatError(wrapperError: Status, err: String) =
     Future(wrapperError(err))

  private def checkAccessAndPerformAction(userId: UUID,
                                          projectId: Long,
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
                                                jobScrapydId: UUID,
                                                jobId: Long): (Option
                                                   [JobExecution] =>
                                                   Future[Result]) =>
                                                    Future[Result] =
                                                   {
  action =>

  securityService
    .checkUserProjectAndJob(userId, projectId, jobId,
       jobScrapydId, MembershipAccessRight.ReadAndWrite)
    .flatMap(action)
    .recoverWith {
      case e: RuntimeException => Future(Forbidden(e.
         getMessage))
    }
}

@ApiOperation(
  value = "Schedule␣onetime␣job",
  notes = "User␣has␣to␣have␣‘ReadAndWrite‘␣access␣to␣project.
     ␣Initial␣status␣of␣the␣job␣=␣‘pending‘." +
    "Creates␣and␣starts␣new␣job␣with␣chosen␣crawler")
@ApiImplicitParams(Array(
  new ApiImplicitParam(
    value = "Form␣with␣settings␣of␣onetime␣job␣for␣scheduling
       ",
    required = true,
    dataType = "forms.OnetimeJobForm",
    paramType = "body")))
@ApiResponses(Array(
  new ApiResponse(code = 409, message = "Couldn't␣schedule␣
     job"),
  new ApiResponse(code = 400, message = "Bad␣format␣
     OnetimeJobForm"),
  new ApiResponse(code = 403, message = "Can't␣schedule␣job␣
     due␣to␣access␣right␣permission"),
  new ApiResponse(code = 401, message = "Unauthorized")))
def schedule(projectId: Long): Action[OnetimeJobForm] =
   silhouette.SecuredAction.async(parse.json[OnetimeJobForm])
    { implicit request =>

  val userId = request.identity.id
  val crawlerId = request.body.crawlerId

  val scheduleJob: Option[Crawler] => Future[Result] = {
    case Some(crawler) =>
      jobService
        .scheduleCrawler(crawler, projectId, userId, request.
```

```
            body)
          .flatMap {
            case Left(err) => formatError(Conflict, err)
            case Right(simpleJob) => Future(Ok(Json.toJson(
               simpleJob)))
          }
      case None =>
        Future(Forbidden)
    }

    securityService
      .checkUserAndCrawler(userId, projectId, crawlerId,
         MembershipAccessRight.ReadAndWrite)
      .flatMap(scheduleJob)
      .recoverWith {
        case e: RuntimeException => Future(Forbidden(e.
           getMessage))
      }
  }

  @ApiOperation(
    value = "Get␣list␣of␣job␣executions␣with␣pagination",
    notes =
      """
␣␣␣␣␣␣␣␣With␣pagination.␣Get␣all␣of␣the␣current␣jobs␣for␣all␣
   user's␣project.␣Returns␣`{pending,␣running,␣finished}`␣jobs.
␣␣␣␣␣␣␣␣␣␣-␣Logic:␣every␣time␣user␣asks␣for␣current␣statuses␣of␣
   tasks␣(GET),␣backend␣requests␣current␣data
␣␣␣␣␣␣␣␣␣␣from␣_scrapyd_␣and␣maps␣existing␣**job_instance**␣and␣
   **job_execution**␣objects␣in␣db.
␣␣␣␣␣␣␣␣␣␣After␣mapping,␣server␣responds␣to␣user␣with␣updated␣**
   List[Job]**.
␣␣␣␣␣␣␣␣␣␣_Note_:␣if␣scrapyd␣deleted␣jobId␣or␣server␣was␣
   restarted,␣then␣change␣status␣of␣the␣job␣to␣**finished**.
␣␣␣␣␣␣␣␣"""",
    response = classOf[Job],
    responseContainer = "Set")
  @ApiResponses(Array(
    new ApiResponse(code = 500, message = "Couldn't␣get␣jobs"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def getJobsExecutions(@ApiParam(value = "Limit␣for␣request",
     example = "10") limit: Int,
                        @ApiParam(value = "Status␣of␣job")
                           status: JobExecutionStatus.
                           JobExecutionStatus,
                        @ApiParam(value = "ID␣excludeFrom")
                           fromId: Option[Long] = None): Action
                           [AnyContent] = silhouette.
                           SecuredAction.async { implicit
```

```
                                request =>

    val r = Request[Long](limit, fromId)

    /** Do I need to return count? */
    jobService
      .get(r, request.identity.id, status)
      .flatMap { case (_, jobs) => Future(Ok(Json.toJson(jobs))
        ) }
      .recoverWith {
        case e: Exception => formatError(InternalServerError, e
          .getMessage)
      }
  }


  @ApiOperation(
    value = "Deletes␣finished␣job␣execution␣instance",
    notes = "It␣removes␣all␣the␣information␣from␣DB",
    response = classOf[UUID])
  @ApiResponses(Array(
    new ApiResponse(code = 422, message = "Couldn't␣delete␣job␣
      execution"),
    new ApiResponse(code = 403, message = "User␣doesn't␣have␣at
      ␣least␣ReadAndWrite␣access"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def deleteJob(projectId: Long, jobScrapydId: UUID, jobId:
     Long): Action[AnyContent] = silhouette.SecuredAction.async
      { implicit request =>

    val userId = request.identity.id
    val deleteAction: Option[JobExecution] => Future[Result] =
       { _ =>
      jobService
        .delete(jobId)
        .flatMap {
          case Left(err) => formatError(UnprocessableEntity,
            err)
          case Right(value) => Future(Ok(Json.toJson(value)))
        }
    }

    checkAccessAndPerformAction(userId, projectId, jobScrapydId
      , jobId)(deleteAction)
  }

  /** Method for cancelling running and pending tasks. It moves
      both of the statuses to finished. */
  @ApiOperation(
```

```
    value = "Cancel␣running␣and␣pending␣tasks",
    notes = "It␣moves␣both␣of␣the␣statuses␣to␣finished",
    response = classOf[UUID])
  @ApiResponses(Array(
    new ApiResponse(code = 422, message = "WrongStatus"),
    new ApiResponse(code = 403, message = "NoAccess"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def cancel(projectId: Long,
             jobScrapydId: UUID,
             jobId: Long): Action[AnyContent] = silhouette.
                SecuredAction.async { implicit request =>

    val userId = request.identity.id
    val changeRunningStatusAction: Option[JobExecution] =>
       Future[Result] = { _ =>

      jobService
        .changeRunningStatus(jobScrapydId, projectId)
        .flatMap {
          case Left(err) => formatError(UnprocessableEntity,
             err)
          case Right(value) => Future(Ok)
        }
    }

    checkAccessAndPerformAction(userId, projectId, jobScrapydId
       , jobId)(changeRunningStatusAction)
  }
}
```

1.4 controllers/MembershipController.scala

```
package controllers


import java.util.UUID

import com.google.inject.Inject
import com.mohiva.play.silhouette.api.Silhouette
import io.swagger.annotations.{Api, ApiOperation, ApiResponse,
   ApiResponses}
import models.common.enums.MembershipAccessRight
import models.common.enums.MembershipAccessRight.
   MembershipAccessRight
import models.responses.Member
import models.services.{MembershipService, SecurityService}
import play.api.libs.json.Json
import play.api.mvc.{Action, AnyContent, BaseController,
   ControllerComponents}
import utils.DefaultEnv
```

```
import scala.concurrent.{ExecutionContext, Future}

@Api(value = "Membership")
class MembershipController @Inject()(val controllerComponents:
   ControllerComponents,
                                     securityService:
                                        SecurityService,
                                     membershipService:
                                        MembershipService,
                                     silhouette: Silhouette[
                                        DefaultEnv])(implicit
                                        ex: ExecutionContext)
                                        extends BaseController
                                        {

  @ApiOperation(
    value = "Get␣list␣of␣members␣for␣project",
    notes = "Not␣paginated",
    response = classOf[Member],
    responseContainer = "Set")
  @ApiResponses(Array(
    new ApiResponse(code = 403, message = "Dont␣have␣permission
       ␣to␣specified␣project"),
    new ApiResponse(code = 500, message = "Couldn't␣get␣list␣of
       ␣members"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def getParticipants(projectId: Long): Action[AnyContent] =
     silhouette.SecuredAction.async { implicit request =>

    val userId = request.identity.id

    securityService
      .checkUserPermission(projectId, userId)
      .flatMap(_ => membershipService.get(projectId))
      .map(members => Ok(Json.toJson(members)))
      .recoverWith {
        case e: RuntimeException => Future(Forbidden(e.
           getMessage))
      }
  }


  @ApiOperation(
    value = "Delete␣user␣from␣membership␣list",
    notes = "Only␣owner␣can␣delete␣from␣membership␣list")
  @ApiResponses(Array(
    new ApiResponse(code = 403, message = "Do␣not␣have␣
       permission␣to␣delete␣member"),
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
    new ApiResponse(code = 401, message = "Unauthorized")))
  def deleteParticipant(projectId: Long, guestId: UUID): Action
     [AnyContent] = silhouette.SecuredAction.async { implicit
     request =>

    val userId = request.identity.id

    securityService
      .checkUserPermission(projectId, userId,
         MembershipAccessRight.Owner)
      .flatMap(_ => membershipService.delete(projectId, guestId
         ))
      .map(_ => Ok)
      .recoverWith {
        case e: RuntimeException => Future(Forbidden(e.
           getMessage))
      }
  }

  @ApiOperation(
    value = "Add␣or␣change␣participant␣of␣project",
    notes = "Insert␣or␣update")
  @ApiResponses(Array(
    new ApiResponse(code = 403, message = "Dont␣have␣permission
       ␣to␣specified␣project"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def addParticipants(projectId: Long, guestId: UUID,
     guestAccess: MembershipAccessRight): Action[AnyContent] =
     silhouette.SecuredAction.async { implicit request =>

    val userId = request.identity.id

    securityService
      .checkUserPermission(projectId, userId,
         MembershipAccessRight.Owner)
      .flatMap(_ => membershipService.put(projectId, guestId,
         guestAccess))
      .map(_ => Ok)
      .recoverWith {
        case e: RuntimeException => Future(Forbidden(e.
           getMessage))
      }
  }
}
```

1.5 controllers/PeriodicJobsController.scala

```
package controllers

import java.util.UUID
```

```
import com.mohiva.play.silhouette.api.Silhouette
import forms.periodic.{PeriodicJobChangeForm,
   PeriodicJobCreateForm}
import forms.Request
import io.swagger.annotations.{Api, ApiImplicitParam,
   ApiImplicitParams, ApiOperation, ApiResponse, ApiResponses}
import javax.inject.Inject
import models.common.enums.RunningStatus.RunningStatus
import models.common.enums.{MembershipAccessRight,
   RunningStatus}
import models.services.{JobService, SecurityService}
import models.tables.JobInstance.JobInstanceTable
import models.tables.{Crawler, JobExecution, JobInstance,
   Membership}
import play.api.libs.json.{Json, OFormat}
import play.api.mvc.{Action, AnyContent, BaseController,
   ControllerComponents, Result}
import utils.DefaultEnv

import scala.concurrent.{ExecutionContext, Future}

@Api(value = "Periodic␣jobs")
class PeriodicJobsController @Inject()(val controllerComponents
   : ControllerComponents,
                                       silhouette: Silhouette[
                                          DefaultEnv],
                                       securityService:
                                          SecurityService,
                                       jobService: JobService)
                                          (implicit ex:
                                          ExecutionContext)
                                          extends
                                          BaseController {

  implicit val rFormat: OFormat[JobInstance] = Json.format[
     JobInstance]
  implicit val r1Format: OFormat[JobExecution] = Json.format[
     JobExecution]

  type JobAction = Option[JobInstance] => Future[Result]

  // MARK: - Private

  private def mappingToResult[T]: Either[String, T] => Result =
      {
    case Left(err) => UnprocessableEntity(err)
    case Right(_) => Ok
  }
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
private def checkJobPermissionAndPerformAction(userId: UUID,
   projectId: Long, jobId: Long)
                                                (action:
                                                   JobAction):
                                                   Future[
                                                   Result] = {

  securityService
    .checkUserAndPeriodicJob(userId, projectId, jobId,
       MembershipAccessRight.ReadAndWrite)
    .flatMap(action)
    .recoverWith {
      case e: RuntimeException => Future(Forbidden(e.
         getMessage))
    }
}

private def changeStatusAction(status: RunningStatus)(
   changeStatus: () => Future[Result]): JobAction = {

  case Some(value) =>
    if (value.status == status) {
      Future(UnprocessableEntity("Job␣has␣already␣been␣
         disabled"))
    } else {
      changeStatus()
    }
  case None => Future(Forbidden)
}

// MARK: - GET

@ApiOperation(
  value = "Get␣list␣of␣periodic␣jobs␣with␣pagination",
  notes = "Gets␣data␣from␣DB.␣No␣requests␣to␣scrapyd␣needed.␣
     User␣has␣to␣have␣access␣(at␣least␣‘readonly‘)␣to␣
     requested␣project",
  response = classOf[JobInstance],
  responseContainer = "Set")
@ApiResponses(Array(
  new ApiResponse(code = 403, message = "Dont␣have␣at␣least␣
     read␣access␣to␣specified␣project"),
  new ApiResponse(code = 500, message = "Couldn't␣get␣list␣of
     ␣periodic␣jobs"),
  new ApiResponse(code = 401, message = "Unauthorized")))
def getPeriodicJobs(projectId: Long,
                    limit: Int,
                    exclusiveFrom: Option[Long] = None):
```

```
                        Action[AnyContent] = silhouette.
                        SecuredAction.async { implicit request
                         =>

    val userId = request.identity.id
    val requestWithPagination = Request[Long](limit,
       exclusiveFrom)

    /** Check for read access and get periodic jobs */
    val getPeriodicJobsAction: Option[Membership] => Future[
       Result] = { _ =>
      jobService
        .getJobInstances(projectId, requestWithPagination)
        .flatMap { periodicJobsInstance => Future(Ok(Json.
           toJson(periodicJobsInstance))) }
    }

    securityService
      .checkUserPermission(projectId, userId)
      .flatMap(getPeriodicJobsAction)
      .recoverWith {
        case e: RuntimeException => Future(Forbidden(e.
           getMessage))
      }
  }

  // MARK: - POST

  @ApiOperation(
    value = "Creates␣periodic␣job␣(jobInstance)",
    notes = "Creates␣JonInstance␣in␣DB.␣Schedules␣jobs␣
       according␣to␣specified␣‘cron␣expression‘.␣Checks␣for␣
       user␣access␣rights.",
    response = classOf[UUID])
  @ApiImplicitParams(Array(
    new ApiImplicitParam(
      value = "Form␣to␣create␣periodic␣job",
      required = true,
      dataType = "forms.periodic.PeriodicJobCreateForm",
      paramType = "body")))
  @ApiResponses(Array(
    new ApiResponse(code = 422, message = "Couldn’t␣create␣job␣
       instance"),
    new ApiResponse(code = 403, message = "Dont␣have␣at␣least␣
       write␣access␣to␣specified␣project"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def addPeriodicJob(projectId: Long): Action[
     PeriodicJobCreateForm] = silhouette.SecuredAction.async(
     parse.json[PeriodicJobCreateForm]) { implicit request =>
```

```
    val userId = request.identity.id
    val crawlerId = request.body.crawlerId

    /** Check for user access rights and create periodic job */
    val createPeriodicJobAction: Option[Crawler] => Future[
       Result] = {
      case None => Future(Forbidden)
      case Some(crawler) =>
        jobService
          .createPeriodicJobInstance(projectId, crawler,
             request.body)
          .map {
            case Left(err) => UnprocessableEntity(err)
            case Right(id) => Ok(Json.toJson(id))
          }
    }

    securityService
      .checkUserAndCrawler(userId, projectId, crawlerId,
         MembershipAccessRight.ReadAndWrite)
      .flatMap(createPeriodicJobAction)
      .recoverWith {
        case e: RuntimeException => Future(Forbidden(e.
           getMessage))
      }
  }

  // MARK:- PUT

  @ApiOperation(
    value = "Changes␣the␣periodic␣Job␣data",
    notes = "Checks␣for␣user␣access␣rights␣and␣job-project␣
       connection.")
  @ApiImplicitParams(Array(
    new ApiImplicitParam(
      value = "Form␣to␣change␣periodic␣job␣data",
      required = true,
      dataType = "forms.periodic.PeriodicJobChangeForm",
      paramType = "body")))
  @ApiResponses(Array(
    new ApiResponse(code = 422, message = "Couldn't␣change␣job␣
       instance"),
    new ApiResponse(code = 403, message = "Dont␣have␣at␣least␣
       write␣access␣to␣specified␣project␣or␣job-project␣don't␣
       correspond"),
    new ApiResponse(code = 500, message = "Error␣performing␣the
       ␣update␣in␣DB"),
    new ApiResponse(code = 401, message = "Unauthorized")))
```

```
def changePeriodicJob(projectId: Long,
                      periodicJobId: Long): Action[
                         PeriodicJobChangeForm] = silhouette.
                         SecuredAction.async(parse.json[
                         PeriodicJobChangeForm]) { implicit
                         request =>

  val userId = request.identity.id
  val crawlerId = request.body.crawlerId

  val changePeriodicJobAction: JobAction = { _ =>
    jobService
      .changePeriodicJobInstance(periodicJobId, request.body)
      .map(mappingToResult)
  }

  securityService
    .checkUserCrawlerAndPeriodicJob(userId, projectId,
       periodicJobId, crawlerId, MembershipAccessRight.
       ReadAndWrite)
    .flatMap(changePeriodicJobAction)
    .recoverWith {
      case e: RuntimeException => Future(Forbidden(e.
         getMessage))
    }
}


@ApiOperation(
  value = "Delete␣periodic␣job",
  notes = "Deletes␣periodic␣job␣instance␣(changed␣type␣to␣
     Onetime)␣and␣cancels␣all␣of␣the␣future␣job␣scheduled.")
@ApiResponses(Array(
  new ApiResponse(code = 422, message = "JobCouldn't BeDeleted
     "),
  new ApiResponse(code = 403, message = "NoPermission"),
  new ApiResponse(code = 401, message = "Unauthorized")))
def deletePeriodicJob(projectId: Long,
                      periodicJobId: Long): Action[AnyContent
                         ] = silhouette.SecuredAction.async {
                          implicit request =>

  val userId = request.identity.id

  val deletePeriodicJobAction: JobAction = { _ =>
    jobService
      .deletePeriodicJobInstance(periodicJobId)
      .map(mappingToResult)
  }
```

```
    checkJobPermissionAndPerformAction(userId, projectId,
       periodicJobId)(deletePeriodicJobAction)
  }

  @ApiOperation(
    value = "Sets␣status␣of␣periodicJob␣to␣disabled.",
    notes = "Cancels␣all␣of␣the␣future␣job␣scheduled.␣Does␣not␣
       modify␣running␣type,␣only␣running␣status.")
  def disable(projectId: Long,
              periodicJobId: Long): Action[AnyContent] =
                 silhouette.SecuredAction.async { implicit
                 request =>

    val userId = request.identity.id

    def disableAction(): Future[Result] =
      jobService
        .disableScheduling(periodicJobId)
        .map(mappingToResult)

    checkJobPermissionAndPerformAction(userId, projectId,
       periodicJobId)(changeStatusAction(RunningStatus.Disabled
       )(() => disableAction()))
  }

  @ApiOperation(
    value = "Enable␣scheduling␣jobs.",
    notes = "Continues␣to␣schedule␣job␣executions.␣Does␣not␣
       modify␣running␣type,␣only␣running␣status.")
  def enable(projectId: Long,
             periodicJobId: Long): Action[AnyContent] =
                silhouette.SecuredAction.async { implicit
                request =>

    val userId = request.identity.id

    def enableAction(): Future[Result] =
      jobService
        .enableScheduling(periodicJobId)
        .map(mappingToResult)

    checkJobPermissionAndPerformAction(userId, projectId,
       periodicJobId)(changeStatusAction(RunningStatus.Enabled)
       (() => enableAction()))
  }

}
```

1.6 controllers/ProjectsController.scala

```
package controllers


import java.util.UUID

import com.mohiva.play.silhouette.api.exceptions.
   ProviderException
import com.mohiva.play.silhouette.api.{LoginInfo, Silhouette}
import forms.project.{ProjectChangeForm, ProjectForm}
import forms.Request
import io.swagger.annotations.{Api, ApiImplicitParam,
   ApiImplicitParams, ApiOperation, ApiParam, ApiResponse,
   ApiResponses}
import javax.inject.Inject
import models.common.enums.MembershipAccessRight
import models.tables.Project
import models.services.{ProjectService, ScrapydService,
   SecurityService}
import play.api.libs.json.{Json, OFormat, Writes}
import play.api.mvc.{Action, AnyContent, BaseController,
   ControllerComponents, MultipartFormData, Result}
import utils.DefaultEnv
import play.api.libs.json._
import models.tables.{Crawler, Membership}
import play.api.libs.Files

import scala.concurrent.{ExecutionContext, Future}

@Api(value = "Projects")
class ProjectsController @Inject()(val controllerComponents:
   ControllerComponents,
                                   silhouette: Silhouette[
                                      DefaultEnv],
                                   projectService:
                                      ProjectService,
                                   securityService:
                                      SecurityService,
                                   scrapydService:
                                      ScrapydService)(implicit
                                      ex: ExecutionContext)
                                      extends BaseController {

  // MARK: - Formats for json serialization

  implicit val crawlerFormat: OFormat[Crawler] = Json.format[
     Crawler]
  implicit val projectFormat: OFormat[Project] = Json.format[
```

```
    Project]

implicit val membershipWriter: Writes[Membership] = new
    Writes[Membership] {
  def writes(m: Membership): JsObject = Json.obj(
    "pId" -> m.projectId,
    "uId"         -> m.userId,
    "access"     -> m.accessRight
  )
}

// MARK: - Lifecycle

@ApiOperation(value = "Get␣projects", response = classOf[
    Project], responseContainer = "Set")
@ApiResponses(Array(
  new ApiResponse(code = 500, message = "Couldn't␣get␣
      projects"),
  new ApiResponse(code = 401, message = "Unauthorized")))
def getProjects(@ApiParam(value = "Limit␣for␣request",
    example = "10") limit: Int,
                @ApiParam(value = "ID␣excludeFrom") id:
                    Option[Long] = None): Action[AnyContent] =
                     silhouette.SecuredAction.async { implicit
                     request =>
  val r = Request[Long](limit, id)

  projectService
    .get(r, request.identity.id)
    .flatMap {
      p => Future(Ok(Json.toJson(p)))}
    .recover {
      case _: ProviderException =>
        InternalServerError
    }
}

@ApiOperation(value = "Create␣project", response = classOf[
    UUID])
@ApiImplicitParams(Array(
  new ApiImplicitParam(
    value = "Form␣with␣initial␣project␣data",
    required = true,
    dataType = "forms.project.ProjectForm",
    paramType = "body"
  )
))
@ApiResponses(Array(
  new ApiResponse(code = 500, message = "Couldn't␣create␣
```

```
      projects"),
    new ApiResponse(code = 401, message = "Unauthorized"),
    new ApiResponse(code = 400, message = "
      ProjectFormBadRequest")))
  def createProject: Action[ProjectForm] = silhouette.
     SecuredAction.async(parse.json[ProjectForm]) { implicit
     request =>

    projectService
      .create(request.body, request.identity)
      .flatMap {
        pId => Future(Ok(Json.toJson(pId)))
      }
      .recover {
        case _: ProviderException =>
          InternalServerError
      }
  }

  @ApiOperation(value = "Change␣project␣metadata")
  @ApiImplicitParams(Array(
    new ApiImplicitParam(
      value = "Form␣with␣metadata␣to␣be␣changed",
      required = true,
      dataType = "forms.project.ProjectChangeForm",
      paramType = "body"
    )
  ))
  @ApiResponses(Array(
    new ApiResponse(code = 403, message = "Do␣not␣have␣
      permission␣to␣change␣project"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def updateProjectMetadata(projectId: Long): Action[
     ProjectChangeForm] = silhouette.SecuredAction.async(parse.
     json[ProjectChangeForm]) { implicit request =>

    val user = request.identity

    val updateContentOfTheProject: (Option[Membership] =>
       Future[Status]) = {
      case Some(_) =>
        projectService
          .updateMetadataContent(projectId, request.body, user)
          .flatMap(_ => Future(Ok))
      case None =>
        Future(Forbidden)
    }

    securityService
```

```
      .checkUserPermission(projectId, user.id,
         MembershipAccessRight.ReadAndWrite)
      .flatMap(updateContentOfTheProject)
      .recoverWith {
        case _: RuntimeException =>
          Future(Forbidden)
      }
  }

  @ApiOperation(
    value = "Delete␣project",
    notes = "We␣can't␣delete␣project␣from␣our␣DB␣in␣case␣we␣
       encounter␣error␣deleting␣it␣from␣scrapyd")
  @ApiResponses(Array(
    new ApiResponse(code = 422, message = "Error␣deleting␣
       project␣from␣scrapyd"),
    new ApiResponse(code = 403, message = "Do␣not␣have␣
       permission␣to␣delete␣project"),
    new ApiResponse(code = 401, message = "Unauthorized")))
  def deleteProject(projectId: Long): Action[AnyContent] =
     silhouette.SecuredAction.async { implicit request =>
    val user = request.identity

    /** Firstly we need to delete project from scrapyd, in case
        of success - delete from our DB.
     * Otherwise return status */
    val deleteContentOfTheProject: (Option[Membership] =>
       Future[Status]) = {
      case Some(_) =>
        scrapydService
          .delProject(projectId)
          .flatMap {
            case JsSuccess(_, _) =>
              projectService.delete(projectId).map(_ => Ok)
            case JsError(_) =>
              Future(UnprocessableEntity)
          }
      case None =>
        Future(Forbidden)
    }

    securityService
      .checkUserPermission(projectId, user.id,
         MembershipAccessRight.Owner)
      .flatMap(deleteContentOfTheProject)
      .recoverWith {
        case _: RuntimeException => Future(Forbidden)
      }
  }
```

```
@ApiOperation(
  value = "Deploy␣project's␣eggfile␣to␣scrapyd",
  response = classOf[Crawler],
  responseContainer = "Set")
@ApiImplicitParams(Array(
  new ApiImplicitParam(
    name = "eggFile",
    required = true,
    paramType = "form")))
@ApiResponses(Array(
  new ApiResponse(code = 422, message = "Error␣getting␣
     eggfile␣from␣multipart␣form/data␣or␣error␣deploy␣to␣
     scrapyd"),
  new ApiResponse(code = 403, message = "Do␣not␣have␣
     permission␣to␣deploy␣project"),
  new ApiResponse(code = 401, message = "Unauthorized")))
def deployProject(projectId: Long): Action[MultipartFormData[
   Files.TemporaryFile]] = silhouette.SecuredAction.async(
   parse.multipartFormData) { implicit request =>

  /** Either deploy or say it's unprocessable */
  val mapEggFile: Option[MultipartFormData.FilePart[Files.
     TemporaryFile]] => Future[Result] = {
    case Some(egg) =>
      projectService
        .deployEggFile(egg.ref, projectId, request.identity.
           id)
        .flatMap {
          case Left(errors) =>
            Future(UnprocessableEntity(Json.toJson(Map("
               errors" -> errors))))
          case Right(spiders) =>
            Future(Ok(Json.toJson(spiders)))
        }
    case None =>
      Future(UnprocessableEntity)
  }

  /** Either pass further or say it's forbidden */
  val deployEggFileOfTheProject: Option[Membership] => Future
     [Result] = {
    case Some(_) =>
      mapEggFile(request.body.file("eggFile"))
    case None =>
      Future(Forbidden)
  }

  securityService
```

```
        .checkUserPermission(projectId, request.identity.id,
           MembershipAccessRight.ReadAndWrite)
        .flatMap(deployEggFileOfTheProject)
        .recoverWith {
          case _: RuntimeException => Future(Forbidden)
        }
  }

}
```

1.7 controllers/SignInController.scala

```
package controllers

import com.mohiva.play.silhouette.api.exceptions.
   ProviderException
import com.mohiva.play.silhouette.api.{LoginEvent, LoginInfo,
   Silhouette}
import com.mohiva.play.silhouette.api.util.Credentials
import com.mohiva.play.silhouette.impl.providers.
   CredentialsProvider
import forms.SignIn
import io.swagger.annotations.{Api, ApiImplicitParam,
   ApiImplicitParams, ApiOperation, ApiResponse, ApiResponses}
import javax.inject._
import models.Cookie
import play.api.libs.json.{Json, OFormat}
import play.api.mvc._
import models.services.UserService
import utils.DefaultEnv

import scala.concurrent.{ExecutionContext, Future}

/**
 * This controller creates an `Action` to handle HTTP requests
    to the
 * application's home page.
 */
@Api(value = "Login")
@Singleton
class SignInController @Inject()(val controllerComponents:
   ControllerComponents,
                                 userService: UserService,
                                 silhouette: Silhouette[
                                    DefaultEnv],
                                 credentialsProvider:
                                    CredentialsProvider)(implicit
                                     ex: ExecutionContext)
                                    extends BaseController {
```

```
@ApiOperation(value = "Get␣authentication␣token", response =
   classOf[Cookie])
@ApiImplicitParams(Array(
  new ApiImplicitParam(
    value = "Credentials",
    required = true,
    dataType = "forms.SignIn",
    paramType = "body"
  )
))
@ApiResponses(Array(
  new ApiResponse(code = 403, message = "
     InvalidCredentialsProvided"),
  new ApiResponse(code = 400, message = "SignInBadRequest")))
def signIn(): Action[SignIn] = Action.async(parse.json[SignIn
   ]) { implicit request =>
  val credentials = Credentials(request.body.email, request.
     body.password)
  credentialsProvider
    .authenticate(credentials)
    .flatMap(login)
    .recover {
      case _: ProviderException =>
        Forbidden
    }
}

def login(loginInfo: LoginInfo)(implicit request:
   RequestHeader): Future[Result] = {
  val mbUser = userService.retrieve(loginInfo)
  mbUser.flatMap {
    case Some(user) =>
      for {
        authenticator <- silhouette.env.authenticatorService.
           create(loginInfo)
        v             <- silhouette.env.authenticatorService.
           init(authenticator)
        result <- silhouette.env.authenticatorService.embed(v
           ,
          Ok(Json.toJson(Cookie(cookie = v.name, login = v.
             value)))
        )
      } yield {
        silhouette.env.eventBus.publish(LoginEvent(user,
           request))
        result
      }
    case None =>
      Future(Forbidden)
```

```
    }
  }
}
```

1.8 controllers/SignUpController.scala

```
package controllers


import com.mohiva.play.silhouette.api.{LoginEvent, LoginInfo,
   SignUpEvent, Silhouette}
import com.mohiva.play.silhouette.impl.providers.
   CredentialsProvider
import forms.{SignIn, SignUp}
import io.swagger.annotations.{Api, ApiImplicitParam,
   ApiImplicitParams, ApiOperation, ApiResponse, ApiResponses}
import javax.inject._
import models.Cookie
import play.api.libs.json.Json
import play.api.mvc._
import models.services.UserService
import utils.DefaultEnv

import scala.concurrent.{ExecutionContext, Future}
import models.common.extensions._

/**
 * This controller creates an `Action` to handle HTTP requests
    to the
 * application's home page.
 */
@Api("Registration")
@Singleton
class SignUpController @Inject()(val controllerComponents:
   ControllerComponents,
                                 userService: UserService,
                                 silhouette: Silhouette[
                                    DefaultEnv],
                                 credentialsProvider:
                                    CredentialsProvider)(implicit
                                     ex: ExecutionContext)
                                    extends BaseController {

  val UserAlreadyExistsMessage = "User already exists"

  @ApiOperation(value = "Get authentication token", response =
     classOf[Cookie])
  @ApiImplicitParams(Array(
    new ApiImplicitParam(
      value = "Credentials",
```

```
      required = true,
      dataType = "forms.SignUp", // complete path
      paramType = "body"
    )
  ))
  @ApiResponses(Array(
    new ApiResponse(code = 409, message = "
       UserAlreadyExistsMessage"),
    new ApiResponse(code = 403, message = "EmailWrongFormat"),
    new ApiResponse(code = 400, message = "SignUp␣body␣bad␣
       request")))
  def signUp(): Action[SignUp] = Action.async(parse.json[SignUp
     ]) { implicit request =>

    if (!request.body.email.isEmail) {
      Future(Forbidden)
    } else {
      val loginInfo = LoginInfo(CredentialsProvider.ID, request
         .body.email)
      userService
        .retrieve(loginInfo)
        .flatMap {
          case Some(_) =>
            Future(Conflict(Json.toJson(
               UserAlreadyExistsMessage)))
          case None =>
            for {
              user <- userService.create(request.body)
              authenticator <- silhouette.env.
                 authenticatorService.create(loginInfo)
              v <- silhouette.env.authenticatorService.init(
                 authenticator)
              result <- silhouette.env.authenticatorService.
                 embed(v,
                Ok(Json.toJson(Map("user" -> loginInfo)))
              )
            } yield {
              silhouette.env.eventBus.publish(SignUpEvent(user,
                  request))
              result
            }
        }
    }
  }
}
```

1.9 forms/OnetimeJobForm.scala

```
package forms
```

```
import java.time.Instant
import java.util.UUID

import io.swagger.annotations.ApiModel
import models.common.enums.JobPriority.JobPriority
import models.common.enums.RunningType.RunningType
import play.api.libs.json.{JsValue, Json, OFormat}

object OnetimeJobForm {
  implicit val jobFormat: OFormat[OnetimeJobForm] = Json.format
     [OnetimeJobForm]
}

@ApiModel(description = "Form␣for␣onetime␣job␣scheduling")
case class OnetimeJobForm(crawlerId: Long,
                          priority: JobPriority,
                          args: Map[String, String] = Map.empty
                             ,
                          settings: Map[String, String] = Map.
                             empty,
                          start: Option[Instant] = None)
```

1.10 forms/periodic/PeriodicJobChangeForm.scala

```
package forms.periodic

import io.swagger.annotations.ApiModel
import models.common.enums.JobPriority
import models.common.enums.JobPriority.JobPriority
import play.api.libs.json.{JsValue, Json, OFormat}

object PeriodicJobChangeForm {
  implicit val jobFormat: OFormat[PeriodicJobChangeForm] = Json
     .format[PeriodicJobChangeForm]
}

@ApiModel(description = "PeriodicJob␣change␣form")
case class PeriodicJobChangeForm(title: String,
                                 description: Option[String] =
                                    None,
                                 crawlerId: Long,
                                 priority: JobPriority =
                                    JobPriority.Normal,
                                 cronExpression: String,
                                 settings: Option[JsValue] =
                                    None,
                                 args: Option[JsValue] = None)
```

1.11 forms/periodic/PeriodicJobCreateForm.scala

```
package forms.periodic

import io.swagger.annotations.ApiModel
import models.common.enums.JobPriority.JobPriority
import models.common.enums.RunningStatus.RunningStatus
import models.common.enums.{JobPriority, RunningStatus}
import play.api.libs.json.{JsValue, Json, OFormat}

object PeriodicJobCreateForm {
  implicit val jobFormat: OFormat[PeriodicJobCreateForm] = Json
     .format[PeriodicJobCreateForm]
}

@ApiModel(description = "PeriodicJob␣create␣form")
case class PeriodicJobCreateForm(title: String,
                                 description: Option[String] =
                                    None,
                                 crawlerId: Long,
                                 status: RunningStatus =
                                    RunningStatus.Enabled,
                                 priority: JobPriority =
                                    JobPriority.Normal,
                                 cronExpression: String,
                                 settings: Option[JsValue] =
                                    None,
                                 args: Option[JsValue] = None)
```

1.12 forms/project/ProjectChangeForm.scala

```
package forms.project

import io.swagger.annotations.ApiModel
import play.api.libs.json.{Json, OFormat}

object ProjectChangeForm {
  implicit val projectChangeFormat: OFormat[ProjectChangeForm]
     = Json.format[ProjectChangeForm]
}

@ApiModel(description = "Form␣to␣change␣metadata␣of␣the␣project
   ")
case class ProjectChangeForm(name: String,
                             description: Option[String],
                             spiderSettings: Map[String, String
                                ],
                             spiderArgs: Map[String, String])
```

1.13 forms/project/ProjectDeployForm.scala

```
package forms.project
```

```
import io.swagger.annotations.ApiModel
import play.api.libs.json.{Json, OFormat}

object ProjectDeployForm {
  implicit val projectDeployFormat: OFormat[ProjectDeployForm]
     = Json.format[ProjectDeployForm]
}

@ApiModel(description = "Form␣to␣deploy␣project␣to␣scrapyd")
case class ProjectDeployForm(eggFile: Array[Byte])
```

1.14 forms/project/ProjectForm.scala

```
package forms.project

import io.swagger.annotations.{ApiModel, ApiModelProperty}
import play.api.libs.json.{Json, OFormat}

object ProjectForm {
  implicit val projectFormat: OFormat[ProjectForm] = Json.
     format[ProjectForm]
}

@ApiModel(description = "ProjectCreate␣form")
case class ProjectForm(@ApiModelProperty(example = "Project1")
   name: String,
                       @ApiModelProperty(example = "Some␣
                          description") description: Option[
                          String])
```

1.15 forms/Request.scala

```
package forms

import java.util.UUID

/**
  Requests with pagination (projects, users, crawlers...).
  For request from the very beginning send null in `
     exclusiveFrom`.
  Note:    request                        null        id
                    ,
                                    ...
 */
case class Request[T] (limit: Int,
                       exclusiveFrom: Option[T])
```

1.16 forms/SignIn.scala

```
package forms

import io.swagger.annotations.{ApiModel, ApiModelProperty}
import play.api.libs.json.{Json, OFormat}

@ApiModel(description = "SignIn␣form")
case class SignIn(@ApiModelProperty(example = "Vasya@mail.ru")
   email: String,
                  @ApiModelProperty(example = "1234") password:
                      String)

object SignIn {
  implicit val signInFormat: OFormat[SignIn] = Json.format[
     SignIn]
}
```

1.17 forms/SignUp.scala

```
package forms

import io.swagger.annotations.{ApiModel, ApiModelProperty}
import play.api.libs.json.{Json, OFormat}

@ApiModel(description = "SignUn␣form")
case class SignUp (@ApiModelProperty(example = "Vasya") name:
   String,
                   @ApiModelProperty(example = "vasya99") login
                      : String,
                   @ApiModelProperty(example = "Vasya@mail.ru")
                       email: String,
                   @ApiModelProperty(example = "1234") password
                      : String)

object SignUp {
  implicit val signUpFormat: OFormat[SignUp] = Json.format[
     SignUp]
}
```

1.18 forms/SpiderChangeForm.scala

```
package forms

import io.swagger.annotations.ApiModel
import play.api.libs.json.{JsValue, Json, OFormat}

@ApiModel(description = "Form␣to␣change␣spider's␣settings")
case class SpiderChangeForm(settings: Map[String, String] = Map
   .empty,
                            args: Map[String, String] = Map.
                               empty)
```

```
object SpiderChangeForm {
  implicit val crawlerChangeFormat: OFormat[SpiderChangeForm] =
      Json.format[SpiderChangeForm]
}
```

1.19 models/actors/JobSchedulerActor.scala

```
package models.actors

import java.util.UUID

import akka.actor._
import akka.pattern.pipe
import javax.inject.{Inject, Singleton}
import models.common.settings.ScrapydSettings
import models.daos.JobDAO
import models.responses.SimpleJob
import models.services.ScrapydService
import models.tables.{Crawler, JobInstance}
import play.api.libs.json.{JsError, JsSuccess}

import scala.concurrent.{ExecutionContext, Future}

object JobSchedulerActor {

  def props: Props = Props[JobSchedulerActor]

  /**
   * Class for scheduling one time job
   * @param crawler what will be crawling websites
   * @param periodicJob parent
   */
  case class ScheduleJob(crawler: Crawler, periodicJob:
    JobInstance)

  /**
   * Class for scheduling many one time jobs for existing job
      instance
   * @param crawler what will be crawling websites
   * @param periodicJob already existing parent of one of many
   */
  case class ScheduleJobRepeatedly(crawler: Crawler,
    periodicJob: JobInstance)
}

/**
 * Class for scheduling onetime jobs.
 *
 * It is used while scheduling onetime job or creating periodic
```

```
     job instance.
 */
@Singleton
class JobSchedulerActor @Inject()(scrapydService:
   ScrapydService,
                                  jobDAO: JobDAO) extends Actor
                                      {
  import JobSchedulerActor._
  implicit val ec: ExecutionContext = context.dispatcher

  def receive: PartialFunction[Any, Unit] = {

    // Call schedule onetime job
    case ScheduleJob(crawler: Crawler, jobInstance: JobInstance
       ) =>
      val jobInstanceAction =
        jobDAO
        .addAction(jobInstance)
        .flatMap(mapToPrioritySettings(crawler.projectId,
           crawler, crawler.id))

      jobInstanceAction pipeTo sender


    // Call schedule one time job for existing job instance (
       periodic)
    case ScheduleJobRepeatedly(crawler: Crawler, jobInstance:
       JobInstance) =>
      val jobExecutionInsertAction =
        jobDAO
          .addJobExecution(jobInstance)
          .flatMap(mapToPrioritySettings(crawler.projectId,
             crawler, crawler.id))

      jobExecutionInsertAction pipeTo sender
  }

  def mapToPrioritySettings(projectId: Long, crawler: Crawler,
     crawlerId: Long): ((Long, UUID)) => Future[Either[String,
     SimpleJob]] = { case (id, scrapydId) =>

    jobDAO
      .mergeOnetimeSettings(projectId, crawlerId, id)
      .flatMap(mapToScheduleJob(projectId, crawler, id,
         scrapydId))
  }


  def mapToScheduleJob(projectId: Long,
```

```
                        crawler: Crawler,
                        id: Long,
                        scrapydId: UUID): (ScrapydSettings) =>
                          Future[Either[String, SimpleJob]] = {
    setting =>

    scrapydService
      .scheduleJob(projectId = projectId, spiderName = crawler.
        name,
        jobId = scrapydId, setting = setting.setting, args =
          setting.args)
      .flatMap {
        //  immediately should revert changes in DB (delete
          added jobInstance and jobExecution)
        //  and return error
        case JsError(errors) => jobDAO.deleteJob(id).map { _ =>
            Left(errors.toString) }
        case JsSuccess(_, _) => Future(Right(SimpleJob(id = id,
            scrapydId = scrapydId)))
      }
  }
}
```

1.20 models/common/DBCreator.scala

```
package models.common


import com.github.tminglei.slickpg.PgEnumSupportUtils
import com.github.tminglei.slickpg.PgEnumSupportUtils.sqlName
import models.common.enums._
import models.tables._
import models.common.PGProfile.api._
import models.common.extensions._

import scala.util.Try


object DBCreator {

  // MARK: - Main

  def main(args: Array[String]): Unit = {

    val db = Database.forConfig("slick.dbs.default.db")

    Try(down(db))
    Try(downTypes(db))

    upTypes(db)
```

```
    up(db)

    print(
      """
␣␣␣␣␣␣␣␣|------------------------
␣␣␣␣␣␣␣␣|␣␣␣Tables␣are␣ready!
␣␣␣␣␣␣␣␣|------------------------
␣␣␣␣␣␣␣␣|""".stripMargin)
  }

  // MARK: - Tables

  val types = Seq(
    ("job_execution_status", JobExecutionStatus),
    ("job_priority", JobPriority),
    ("running_status", RunningStatus),
    ("running_type", RunningType),
    ("access_rights", MembershipAccessRight)
  )

  val tablesQueries = Seq(
    User.dbUsers,
    Password.dbPasswords,
    Project.dbProjects,
    Crawler.dbCrawlers,
    JobInstance.dbJobInstances,
    JobExecution.dbJobExecutions,
    Membership.dbMembership
  )

  // MARK: - Get db

  def upTypes(db: Database): Unit = {
    db.run(createEnum()).awaitForResult
  }

  def downTypes(db: Database): Unit = {
    db.run(dropEnums()).awaitForResult
  }

  def up(db: Database): Unit = {
    db.run(createSchemas()).awaitForResult
  }

  def down(db: Database): Unit =
    db.run(dropTables()).awaitForResult

  // MARK: - Private
```

```
  private def dropTable(schemaName: String): DBIOAction[Int,
     NoStream, Effect] = {
    sqlu"drop␣table␣if␣exists␣#${sqlName(schemaName,␣quoteName␣
      =␣true)}"
  }

  private def buildDropIfExistSql(sqlTypeName: String,
     quoteName: Boolean = false): DBIOAction[Int, NoStream,
     Effect] = {
    sqlu"drop␣type␣if␣exists␣#${sqlName(sqlTypeName,␣quoteName)
      }"
  }


  private def createEnum() = types.map {
    case (n, t) => PgEnumSupportUtils.buildCreateSql(n, t).
      asInstanceOf[DBIOAction[Int, NoStream, Effect]]
  }.reduce(_ andThen _)


  private def createSchemas() = {

    val schemas = tablesQueries.map(_.schema)
    def createSchema(schema: PGProfile.SchemaDescription):
      DBIOAction[Unit, NoStream, Effect.Schema] = schema.
      create

    schemas
      .map(createSchema)
      .reduce(_ andThen _)
  }

  private def dropTables() = {
    tablesQueries
      .reverse
      .map(_.baseTableRow.tableName)
      .map(dropTable)
      .reduce(_ andThen _)
  }

  private def dropEnums() =
    types.map { case (name, _) => buildDropIfExistSql(name) }.
      reduce(_ andThen _)
}
```

1.21 models/common/enums/JobExecutionStatus.scala

```
package models.common.enums

import play.api.libs.json.{Reads, Writes}
```

```
import play.api.mvc.PathBindable

object JobExecutionStatus extends Enumeration {
  type JobExecutionStatus = Value
  val Running, Pending, Finished = Value

  implicit val myEnumReads = Reads.enumNameReads(
     JobExecutionStatus)
  implicit val myEnumWrites = Writes.enumNameWrites

  implicit object searchTypeQueryStringBinder extends
     PathBindable.Parsing[JobExecutionStatus.JobExecutionStatus
     ](
      withName, _.toString,
      (k: String, e: Exception) => "Cannot␣parse␣%s␣as␣
         SearchTypes:␣%s".format(k, e.getMessage)
  )
}
```

1.22 models/common/enums/JobPriority.scala

```
package models.common.enums

import play.api.libs.json.{Reads, Writes}

object JobPriority extends Enumeration {
  type JobPriority = Value
  val Low, Normal, High, Highest  = Value

  implicit val myEnumReads = Reads.enumNameReads(JobPriority)
  implicit val myEnumWrites = Writes.enumNameWrites
}
```

1.23 models/common/enums/MembershipAccessRight.scala

```
package models.common.enums

import play.api.libs.json.{Reads, Writes}
import play.api.mvc.PathBindable

object MembershipAccessRight extends Enumeration {
  type MembershipAccessRight = Value
  val Readonly, ReadAndWrite, Owner = Value

  implicit val myEnumReads = Reads.enumNameReads(
     MembershipAccessRight)
  implicit val myEnumWrites = Writes.enumNameWrites

  implicit object searchTypeQueryStringBinder extends
     PathBindable.Parsing[MembershipAccessRight.
```

```
     MembershipAccessRight](
    withName, _.toString,
    (k: String, e: Exception) => "Cannot␣parse␣%s␣as␣
       SearchTypes:␣%s".format(k, e.getMessage)
  )
}
```

1.24 models/common/enums/RunningStatus.scala

```
package models.common.enums

import play.api.libs.json.{Reads, Writes}

object RunningStatus extends Enumeration {
  type RunningStatus = Value
  val Enabled, Disabled = Value

  implicit val myEnumReads = Reads.enumNameReads(RunningStatus)
  implicit val myEnumWrites = Writes.enumNameWrites
}
```

1.25 models/common/enums/RunningType.scala

```
package models.common.enums

import play.api.libs.json.{Reads, Writes}

object RunningType extends Enumeration {
  type RunningType = Value
  val Onetime, Periodic = Value

  implicit val myEnumReads = Reads.enumNameReads(RunningType)
  implicit val myEnumWrites = Writes.enumNameWrites
}
```

1.26 models/common/extensions.scala

```
package models.common

import play.api.libs.json._ // JSON library
import play.api.libs.json.Reads._ // Custom validation helpers
import play.api.libs.functional.syntax._ // Combinator syntax

import scala.concurrent.{Await, Future}
import scala.concurrent.duration.Duration

object extensions {

  implicit class RichFuture[T](future: Future[T]) {
    def awaitForResult: T = Await.result(future, Duration.Inf)
```

```
  }

  implicit class ValidateString(string: String) {
    def isEmail: Boolean = string.matches("""(\w+@\w+\.\w+)""")
  }

  implicit class ScrapydSettingJsValue(jsVal: Option[JsValue])
     {

    def toMap: Map[String, String] = {

      def mapToMap(v: JsValue): Map[String, String] = {
        Json.fromJson[Map[String, String]](v) match {
          case JsSuccess(value, _) => value
          case JsError(_) => Map.empty
        }
      }

      jsVal match {
        case Some(value) =>
          mapToMap(value)
        case None =>
          Map.empty
      }
    }
  }

}
```

1.27 models/common/PGProfile.scala

```
package models.common

import com.github.tminglei.slickpg._
import models.common.enums._
import slick.basic.Capability
import slick.jdbc.{JdbcCapabilities, PostgresProfile}

trait PGProfile extends PostgresProfile with PgEnumSupport with
    PgPlayJsonSupport {
  def pgjson = "jsonb"

  override protected def computeCapabilities: Set[Capability] =
    super.computeCapabilities + JdbcCapabilities.insertOrUpdate

  override val api = PostgresJsonSupportAPI

  object PostgresJsonSupportAPI extends API with JsonImplicits

  trait API extends super.API {
```

```
    implicit val weekDayTypeMapper = createEnumJdbcType("
       access_rights", MembershipAccessRight)
    implicit val weekDayListTypeMapper = createEnumListJdbcType
       ("access_rights", MembershipAccessRight)

    implicit val jobStatusTypeMapper = createEnumJdbcType("
       running_status", RunningStatus)
    implicit val jobStatusListTypeMapper =
       createEnumListJdbcType("running_status", RunningStatus)

    implicit val jobPriorityTypeMapper = createEnumJdbcType("
       job_priority", JobPriority)
    implicit val jobPriorityListTypeMapper =
       createEnumListJdbcType("job_priority", JobPriority)

    implicit val jobTypeTypeMapper = createEnumJdbcType("
       running_type", RunningType)
    implicit val jobTypeListTypeMapper = createEnumListJdbcType
       ("running_type", RunningType)

    implicit val jobEStatusTypeTypeMapper = createEnumJdbcType(
       "job_execution_status", JobExecutionStatus)
    implicit val jobEStatusTypeListTypeMapper =
       createEnumListJdbcType("job_execution_status",
       JobExecutionStatus)
  }
}

object PGProfile extends PGProfile
```

1.28 models/common/settings/ScrapydSettings.scala

```
package models.common.settings

/**
 * Class for representing scrapyd settings
 * @param setting  Array like value for field ("setting": ["
    DOWNLOAD_DELAY=2", "XYZ=Z"])
 * @param args Dictionary where keys and values specified by
    user ("arg1": "value2")
 */
case class ScrapydSettings(setting: Seq[String] = Seq.empty,
                           args: Map[String, String] = Map.
                              empty)
```

1.29 models/common/settings/SettingsFromDB.scala

```
package models.common.settings

import play.api.libs.json.JsValue
```

```
case class SettingsFromDB(settings: Option[JsValue] = None,
                          args: Option[JsValue] = None)
```

1.30 models/common/settings/SettingsMerger.scala

```
package models.common.settings

import models.common.extensions._
import play.api.libs.json.JsValue

object SettingsMerger {

  private def getPrioritized(projectSetting: Option[JsValue],
                             crawlerSettings: Option[JsValue],
                             jobExecutionSettings: Option[
                                JsValue]): Map[String, String]
                                = {

    val projectMap = projectSetting.toMap
    val crawlerMap = crawlerSettings.toMap
    val jobExecutionMap = jobExecutionSettings.toMap

    projectMap++crawlerMap++jobExecutionMap
  }

  def mergeSettings(projectSettings: SettingsFromDB,
                    crawlerSettings: SettingsFromDB,
                    jobExecutionSettings: SettingsFromDB):
                       ScrapydSettings = {


    val prioritizedSettings = getPrioritized(projectSettings.
       settings, crawlerSettings.settings, jobExecutionSettings
       .settings)
    val prioritizedArgs = getPrioritized(projectSettings.args,
       crawlerSettings.args, jobExecutionSettings.args)

    val mappedSettings = prioritizedSettings.map { case (k, v)
       => s"$k=$v" }.toSeq
    ScrapydSettings(mappedSettings, prioritizedArgs)
  }
}
```

1.31 models/Cookie.scala

```
package models

import play.api.libs.json.{Json, OFormat}
```

```
case class Cookie(cookie: String, login: String)

object Cookie {
  implicit val cookieFormat: OFormat[Cookie] = Json.format[
     Cookie]
}
```

1.32 models/daos/CrawlersDAO.scala

```
package models.daos

import java.util.UUID

import javax.inject.Inject
import models.common.PGProfile
import models.scrapyd\_response.ListSpidersResponse
import models.services.ProjectService
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}
import models.tables.Crawler.dbCrawlers
import models.tables.Crawler
import play.api.libs.json.JsValue

import scala.concurrent.{ExecutionContext, Future}

class CrawlersDAO @Inject() (protected val dbConfigProvider:
   DatabaseConfigProvider,
                             projectDAO: ProjectDAO) (implicit
                                ec: ExecutionContext) extends
                                HasDatabaseConfigProvider[
                                PGProfile] {

  import models.common.PGProfile.api._

  /* Adds spiders with names from response */
  def add(projectId: Long, spidersList: List[String]): DBIO[
     List[Crawler]] = {
    val insertCrawlers = dbCrawlers returning dbCrawlers.map(_.
       id)

    val spiders = spidersList.map { name =>
      Crawler(projectId = projectId, name = name)
    }

    (insertCrawlers ++= spiders)
      .map { ids =>
        spiders
          .zip(ids)
          .map { case (crawler, id) => crawler.copy(id = id) }
      }
```

```
  }

  // MARK: - Update

  private def updateAction(projectId: Long,
                           crawlerId: Long,
                           settings: Option[JsValue],
                           args: Option[JsValue]): DBIO[Int] =
                              {
    dbCrawlers
      .filter(c => c.projectId === projectId && c.id ===
         crawlerId)
      .map(c => (c.settings, c.args))
      .update((settings, args))
  }

  /* Updates settings for crawler `id` in project `projectId`
  * Also updates project `changedBy` and `changedAt`
  * Is calling justUpdate(changedBy: UUID, projectId: Long):
     DBIO[Int] */
  def update(projectId: Long,
             crawlerId: Long,
             settings: Option[JsValue],
             args: Option[JsValue],
             changedBy: UUID): Future[Int] = {
    db.run(
      (for {
        updsNumber <- updateAction(projectId, crawlerId,
           settings, args)
        _ <- projectDAO.justUpdate(changedBy, projectId)
      } yield (updsNumber)).transactionally)
  }

  /* without pagination */
  def get(projectId: Long): Future[Seq[Crawler]] = {
    db.run(
      dbCrawlers.filter(_.projectId === projectId).sortBy(_.
         name).result
    )
  }
}
```

1.33 models/daos/JobDAO.scala

```
package models.daos

import java.time.Instant
import java.util.UUID

import com.google.inject.Inject
```

```
import forms.periodic._
import models.common.PGProfile
import models.common.enums.{JobExecutionStatus, RunningStatus,
   RunningType}
import models.common.settings.{ScrapydSettings, SettingsFromDB,
    SettingsMerger}
import models.tables.Crawler.dbCrawlers
import models.tables.{Crawler, JobExecution, JobInstance}
import models.tables.JobExecution.dbJobExecutions
import models.tables.JobInstance.dbJobInstances
import models.tables.Project.dbProjects
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}

import scala.concurrent.{ExecutionContext, Future}

class JobDAO @Inject() (protected val dbConfigProvider:
   DatabaseConfigProvider) (implicit ec: ExecutionContext)
   extends HasDatabaseConfigProvider[PGProfile] {

  import models.common.PGProfile.api._

  private val insertJobInstanceQuery = dbJobInstances returning
      dbJobInstances.map(_.id)
  private val insertJobExecutionQuery = dbJobExecutions
     returning dbJobExecutions.map(jEx => (jEx.id, jEx.
     scrapydId))

  /** Adds onetime periodic job in DB: job instance and job
     execution. */
  def addAction(job: JobInstance): Future[(Long, UUID)] = {

    db.run(
      (for {
        insertedJobInstanceId <- insertJobInstanceQuery += job
        (id, scrapydId) <- insertJobExecutionQuery +=
           JobExecution(
          scrapydId = UUID.randomUUID(),
          jobInstanceId = insertedJobInstanceId,
          executionSettings = job.settings,
          executionArgs = job.args,
          createTime = Instant.now()
        )
      } yield ((id, scrapydId))).transactionally
    )
  }

  /** Adds periodic job in DB and returns its id */
  def addPeriodicJobAction(jobInstance: JobInstance): Future[
```

```
   Long] = {
  db.run(insertJobInstanceQuery += jobInstance)
}

/**
 * Adds job execution instance for existing job instance.
    Preserves given settings and args.
 *
 * @param existingJobInstance parent for execution
 * @return ids for created execution
 */
def addJobExecution(existingJobInstance: JobInstance): Future
   [(Long, UUID)] = {
  db.run (
    (for {
      (id, scrapydId) <- insertJobExecutionQuery +=
         JobExecution(
        scrapydId = UUID.randomUUID(),
        jobInstanceId = existingJobInstance.id,
        executionSettings = existingJobInstance.settings,
        executionArgs = existingJobInstance.args,
        createTime = Instant.now()
      )
    } yield (id, scrapydId)).transactionally)
}

/** Cache jobs to finished status.
 * Change only status value, not (startTime, endTime) */
def changeStatusActionToFinished(jobId: UUID): DBIO[Int] = {
  dbJobExecutions
    .filter(_.scrapydId === jobId )
    .map(_.status)
    .update(JobExecutionStatus.Finished)
}

/**
 * Finds job by given id
 * @param id job execution
 * @return Option of job execution instance found
 */
def findJobById(id: Long): DBIO[Option[JobExecution]] = {
  dbJobExecutions
    .filter(_.id === id)
    .result
    .headOption
    .transactionally
}

/**
```

```
  * Deletes executing job. If it was onetime job it also
     deletes connected jobInstance.
  *
  * @note Can be applied only for finished jobs.
  * @param jobExId id for job which will be deleted
  * @return either Id of deleted job or error string message
  */
 def deleteJob(jobExId: Long): Future[Either[String, Long]] =
    {

   def jobExecutionAction: DBIO[Int] = {
     dbJobExecutions
       .filter(_.id === jobExId)
       .delete
   }

   def jobInstanceAction(jId: Long): DBIO[Int] = {
     dbJobInstances
       .filter(_.id === jId)
       .delete
   }

   def onetimeJobExecutionRemoval(jId: Long): DBIO[Unit] = {
     (for {
       v <- jobExecutionAction
       _ <- jobInstanceAction(jId)
     } yield ()).transactionally
   }

   def periodicJobExecutionRemoval: DBIO[Unit] = {
     (for {
       _ <- jobExecutionAction
     } yield ()).transactionally
   }

   val q = for {
     (_,jobInst) <- dbJobExecutions
       .filter(j => j.id === jobExId && j.status ===
          JobExecutionStatus.Finished)
       .join(dbJobInstances).on(_.jobInstanceId === _.id)
   } yield (jobInst.id, jobInst.runType)

   val t = db.run(q.result.headOption)

   t.flatMap {
     case Some((jId, jRunType)) =>
       db.run((for {
         _ <- (if (jRunType == RunningType.Onetime)
            onetimeJobExecutionRemoval(jId) else
```

```
            periodicJobExecutionRemoval)
        } yield (Right(jobExId))).transactionally)
      case None =>
        Future(Left("There␣was␣no␣finished␣job␣with␣given␣id"))
    }
  }

  /**
   * Change periodic job data with new 'PeriodicJobChangeForm'
   * @param periodicJobId job id to be changed
   * @param data new data without running status
   * @return db inserted number of rows
   */
  def changePeriodicJobData(periodicJobId: Long, data:
     PeriodicJobChangeForm): Future[Int] = {

    val updateData = (data.settings, data.priority, data.
       crawlerId, Some(data.cronExpression),
      Some(data.title), data.description)

    db.run(
      dbJobInstances
        .filter(_.id === periodicJobId)
        .map(pJ => (pJ.settings, pJ.priority, pJ.spiderId, pJ.
           cron, pJ.title, pJ.description))
        .update(updateData)
        .transactionally
    )
  }

  /**
   * Get periodic JobInstance and corresponding crawler.
   * @param periodicJobId periodic job id
   * @return pair
   */
  def getCrawlerAndPeriodicJob(periodicJobId: Long): Future[(
     JobInstance, Crawler)] = {

    val q = for {
      (jobInstance, crawler) <-
        dbJobInstances
          .filter(p => p.id === periodicJobId && p.runType ===
             RunningType.Periodic)
          .join(dbCrawlers).on(_.spiderId === _.id)
    } yield (jobInstance, crawler)

    // I am pretty sure by that moment that periodicJobId is
       valid and there would be no exception
    db.run(q.result.head)
```

```
}

/**
 * Change running status of periodic job instance to onetime
    job instance.
 * This job instance will no longer be availiable as periodic
    jobInstance.
 * @param periodicJobId id job
 * @return Int Success or Failure of db
 */
def setOnetimeStatusForPeriodicJob(periodicJobId: Long):
   Future[Int] = {

  db.run(
    dbJobInstances
      .filter(_.id === periodicJobId)
      .map(_.runType)
      .update(RunningType.Onetime)
      .transactionally
  )
}

/**
 * Disable or enable scheduling of periodic job instance
 * @param periodicJobId job to be changed
 * @param newStatus disabled/enabled
 * @return Int Success or Failure of db
 */
def changeRunningStatusForPeriodicJob(periodicJobId: Long,
                                      newStatus:
                                         RunningStatus.
                                         RunningStatus):
                                         Future[Int] = {

  db.run(
    dbJobInstances
      .filter(_.id === periodicJobId)
      .map(_.status)
      .update(newStatus)
  )
}

/**
 * Merger for onetime job
 * @param projectId project id
 * @param crawlerId crawler id
 * @param jobId job execution id
 * @return settings for scrapyd, merged with priority
 */
```

```
  def mergeOnetimeSettings(projectId: Long,
                           crawlerId: Long,
                           jobId: Long): Future[ScrapydSettings
                              ] = {

    val q = for {
      crawler <- dbCrawlers.filter(c => c.id === crawlerId)
      project <- dbProjects.filter(p => p.id === projectId)
      jobExecution <- dbJobExecutions.filter(j => j.id ===
         jobId)
    } yield (crawler.settings, crawler.args, project.settings,
       project.args, jobExecution.executionSettings,
       jobExecution.executionArgs)

    for {
      settingsData <- db.run(q.result.head)
    } yield {
      settingsData match {
        case (crawlerSettings, crawlerArgs, projectSettings,
           projectArgs, jobSettings, jobArgs) =>
          SettingsMerger.mergeSettings(
            projectSettings = SettingsFromDB(projectSettings,
               projectArgs),
            crawlerSettings = SettingsFromDB(crawlerSettings,
               crawlerArgs),
            jobExecutionSettings = SettingsFromDB(jobSettings,
               jobArgs))
      }
    }
  }
}
```

1.34 models/daos/MembershipDAO.scala

```
package models.daos

import java.util.UUID

import javax.inject.Inject
import models.common.PGProfile
import models.common.enums.MembershipAccessRight._
import models.responses.{Member, UserData}
import models.tables.Membership.dbMembership
import models.tables.User.dbUsers
import models.tables.{Membership, User}
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}

import scala.concurrent.{ExecutionContext, Future}
```

```
class MembershipDAO @Inject() (protected val dbConfigProvider:
   DatabaseConfigProvider) (implicit ec: ExecutionContext)
   extends HasDatabaseConfigProvider[PGProfile] {
  import PGProfile.api._

  /**
   * Insert or update existing membership with new accessRight
      value
   * @param userId user which accessRights will be updated
   * @param projectId project for which membership will be
      updated/created
   * @param accessRight new access rights for user in project
   * @return number of rows inserted/updated
   */
  def insertOrUpdateMembership(userId: UUID, projectId: Long,
     accessRight: MembershipAccessRight): DBIO[Int] =
    dbMembership.insertOrUpdate(Membership(userId, projectId,
       accessRight))

  /**
   * Filters membership table to get people with any access to
      specified project.
   * @param projectId project to get members
   * @return members of specified project
   */
  def getMembers(projectId: Long): Future[Seq[Member]] = {
    val q = (for {
      (membership, user) <- dbMembership
        .filter(m => m.projectId === projectId)
        .join(dbUsers).on(_.userId === _.id)
    } yield (user.id, user.name, user.login, user.email,
       membership.accessRight))

    db
      .run(q.result)
      .map(f => f.map { case (id, name, login, email,
         accessRight) => Member(UserData(id, name, email, login
         ), accessRight)})
  }

  /**
   * Deletes occurrence of membership with given parameters.
   * @param projectId project in membership
   * @param userId user in membership
   * @return number of deleted occurrences(1)
   */
  def deleteMember(projectId: Long, userId: UUID): Future[Int]
     = {
    val deleteAction = dbMembership
```

```
      .filter(membership => membership.userId === userId &&
        membership.projectId === projectId)
      .delete

    db.run(deleteAction)
  }
}
```

1.35 models/daos/PasswordDAO.scala

```
package models.daos

import com.mohiva.play.silhouette.api.LoginInfo
import com.mohiva.play.silhouette.api.util.PasswordInfo
import com.mohiva.play.silhouette.persistence.daos.
   DelegableAuthInfoDAO
import javax.inject.Inject
import models.tables.Password
import models.tables.Password._
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}
import slick.jdbc.JdbcProfile

import scala.concurrent.{ExecutionContext, Future}
import scala.reflect.ClassTag

class PasswordDAO @Inject()(protected val dbConfigProvider:
   DatabaseConfigProvider)(implicit ec: ExecutionContext)
  extends DelegableAuthInfoDAO[PasswordInfo]
    with HasDatabaseConfigProvider[JdbcProfile] {

  import profile.api._

  override val classTag: ClassTag[PasswordInfo] = scala.reflect
     .classTag[PasswordInfo]

  override def find(loginInfo: LoginInfo): Future[Option[
     PasswordInfo]] =
    db.run (
      dbPasswords
        .filter(password => password.key === loginInfo.
           providerKey)
        .result
        .headOption
        .map(_.map(el => PasswordInfo(el.hasher, el.hash, el.
           salt)))
    )

  override def add(loginInfo: LoginInfo, authInfo: PasswordInfo
     ): Future[PasswordInfo] =
```

```
    db.run(
      dbPasswords += Password(loginInfo.providerKey, authInfo.
         hasher, authInfo.password, authInfo.salt)
    ).map( _ => {
      authInfo
    })

  override def update(loginInfo: LoginInfo, authInfo:
     PasswordInfo): Future[PasswordInfo] = {
    val q = for {
      pass <- dbPasswords if pass.key === loginInfo.providerKey
    } yield (pass.hash, pass.hasher, pass.salt)

    db.run(q.update(authInfo.hasher, authInfo.password,
       authInfo.salt)).map(_ => authInfo)
  }

  override def save(loginInfo: LoginInfo, authInfo:
     PasswordInfo): Future[PasswordInfo] =
    find(loginInfo).flatMap {
    case Some(_) => update(loginInfo, authInfo)
    case None => add(loginInfo, authInfo)
  }

  override def remove(loginInfo: LoginInfo): Future[Unit] = db.
     run(
    dbPasswords.filter(password => password.key === loginInfo.
       providerKey).delete
  ).map( _ => ())
}
```

1.36 models/daos/ProjectDAO.scala

```
package models.daos

import java.time.Instant
import java.util.UUID

import forms.project.{ProjectChangeForm, ProjectForm}
import javax.inject.Inject
import models.common.PGProfile
import models.common.enums.MembershipAccessRight
import models.tables.User
import models.tables.Project
import models.tables.Project.dbProjects
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}
import play.api.libs.json.Json

import scala.concurrent.{ExecutionContext, Future}
```

```
class ProjectDAO @Inject()(protected val dbConfigProvider:
   DatabaseConfigProvider,
                           membershipDAO: MembershipDAO) (
                              implicit ec: ExecutionContext)
                              extends HasDatabaseConfigProvider
                              [PGProfile] {

  import models.common.PGProfile.api._


  def create(projectForm: ProjectForm, createdBy: User): Future
     [Long] = {

    val projectInsertion = dbProjects returning dbProjects.map(
       _.id)
    val newProject = Project(
      name = projectForm.name,
      description = projectForm.description,
      ownerId = createdBy.id,
      createdAt = Instant.now(),
      changedBy = createdBy.id,
      changedAt = Instant.now()
    )

    db.run(
      (for {
        pId <- projectInsertion += newProject
        _ <- membershipDAO.insertOrUpdateMembership(createdBy.
           id, pId, MembershipAccessRight.Owner)
      } yield (pId)).transactionally
    )
  }

  def find(id: Option[Long]): Future[Option[Project]] = {
    db.run(dbProjects.filter(_.id === id).result.headOption)
  }

  def updateMetadataAction(changedBy: UUID, projectId: Long, p:
      ProjectChangeForm): DBIO[Int] = {
    val settings = Some(Json.toJson(p.spiderSettings))
    val args = Some(Json.toJson(p.spiderArgs))

    dbProjects
      .filter(_.id === projectId)
      .map(project => (project.name, project.description,
         project.settings, project.args, project.changedBy,
         project.changedAt))
```

```
      .update((p.name, p.description, settings, args, changedBy
        , Instant.now()))
  }

  def deployAction(changedBy: UUID, projectId: Long, egg:
     Option[Array[Byte]]): DBIO[Int] =
    dbProjects
      .filter(_.id === projectId)
      .map(p => (p.eggfile, p.changedBy, p.changedAt))
      .update((egg, changedBy, Instant.now()))

  /** Can be called in cases of updates in job statuses and
     crawlers settings, etc. */
  def justUpdate(changedBy: UUID, projectId: Long): DBIO[Int] =
      {
    dbProjects
      .filter(_.id === projectId)
      .map(p => (p.changedBy, p.changedAt))
      .update((changedBy, Instant.now()))
  }

  /** Delete project and all membership occurrences.
   * Deletion can be performed only by owner
   * ===Note===
   * Membership will be deleted automatically in postgresql.*/
  def delete(projectId: Long): Future[Int] = {
    db.run((for {
      deletedNumber <-
        dbProjects
        .filter(_.id === projectId)
        .delete
    } yield (deletedNumber)).transactionally)
  }

}
```

1.37 models/responses/Job.scala

```
package models.responses

import java.time.Instant
import java.util.UUID

import io.swagger.annotations.ApiModel
import models.common.enums.JobExecutionStatus.
   JobExecutionStatus
import models.common.enums.JobPriority.JobPriority
import models.common.enums.RunningType.RunningType
import play.api.libs.json.{Json, OFormat}
```

```
object ProjectData {
  implicit val r4Format: OFormat[ProjectData] = Json.format[
     ProjectData]
}

object SpiderData {
  implicit val r3Format: OFormat[SpiderData] = Json.format[
     SpiderData]
}

object Job{
  implicit val r2Format: OFormat[Job] = Json.format[Job]
}

@ApiModel(description = "Project␣data")
case class ProjectData(id: Long, name: String)

@ApiModel(description = "Crawler␣data")
case class SpiderData(id: Long, name: String)

@ApiModel(description = "Struct␣for␣‘listJob‘␣response")
case class Job(id: Long,
               scrapydId: UUID,
               jobInstanceId: Long,
               status: JobExecutionStatus,
               priority: JobPriority,
               //name: String,
               runningType: RunningType,

               startTime: Option[Instant],
               endTime: Option[Instant],

               spider: SpiderData,
               project: ProjectData)
```

1.38 models/responses/Member.scala

```
package models.responses

import java.util.UUID

import io.swagger.annotations.ApiModel
import models.common.enums.MembershipAccessRight.
   MembershipAccessRight
import play.api.libs.json.{Json, OFormat}


object Member {
  implicit val r3Format: OFormat[Member] = Json.format[Member]
```

```
}

object UserData {
  implicit val r2Format: OFormat[UserData] = Json.format[
    UserData]
}

@ApiModel(description = "Important␣user␣data")
case class UserData(id: UUID,
                    name: String,
                    email: String,
                    login: String)

@ApiModel(description = "Member␣format")
case class Member(user: UserData,
                  accessRight: MembershipAccessRight)
```

1.39 models/responses/SimpleJob.scala

```
package models.responses

import java.util.UUID

import io.swagger.annotations.ApiModel
import play.api.libs.json.{Json, OFormat}

object SimpleJob {
  implicit val r4Format: OFormat[SimpleJob] = Json.format[
    SimpleJob]
}

@ApiModel(description = "ID␣and␣ScrapydId␣for␣job")
case class SimpleJob(id: Long,
                     scrapydId: UUID)
```

1.40 models/scrapyd_response/AddVersionResponse.scala

```
package models.scrapyd\_response

import play.api.libs.json.{Json, OFormat}

case class AddVersionResponse(status: String,
                              spiders: Int)

object AddVersionResponse {
  implicit val rFormat: OFormat[AddVersionResponse] = Json.
    format[AddVersionResponse]
}
```

1.41 models/scrapyd_response/CancelJobResponse.scala

```
package models.scrapyd\_response

import play.api.libs.json.{Json, OFormat}

case class CancelJobResponse(status: String,
                             prevstate: Option[String])

object CancelJobResponse {
  implicit val rFormat: OFormat[CancelJobResponse] = Json.
     format[CancelJobResponse]
}
```

1.42 models/scrapyd_response/GeneralResponse.scala

```
package models.scrapyd\_response

import play.api.libs.json.{Json, OFormat}

case class GeneralResponse(status: String)

object GeneralResponse {
  implicit val rFormat: OFormat[GeneralResponse] = Json.format[
     GeneralResponse]
}
```

1.43 models/scrapyd_response/JobScrapyd.scala

```
package models.scrapyd\_response


import java.text.SimpleDateFormat
import java.time.Instant
import java.util.UUID

import play.api.libs.json.{Format, JsString, JsSuccess, Json,
   OFormat, Reads, Writes}

object JobScrapyd {
  implicit val r1Format: OFormat[JobScrapyd] = {
    val format = new SimpleDateFormat("yyyy-MM-dd␣HH:mm:ss.
       SSSSSS")

    implicit val customLocalDateFormat: Format[Instant] =
       Format(
      Reads(js => JsSuccess(format.parse(js.as[String]).
         toInstant)),
      Writes(d => JsString(d.toString)))
    Json.format[JobScrapyd]
  }
}
```

```
case class JobScrapyd(id: UUID,
                      spider: String,
                      start_time: Option[Instant],
                      end_time: Option[Instant])
```

1.44 models/scrapyd_response/ListJobsResponse.scala

```
package models.scrapyd\_response

import models.tables.JobInstance
import play.api.libs.json.{Json, OFormat}

object ListJobsResponse {
  implicit val rFormat: OFormat[ListJobsResponse] = Json.format
     [ListJobsResponse]
}

case class ListJobsResponse(status: String,
                            pending: List[JobScrapyd],
                            running: List[JobScrapyd],
                            finished: List[JobScrapyd])
```

1.45 models/scrapyd_response/ListSpidersResponse.scala

```
package models.scrapyd\_response

import play.api.libs.json.{Json, OFormat}

/* 'spiders' contains list of project's spiders names from egg
   file, unchangeable */
case class ListSpidersResponse(status: String,
                               spiders: List[String])

object ListSpidersResponse {
  implicit val rFormat: OFormat[ListSpidersResponse] = Json.
     format[ListSpidersResponse]
}
```

1.46 models/scrapyd_response/ScheduleResponse.scala

```
package models.scrapyd\_response

import java.util.UUID

import play.api.libs.json.{Json, OFormat}

case class ScheduleResponse(status: String,
                            jobid: UUID)
```

```
object ScheduleResponse {
  implicit val rFormat: OFormat[ScheduleResponse] = Json.format
     [ScheduleResponse]
}
```

1.47 models/services/JobService.scala

```
package models.services

import java.util.{Date, UUID}

import akka.actor._
import akka.util.Timeout

import scala.concurrent.duration._
import akka.pattern.ask
import com.google.inject.name.Named
import com.google.inject.{Inject, Singleton}
import com.typesafe.akka.extension.quartz.
   QuartzSchedulerExtension
import forms.periodic.{PeriodicJobChangeForm,
   PeriodicJobCreateForm}
import forms.{OnetimeJobForm, Request}
import models.actors.JobSchedulerActor.{ScheduleJob,
   ScheduleJobRepeatedly}
import models.common.PGProfile
import models.common.enums._
import models.daos.JobDAO
import models.responses.{Job, ProjectData, SimpleJob,
   SpiderData}
import models.scrapyd\_response.CancelJobResponse
import models.tables.{Crawler, JobInstance}
import models.tables.Crawler.dbCrawlers
import models.tables.Project.dbProjects
import models.tables.JobExecution.dbJobExecutions
import models.tables.JobInstance.{JobInstanceTable,
   dbJobInstances}
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}
import play.api.libs.json.{JsError, JsResult, JsSuccess, Json}
import models.common.enums.RunningStatus
import org.quartz.CronExpression

import scala.concurrent.{ExecutionContext, Future}

/**
 * Job Service for business logic for Onetime jobs and periodic
    jobs.
 * @param dbConfigProvider for db
 * @param system actor system
```

```
 * @param jobActor actor for scheduling job
 * @param scrapydService service to get to scrapyd api
 * @param updaterService use for get data logic
 * @param projectService for project stuff
 * @param jobDAO direct call to db
 * @param ec execution context
 */
@Singleton
class JobService @Inject() (protected val dbConfigProvider:
   DatabaseConfigProvider,
                            system: ActorSystem,
                            @Named("job-actor") jobActor:
                               ActorRef,
                            scrapydService: ScrapydService,
                            updaterService: UpdaterService,
                            projectService: ProjectService,
                            jobDAO: JobDAO)(implicit ec:
                               ExecutionContext) extends
                               HasDatabaseConfigProvider[
                               PGProfile] {

  import models.common.PGProfile.api._

  /** Timout for job actor */
  implicit val timeout: Timeout = Timeout(5 seconds)
  /** Scheduler for periodic jobs. See github Akka-quartz-
     scheduler */
  val scheduler: QuartzSchedulerExtension =
     QuartzSchedulerExtension(system)
  /** Name for periodic job */
  private def nameForPeriodicJob(id: Long): String = s"$id-
     periodic-job"

  /**
   * Schedule onetime crawler, in future this func can be
      modified to be used with scheduler.
   * @note 1. Check if given project has given crawlerId,
   *       2. Go to scrapyd and execute job,
   *       3. Add data about jobInstance and jobExecution to DB
   * @param crawler crawler for which job is run
   * @param projectId project of the job
   * @param userId user performing action
   * @param jobForm job data to schedule
   * @return either string error or (Long, UUID) with IDs of
      created jobs
   */
  def scheduleCrawler(crawler: Crawler, projectId: Long, userId
     : UUID, jobForm: OnetimeJobForm): Future[Either[String,
     SimpleJob]] = {
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
  val settings = Some(Json.toJson(jobForm.settings))
  val args = Some(Json.toJson(jobForm.args))

  val job = JobInstance(
    projectId = projectId,
    settings = settings,
    args = args,
    priority = jobForm.priority,
    status = RunningStatus.Enabled,
    spider = jobForm.crawlerId,
    runType = RunningType.Onetime)

  (jobActor ? ScheduleJob(crawler, job)).mapTo[Either[String,
      SimpleJob]]
}

/**
 * Simple deletion from DB only for jobs with finished status
    .
 * @note Do not need to go to scrapyd at all.
 * @param jobId job to delete
 * @return either id of deleted job or error message
 */
def delete(jobId: Long): Future[Either[String, Long]] = {
  jobDAO.deleteJob(jobId)
}

/**
 * Changes execution status for running and pending jobs.
 * In case of changing the wrong type of job (finished) -
    return the error.
 * @param jobId job to be modified
 * @param projectId project for job
 * @return Either Success or Failure
 */
def changeRunningStatus(jobId: UUID, projectId: Long): Future
   [Either[String, String]] = {

  /** Change job status anyway */
  val cancelJobResponseMapper: JsResult[CancelJobResponse] =>
      Future[Either[String, String]] = {
    case JsSuccess(CancelJobResponse(_, _), _) =>
      db.run(jobDAO.changeStatusActionToFinished(jobId)).
         flatMap(_ => Future(Right("Successfully␣changed")))
    case JsError(errors) =>
      Future(Left(errors.toString()))
  }
```

```
    updaterService.fetchAndUpdateData(Seq(projectId)).flatMap {
        _ =>
      scrapydService
        .cancelJob(projectId, jobId)
        .flatMap(cancelJobResponseMapper)
    }
  }


  /**
   * Go to __scrapyd__ to get list of **one of a kind** of jobs
       stored = {pending, running, finished}.
   * Go to my data base and update uuid-s with new statuses.
   *
   * @param r request with pagination parameters
   * @param userId to get projects for user
   * @param status job status
   * @return tuple of total number of jobs and seq of jobs with
       pagination
   */
  def get(r: Request[Long], userId: UUID, status:
     JobExecutionStatus.JobExecutionStatus): Future[(Long, Seq[
     Job])] = {

    /** Constructing response from joining already updated db
       tables */
    def getResponseAction(projectId: Long): Future[Seq[Job]] =
       {

      val jobsDataQuery = for {
        (((p, jInst), jExec), c) <- dbProjects
          .filter(_.id === projectId)
          .join(dbJobInstances).on(_.id === _.projectId)
          .join(dbJobExecutions).on { case ((_ , jInst), jExec)
              => jInst.id === jExec.jobInstanceId && jExec.
            status === status } //_._2.id === _.jobInstanceId
          .join(dbCrawlers).on(_._1._2.spiderId === _.id)
      } yield (jExec.id, jExec.scrapydId, jInst.id, jExec.
         status, jInst.priority, jInst.runType, jExec.startTime
         , jExec.endTime, c.id, c.name, p.id, p.name)

      for {
        jobsData <- db.run(jobsDataQuery.result.transactionally
           )
      } yield {
        jobsData.map { case (id, scrapydId, jobInstanceId, s,
           prior, runT, sT, eT, cId, cName, pId, pName) =>
          Job(
            id = id,
```

```
            scrapydId = scrapydId,
            jobInstanceId = jobInstanceId,
            status = s,
            priority = prior,
            runningType = runT,
            startTime = sT,
            endTime = eT,
            spider = SpiderData(cId, cName),
            project = ProjectData(pId, pName))
        }
      }
    }

    def getPaginatedResult(jobs: Seq[Job], exclusiveFrom:
       Option[Long]): (Long, Seq[Job]) = {

      val total = jobs.length
      def f(limit: Int, value: Long): (Seq[Job] => Seq[Job]) =
         { s => f1(limit)(s.filter(e => e.id < value)) }
      def f1(limit: Int): (Seq[Job] => Seq[Job]) = { s => s.
         sortBy(_.id)(Ordering[Long].reverse).take(limit) }

      exclusiveFrom match {
        case Some(value) => (total, f(r.limit, value)(jobs))
        case None => (total, f1(r.limit)(jobs))
      }
    }

    /* After fetching user's projects ids, we pass 'pIds' to
       fetch and update projects' jobs and their data in DB.
    * We also pass mapping function to get Seq[Jobs] in return.
    * After that we pass Seq[Jobs] to get paginated result to
       return.  */
    projectService
      .getProjectsForUser(userId)
      .flatMap { pIds => updaterService.fetchAndUpdateData(pIds
         , (pId: Long) => getResponseAction(pId)) }
      .flatMap(Future.sequence(_))
      .map(e => getPaginatedResult(e.flatten, r.exclusiveFrom))
  }

  /**
   * GET request with pagination for periodic jobs
   * @param projectId project in which request is performed
   * @param request params for pagination
   * @return seq of periodic jobs in a specified project
   */
  def getJobInstances(projectId: Long, request: Request[Long]):
      Future[Seq[JobInstance]] = {
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
  val filterPredicate: JobInstanceTable => Rep[Boolean] = {
     jInst =>

    val defaultCondition = jInst.projectId === projectId &&
       jInst.runType === RunningType.Periodic

    request.exclusiveFrom match {
      case Some(value) => defaultCondition && jInst.id <
         value
      case None => defaultCondition
    }
  }

  db.run (
    dbJobInstances
      .filter(filterPredicate)
      .sortBy(_.id.desc)
      .take(request.limit)
      .result
      .transactionally
  )
}

/**
 * Construction to check for condition of scheduler and
    perform action on success.
 * @param errorMessage message to be returned in Left
 * @param checker boolean func to check some condition before
     performing action
 * @param ifSuccess action to be performed on successful
    condition
 * @tparam T for return type
 * @return
 */
def checkSchedulerAndPerformAction[T](errorMessage: String)(
   checker: () => Boolean)(ifSuccess: () => Future[Either[
   String, T]]): Future[Either[String, T]] = {
  if (checker()) {
    ifSuccess()
  } else {
    Future(Left(errorMessage))
  }
}

/**
 * Creation of scheduler
 * @param crawler crawler
 * @param periodicJob periodic job
```

```
 * @param cronExpression cron
 * @return Date when first task will be scheduled
 */
def createScheduler(crawler: Crawler,
                    periodicJob: JobInstance,
                    cronExpression: String): Date = {
  scheduler
    .createJobSchedule(name = nameForPeriodicJob(periodicJob.
       id), jobActor,
      ScheduleJobRepeatedly(crawler, periodicJob),
      cronExpression = cronExpression)
}

/**
 * Create periodic job instance and adds it to DB.
 * 1. Check if given crawlerId corresponds to projectId.
 * 2. Check if cron expression is valid.
 * 3. Add periodic job (jobInstance) to DB.
 * 4. Actor for scheduling
 *
 * @param projectId project id
 * @param crawler crawler which will be performing crawling
 * @param pJobForm job data to be scheduled
 * @return Success id or Failure string message
 */
def createPeriodicJobInstance(projectId: Long,
                              crawler: Crawler,
                              pJobForm: PeriodicJobCreateForm
                                 ): Future[Either[String,
                                 Long]] = {

  val checker = () => org.quartz.CronExpression.
     isValidExpression(pJobForm.cronExpression)

  def addPeriodicJob(): Future[Either[String, Long]] = {
    val pJob = JobInstance(
      projectId = projectId,
      title = Some(pJobForm.title),
      description = pJobForm.description,
      settings = pJobForm.settings,
      args = pJobForm.args,
      priority = pJobForm.priority,
      status = pJobForm.status,
      spider = pJobForm.crawlerId,
      runType = RunningType.Periodic,
      cron = Some(pJobForm.cronExpression)
    )

    jobDAO.addPeriodicJobAction(pJob).flatMap {
```

```
      id =>
        createScheduler(crawler, pJob.copy(id = id), pJobForm
           .cronExpression)
        Future(Right(id))
    }
  }

  checkSchedulerAndPerformAction("invalid␣cron␣expression")(
     checker)(() => addPeriodicJob())
}

/**
 * Changes periodic job parameters except `RunningStatus`
 * @param periodicJobId job to be modified
 * @param pJobForm new data
 * @return result of successful db insertion
 */
def changePeriodicJobInstance(periodicJobId: Long,
                              pJobForm: PeriodicJobChangeForm
                                 ): Future[Either[String,
                                 Option[Date]]] = {

  val updateSchedulerMapping = (jobCrawler: (JobInstance,
     Crawler)) => {

    jobCrawler._1.status match {
      case RunningStatus.Disabled => None
      case RunningStatus.Enabled =>
        Some(scheduler.updateJobSchedule(name =
           nameForPeriodicJob(periodicJobId), receiver =
           jobActor,
          msg = ScheduleJobRepeatedly(jobCrawler._2,
             jobCrawler._1), cronExpression = pJobForm.
             cronExpression))
    }
  }

  val updateSchedulerAction: Int => Future[Either[String,
     Option[Date]]] = { _ =>
    jobDAO
      .getCrawlerAndPeriodicJob(periodicJobId)
      .map(updateSchedulerMapping.andThen(Right(_)))
  }

  val onSuccess = () => jobDAO.changePeriodicJobData(
     periodicJobId, pJobForm).flatMap(updateSchedulerAction)
  val checker = () => org.quartz.CronExpression.
     isValidExpression(pJobForm.cronExpression)
```

```
    checkSchedulerAndPerformAction("invalid␣cron␣expression")(
      checker)(onSuccess)
  }

  /**
   * Changes 'JobExecutionStatus' to Onetime job instance in
     order to
   * save all the information for previously scheduled job
     executions.
   *
   * Check if job has been already disabled.
   *
   * Cancels all of the future JobExecutions.
   * @param periodicJobId job id to delete
   * @return Either error message or success
   */
  def deletePeriodicJobInstance(periodicJobId: Long): Future[
     Either[String, Int]] = {

    for {
      (job, _) <- jobDAO.getCrawlerAndPeriodicJob(periodicJobId
         )

      checker = () => scheduler.deleteJobSchedule(name =
         nameForPeriodicJob(periodicJobId)) || job.status ==
         RunningStatus.Disabled
      onSuccess = () => jobDAO.setOnetimeStatusForPeriodicJob(
         periodicJobId).map(Right(_))

      result <- checkSchedulerAndPerformAction("JobCouldn'
         tBeDeleted")(checker)(onSuccess)
    } yield result
  }

  /**
   * Permanently disable scheduling of new job executions.
   * @param periodicJobId job to disable scheduling
   * @return Either error message or success
   */
  def disableScheduling(periodicJobId: Long): Future[Either[
     String, Int]] = {

    val checker = () => scheduler.deleteJobSchedule(name =
       nameForPeriodicJob(periodicJobId))
    val onSuccess = () => jobDAO.
       changeRunningStatusForPeriodicJob(periodicJobId,
       RunningStatus.Disabled).map(Right(_))

    checkSchedulerAndPerformAction("JobCouldn'tBeDisabled")(
```

```
      checker)(onSuccess)
  }

  /**
   * Enable periodic job execution.
   * Start actor again.
   * @param periodicJobId periodic job to be enabled
   * @return Success or Failure
   */
  def enableScheduling(periodicJobId: Long): Future[Either[
     String, Date]] = {

    jobDAO
      .changeRunningStatusForPeriodicJob(periodicJobId,
         RunningStatus.Enabled)
      .flatMap { _ =>
        jobDAO
          .getCrawlerAndPeriodicJob(periodicJobId)
          .map { case (pJob, crawler) => Right(createScheduler(
             crawler, pJob, pJob.cron.get)) }
      }
  }
}
```

1.48 models/services/MembershipService.scala

```
package models.services

import java.util.UUID

import javax.inject.Inject
import models.common.PGProfile
import models.common.enums.MembershipAccessRight.
   MembershipAccessRight
import models.daos.MembershipDAO
import models.responses.Member
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}

import scala.concurrent.{ExecutionContext, Future}

/**
 * Membership service to call from controller.
 * @param dbConfigProvider for db
 * @param membershipDAO access to db functions
 * @param ec ExecutionContext
 */
class MembershipService @Inject()(protected val
   dbConfigProvider: DatabaseConfigProvider,
                                  membershipDAO: MembershipDAO)
```

```
                                        (implicit ec:
                                       ExecutionContext) extends
                                       HasDatabaseConfigProvider[
                                       PGProfile] {

  /**
   * Get memberships from db without pagination.
   * @param projectId project
   * @return sequence of members
   */
  def get(projectId: Long): Future[Seq[Member]] =
    membershipDAO.getMembers(projectId)

  /**
   * Deletes user's membership in specified project from db.
   * @param projectId project
   * @param userId user
   * @return count of deleted occurrences
   */
  def delete(projectId: Long, userId: UUID): Future[Int] =
    membershipDAO.deleteMember(projectId, userId)

  /**
   * Updates or creates membership for user and project with
      specified access rights.
   * @param projectId project
   * @param userId user
   * @param accessRight permission
   * @return count of successfully completed inserts/updates
   */
  def put(projectId: Long, userId: UUID, accessRight:
     MembershipAccessRight): Future[Int] =
    db.run(membershipDAO.insertOrUpdateMembership(userId,
       projectId, accessRight))
}
```

1.49 models/services/ProjectService.scala

```
package models.services

import java.nio.file.{Files, Path}
import java.util.UUID

import forms.project.{ProjectChangeForm, ProjectForm}
import forms.Request
import javax.inject.Inject
import models.common.PGProfile
import models.daos.{CrawlersDAO, ProjectDAO}
import models.scrapyd\_response.{AddVersionResponse,
   ListSpidersResponse}
```

```
import models.tables.{Crawler, Project, User}
import models.tables.Membership.dbMembership
import models.tables.Project.dbProjects
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}
import play.api.libs.json.{JsError, JsSuccess}

import scala.concurrent.{ExecutionContext, Future}

/**
 * Project service to be called from project controller
 * for business logic method on ‘project‘.
 * @param dbConfigProvider for db
 * @param crawlersDAO crawlers db access
 * @param projectDAO projects db access
 * @param scrapydService for calling scrapyd api methods
 * @param ec ExecutionContext
 */
class ProjectService @Inject()(protected val dbConfigProvider:
   DatabaseConfigProvider,
                               crawlersDAO: CrawlersDAO,
                               projectDAO: ProjectDAO,
                               scrapydService: ScrapydService)(
                                  implicit ec: ExecutionContext
                                  ) extends
                                  HasDatabaseConfigProvider[
                                  PGProfile] {

  import models.common.PGProfile.api._

  /**
   * Creation of project
   * @param projectForm project data
   * @param createdBy creator
   * @return id of created project
   */
  def create(projectForm: ProjectForm, createdBy: User): Future
     [Long] = projectDAO.create(projectForm, createdBy)

  /// MARK: - Update actions for project's metadata and eggfile

  /**
   * Update metadata of the project (name, description)
   * @param projectId project to be updated
   * @param form new data
   * @param byUser updater
   * @return Success
   */
  def updateMetadataContent(projectId: Long, form:
```

```
    ProjectChangeForm, byUser: User): Future[Int] = {
  db.run(projectDAO.updateMetadataAction(byUser.id, projectId
     , form))
}

/**
 * Method to be used in deploying eggfile.
 * @param changedBy changer
 * @param projectId project
 * @param egg archive with crawlers and project
 * @param spiders names of spiders
 * @return seq of crawlers deployed
 */
def deployToDB(changedBy: UUID, projectId: Long, egg: Option[
   Array[Byte]], spiders: List[String]): Future[Seq[Crawler]]
    = {
  db.run((for {
      _ <- projectDAO.deployAction(changedBy, projectId, egg)
      spiders <- crawlersDAO.add(projectId, spiders)
    } yield(spiders)).transactionally)
}

/**
 * Method for deploying eggFile to scrapyd:
 * 1. Go to scrapyd `addVersion` endpoint
 * 2. Go to scrapyd `listSpiders` endpoint
 * 3. Add data to db by calling `deployToDB` method
 *
 * @param eggFile archive with which project will be deployed
 * @param projectId project
 * @param changedBy changer
 * @return either error messages or seq of deployed crawlers
 */
def deployEggFile(eggFile: Path, projectId: Long, changedBy:
   UUID): Future[Either[String, Seq[Crawler]]] = {
  val egg = Files.readAllBytes(eggFile)

  def getCrawlers: Future[Either[String, Seq[Crawler]]] = {
    scrapydService.listSpiders(projectId).flatMap {
      case JsSuccess(ListSpidersResponse(_, spiders), _) =>
        deployToDB(changedBy, projectId, Some(egg), spiders).
           flatMap(r => Future(Right(r)))
      case JsError(errors) =>
        Future(Left(errors.toString()))
    }
  }

  def checkResponseAndGetCrawlers(response:
     AddVersionResponse): Future[Either[String, Seq[Crawler
```

```
    ]]] = {
    response match {
      case AddVersionResponse("ok", _) =>
        getCrawlers
      case _ =>
        Future(Left("Something␣went␣wrong␣with␣deploy␣to␣
           scrapyd"))
    }
  }

  scrapydService.addVersion(projectId, eggFile).flatMap {
    case JsSuccess(value, _) =>
      checkResponseAndGetCrawlers(value)
    case JsError(errors) =>
      Future(Left(errors.toString()))
  }
}

/// MARK: - Get request


/**
 * Projects (return `Seq[UUID]`) which user(`userId: UUID`)
    has access to.
 * We need projects with at least readonly access, that't why
     it is not specified.
 *
 * @param userId user
 * @return sequence of projects
 */
def getProjectsForUser(userId: UUID): Future[Seq[Long]] = {
  db.run((for {
    membership <- dbMembership.filter(_.userId === userId)
  } yield membership.projectId).result.transactionally)
}

/**
 * Method for GETing projects with pagination
 * @param request specifies limit of the request and optional
     starting point
 * @param userId to get projects for particular user
 * @return seq of projects
 */
def get(request: Request[Long], userId: UUID): Future[Seq[
   Project]] = {

  def getProjects(pIds: Seq[Long]): DBIO[Seq[Project]] = {
    request.exclusiveFrom match {
      case Some(value) =>
```

```
          dbProjects.filter { project => project.id < value &&
            project.id.inSet(pIds) }.sortBy(_.id.desc).take(
            request.limit).result
        case None =>
          dbProjects.filter { project => project.id.inSet(pIds)
            }.sortBy(_.id.desc).take(request.limit).result
      }
    }

    getProjectsForUser(userId).flatMap { pIds =>
      db.run(getProjects(pIds))
    }
  }

  /**
   * Deletion of existing project by id
   * @param projectId project id to delete
   * @return Success of deletion
   */
  def delete(projectId: Long): Future[Int] = projectDAO.delete(
    projectId)

}
```

1.50 models/services/ScrapydService.scala

```
package models.services

import java.nio.file.Path
import java.util.UUID

import akka.stream.scaladsl.FileIO
import models.scrapyd\_response.{AddVersionResponse,
  CancelJobResponse, GeneralResponse, ListJobsResponse,
  ListSpidersResponse, ScheduleResponse}
import play.api.libs.json.JsResult
import play.api.libs.ws.WSClient
import javax.inject.Inject

import scala.concurrent.{ExecutionContext, Future}

/**
 * Service to send calls to scrapyd server.
 * @param ws ws client
 * @param ec ExecutionContext
 */
class ScrapydService @Inject()(ws: WSClient)(implicit ec:
  ExecutionContext) {

  object URL {
```

```
  val baseUrl = "http://localhost:6800"

  val addVersionUrl = s"$baseUrl/addversion.json"
  val listSpidersUrl = s"$baseUrl/listspiders.json"
  val delProjectUrl = s"$baseUrl/delproject.json"
  val listJobsUrl = s"$baseUrl/listjobs.json"
  val cancelJobUrl = s"$baseUrl/cancel.json"
  val scheduleJobUrl = s"$baseUrl/schedule.json"
}

/**
 * Scrapyd api method for deploying new project with name
 * (in this case we use uuid of project stored in DB as name)
 * and egg file, containing project structure (crawlers).
 *
 * @param projectId project
 * @param eggPath path to egg archive
 * @return response from scrapyd server
 */
def addVersion(projectId: Long, eggPath: Path): Future[
   JsResult[AddVersionResponse]] = {

  ws
    .url(URL.addVersionUrl)
    .addQueryStringParameters(
      "project" -> s"${projectId}project",
      "version" -> projectId.toString)
    .withBody(FileIO.fromPath(eggPath))
    .execute("POST")
    .map(_.json.validate[AddVersionResponse])
}


/**
 * Scrapyd api method for getting all of project's spiders.
 * @param projectId project id
 * @param version optional version of the project
 * @return response from scrapyd server
 */
def listSpiders(projectId: Long, version: Option[String] =
   None): Future[JsResult[ListSpidersResponse]] = {

  ws
    .url(URL.listSpidersUrl)
    .addQueryStringParameters(
      "project" -> s"${projectId}project") // will add
         version parameter someday
    .execute("GET")
    .map(_.json.validate[ListSpidersResponse])
```

```
}


/**
 * Scrapyd api method for the full deletion of the project
 * @param projectId project id
 * @return response from scrapyd server
 */
def delProject(projectId: Long): Future[JsResult[
   GeneralResponse]] = {

  ws
    .url(URL.delProjectUrl)
    .addQueryStringParameters("project" -> s"${projectId}
       project")
    .execute("POST")
    .map(_.json.validate[GeneralResponse])
}

/**
 * Scrapyd api method for listing all of the existing jobs by
     their status.
 * @param projects project
 * @return response from scrapyd server
 */
def listJobs(projects: Seq[Long]): Future[Seq[(Long, JsResult
   [ListJobsResponse])]] = {

  Future.traverse(projects) ( p =>
    ws
      .url(URL.listJobsUrl)
      .addQueryStringParameters("project" -> s"${p}project")
      .get()
      .map(el => (p, el.json.validate[ListJobsResponse]))
  )
}


/**
 * Change status to disabled.
 *
 * @note Can be applied for running and pending jobs only.
 *       We repeat request for 5 times because of https://
    github.com/scrapy/scrapyd/issues/356.
 * @param projectId project for which job request is
    performed
 * @param jobId job to cancel. Important to send jobId in
    UUID format.
 * @return First response (out of 5)
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
  */
def cancelJob(projectId: Long, jobId: UUID): Future[JsResult[
    CancelJobResponse]] = {

  Future
    .traverse(1 to 5 toList)( _ =>
      ws
        .url(URL.cancelJobUrl)
        .addQueryStringParameters(
          "project" -> s"${projectId}project",
          "job" -> jobId.toString
        )
        .execute("POST")
        .map(_.json.validate[CancelJobResponse])
    )
    .flatMap(lst => Future(lst.head))
}

/**
 *  Scheduler for a one time job. You can specify your own
    jobId.
 * @param projectId project
 * @param spiderName spider
 * @param jobId job
 * @param version optional version number
 * @param setting setting like DOWNLOAD_DELAY
 * @param args arguments to be run
 * @return response from scrapyd server
 */
def scheduleJob(projectId: Long,
                spiderName: String,
                jobId: UUID,
                version: Option[String] = None,
                setting: Seq[String] = Seq.empty,
                args: Map[String, String] = Map.empty):
                   Future[JsResult[ScheduleResponse]] = {

  val settings = setting.map(s => ("setting",s))

  ws
    .url(URL.scheduleJobUrl)
    .addQueryStringParameters(
      "project" -> s"${projectId}project",
      "spider" -> spiderName,
      "jobid" -> jobId.toString,
      "_version" -> version.getOrElse(projectId.toString).
         toString
    )
    .addQueryStringParameters(args.toSeq:_*)
```

```
      .addQueryStringParameters(settings: _*)
      .execute("POST")
      .map(_.json.validate[ScheduleResponse])
  }
}
```

1.51 models/services/SecurityService.scala

```
package models.services

import java.util.UUID

import javax.inject.Inject
import models.common.PGProfile
import models.common.enums.MembershipAccessRight._
import models.common.enums.RunningType._
import models.tables.{Crawler, JobExecution, JobInstance,
   Membership}
import models.tables.JobInstance.dbJobInstances
import models.tables.Crawler.dbCrawlers
import models.tables.Membership.dbMembership
import models.tables.JobExecution.dbJobExecutions
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}

import scala.concurrent.{ExecutionContext, Future}

object SecurityService {
  val UserAccessMessage = "user␣has␣no␣specified␣access␣to␣the␣
     project"
  val CrawlerMessage = "CrawlerDoesntCorrespondToProject"
  val JobInstanceToProjectMessage = "
     JobInstanceDoesntCorrespondToProject"
  val JobExecutionToProjectMessage = "
     JobExecutionDoesntCorrespondToProject"
}

/**
 * Service to provide methods for security checkings.
 * @param dbConfigProvider for db
 * @param ex ExecutionContext
 */
class SecurityService @Inject()(protected val dbConfigProvider:
    DatabaseConfigProvider)(implicit ex: ExecutionContext)
   extends HasDatabaseConfigProvider[PGProfile] {

  import models.common.PGProfile.api._

  // MARK: - Public
```

```
/**
 * Checks if user has access to specified project
 * with specified access rights.
 *
 * @param projectId project
 * @param userId user
 * @param userAccess user access rights
 * @return Option of Membership
 */
def checkUserPermission(projectId: Long,
                        userId: UUID,
                        userAccess: MembershipAccessRight =
                           Readonly): Future[Option[
                           Membership]] = {

  val userAction =
    hasPermissionToAccessProject(projectId, userId,
       userAccess)
    .flatMap(res => checkResultAndPerformNextAction(
       SecurityService.UserAccessMessage, DBIO.successful(res
       ))(res))

  db.run(userAction.transactionally)
}


/**
 * Checks if project and crawler corresponds. Inside checks
    for user permission.
 * @param userId user
 * @param projectId project
 * @param crawlerId crawler
 * @param userAccess user permission
 * @return if everything succeeded
 */
def checkUserAndCrawler(userId: UUID,
                        projectId: Long,
                        crawlerId: Long,
                        userAccess: MembershipAccessRight =
                           Readonly): Future[Option[Crawler]]
                            = {

  val crawlerAction =
    hasPermissionToAccessProject(projectId, userId,
       userAccess)
    .flatMap(checkResultAndPerformNextAction(SecurityService.
       UserAccessMessage, crawlerCorrespondsToProject(
       projectId, crawlerId)))
    .flatMap(res => checkResultAndPerformNextAction(
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
        SecurityService.CrawlerMessage, DBIO.successful(res))(
        res))

    db.run(crawlerAction.transactionally)
  }

  /**
   * Checks if project and periodic job corresponds. Inside
     checks for user permission.
   * @param userId user id
   * @param projectId project
   * @param jobId periodic job
   * @param userAccess permission
   * @return weather or not user can get access to periodic job
     .
   */
  def checkUserAndPeriodicJob(userId: UUID,
                              projectId: Long,
                              jobId: Long,
                              userAccess: MembershipAccessRight
                                  = Readonly): Future[Option[
                                 JobInstance]] = {

    val periodicJobAction =
      hasPermissionToAccessProject(projectId, userId,
         userAccess)
      .flatMap(checkResultAndPerformNextAction(SecurityService.
         UserAccessMessage, periodicJobCorrespondsToProject(
         projectId, jobId)))
      .flatMap(res => checkResultAndPerformNextAction(
         SecurityService.JobInstanceToProjectMessage, DBIO.
         successful(res))(res))

    db.run(periodicJobAction.transactionally)
  }

  /**
   * For periodic job update
   * @param userId user
   * @param projectId project
   * @param jobId job
   * @param crawlerId crawler
   * @param userAccess permission
   * @return if everything is ok or not
   */
  def checkUserCrawlerAndPeriodicJob(userId: UUID,
                                     projectId: Long,
                                     jobId: Long,
                                     crawlerId: Long,
```

```
                                        userAccess:
                                           MembershipAccessRight =
                                            Readonly): Future[
                                           Option[JobInstance]] =
                                           {

    val periodicJobAction =
      hasPermissionToAccessProject(projectId, userId,
         userAccess)
        .flatMap(checkResultAndPerformNextAction(
          SecurityService.UserAccessMessage,
          crawlerCorrespondsToProject(projectId, crawlerId)))
        .flatMap(checkResultAndPerformNextAction(
          SecurityService.CrawlerMessage,
          periodicJobCorrespondsToProject(projectId, jobId)))
        .flatMap(res => checkResultAndPerformNextAction(
          SecurityService.JobInstanceToProjectMessage, DBIO.
          successful(res))(res))

    db.run(periodicJobAction.transactionally)
  }

  /**
   * Checks if project and onetime job corresponds. Inside
      checks for user permission.
   * @param userId user
   * @param projectId project
   * @param jobId onetime job
   * @param jobScrapydId onetime job id in scrapyd
   * @param userAccess permission to check
   * @return weather or not user can get access to onetime job.
   */
  def checkUserProjectAndJob(userId: UUID,
                             projectId: Long,
                             jobId: Long,
                             jobScrapydId: UUID,
                             userAccess: MembershipAccessRight
                                = Readonly): Future[Option[
                                JobExecution]] = {

    val jobExecutionAction =
      hasPermissionToAccessProject(projectId, userId,
         userAccess)
      .flatMap(checkResultAndPerformNextAction(SecurityService.
         UserAccessMessage, jobExecCorrespondsToProject(
         projectId, jobId, jobScrapydId)))
      .flatMap(res => checkResultAndPerformNextAction(
         SecurityService.JobExecutionToProjectMessage, DBIO.
         successful(res))(res))
```

```
  db.run(jobExecutionAction)
}


// MARK: - Private

/**
 * Checks if user has at least readonly access to the project
    .
 * You can specify permission value to check for other type
    of membership permission.
 * Default is 'Readonly'
 *
 * @param projectId project
 * @param userId user
 * @param permissionSpecified permission
 * @return optional if user has access to project or not
 */
private def hasPermissionToAccessProject(projectId: Long,
                                         userId: UUID,
                                         permissionSpecified:
                                            MembershipAccessRight =
                                            Readonly): DBIO[Option[
                                            Membership]] = {
  val membership = dbMembership
    .filter { membership =>
      membership.projectId === projectId &&
        membership.userId === userId &&
        membership.accessRight >= permissionSpecified  }
    .result
    .headOption

  membership
}

/**
 * Method for flatMap to pass next action if current
 * has not failed and error message if it has.
 * @param errForPrevious error for previous failed action
 * @param nextAction next action to be performed if previous
    succeed.
 * @tparam T  type of DBIO previous action
 * @tparam V type of DBIO next action
 * @return optional result
 */
private def checkResultAndPerformNextAction[T,V](
   errForPrevious: String,
                                                  nextAction:
```

```
                                                        DBIO[
                                                        Option[V
                                                        ]]):
                                                        Option[T]
                                                         => DBIO[
                                                        Option[V
                                                        ]] = {
  case Some(_) => // eliminate result, we need to only unwrap
      it
    nextAction
  case None =>
    DBIO.failed(new RuntimeException(errForPrevious))
}

/**
 * Check correspondence between crawler and project.
 * @param projectId project
 * @param crawlerId crawler
 * @return weather or not crawler is in specified project.
 */
private def crawlerCorrespondsToProject(projectId: Long,
   crawlerId: Long): DBIO[Option[Crawler]] = {
  dbCrawlers
    .filter { crawler =>
      crawler.id === crawlerId && crawler.projectId ===
         projectId }
    .result
    .headOption
}

/**
 * Check correspondence between periodic job and project.
 * @param projectId project
 * @param pJobId periodic job
 * @return weather or not projects has specified periodic job
    .
 */
private def periodicJobCorrespondsToProject(projectId: Long,
   pJobId: Long): DBIO[Option[JobInstance]] = {
  dbJobInstances
    .filter { jInstance =>
      jInstance.id === pJobId &&
        jInstance.projectId === projectId &&
        jInstance.runType === Periodic }
    .result
    .headOption
}

/**
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
    * Check correspondence between project id and job execution
       id.
    *
    * @param projectId project
    * @param jobId job execution
    * @param jobScrapydId job execution scrapyd id
    * @return Optional if correspondence was found.
    */
  private def jobExecCorrespondsToProject(projectId: Long,
     jobId: Long, jobScrapydId: UUID): DBIO[Option[JobExecution
     ]] = {
    val q = for {
      (jExec, _) <- dbJobExecutions
        .filter { jExecution => jExecution.id === jobId &&
           jExecution.scrapydId === jobScrapydId }
        .join(dbJobInstances).on(_.jobInstanceId === _.id)
    } yield (jExec)

    q.result.headOption
  }

}
```

1.52 models/services/UpdaterService.scala

```
package models.services

import java.util.UUID

import javax.inject.Inject
import models.common.enums.JobExecutionStatus.
   JobExecutionStatus
import models.common.PGProfile
import models.common.enums.JobExecutionStatus
import models.daos.JobDAO
import models.tables.JobExecution.dbJobExecutions
import models.scrapyd\_response.{JobScrapyd, ListJobsResponse}
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}
import play.api.libs.json.{JsError, JsResult, JsSuccess}

import scala.concurrent.{ExecutionContext, Future}

/**
 * Complex class for performing data syncing scrapyd/db.
 * @param dbConfigProvider for db
 * @param scrapydService for scrapyd api
 * @param ec ExecutionContext
 */
class UpdaterService @Inject() (protected val dbConfigProvider:
```

```
  DatabaseConfigProvider,
                                scrapydService: ScrapydService)
                                  (implicit ec:
                                  ExecutionContext) extends
                                  HasDatabaseConfigProvider[
                                  PGProfile] {

import models.common.PGProfile.api._


/**
 * This function is used for reaching to scrapyd server and
    fetching latest data.
 * After that, it updates our DB and returns with relevant
    info.
 *
 * @note It has to be used by PUT, DELETE and GET requests to
     provide user with the latest info.
 *       In case of restarting scrapyd server - we need to
    return data from DB.
 *
 * @param pIds project ids
 * @param mapping func to map result
 * @tparam T sequence type
 * @return sequence of data
 */
def fetchAndUpdateData[T](pIds: Seq[Long], mapping: (Long) =>
    Future[Seq[T]] = ((pId: Long) => Future(Seq.empty[T]))):
   Future[Seq[Future[Seq[T]]]] = {

  /** Filter jobs and update content in db by mapping new
     status and (startTime, endTime).
   * Note:  https://youtrack.jetbrains.com/issue/SCL-16399 */
  def mapToStatus(status: JobExecutionStatus, jobs: List[
     JobScrapyd]): DBIOAction[List[Int], NoStream, Effect.
     Write] =
    DBIO.sequence(jobs.map(job => {
      dbJobExecutions
        .filter(_.scrapydId === job.id)
        .map(updJob => (updJob.status, updJob.startTime,
           updJob.endTime))
        .update((status, job.start_time, job.end_time))
    }))

  /** Update statuses of jobs in db */
  def mapFromScrapyd(response: ListJobsResponse, projectId:
     Long): Future[Unit] = {
    db.run((for {
      _ <- mapToStatus(JobExecutionStatus.Pending, response.
```

```
            pending)
          _ <- mapToStatus(JobExecutionStatus.Running, response.
             running)
          _ <- mapToStatus(JobExecutionStatus.Finished, response.
             finished)
        } yield ()).transactionally)
    }

    /** Mapper for scrapyd JsResult, passes successful result
       further to mapper */
    val scrapydResponseMapper: ((Long, JsResult[
       ListJobsResponse])) => Future[Unit] = { case (pId,
       response) =>
      response match {
        case JsSuccess(value, _) => mapFromScrapyd(value, pId)
        case JsError(_) => Future.successful(())
      }
    }

    scrapydService
      .listJobs(pIds)
      .map(result => result.map(scrapydResponseMapper))
      .map(e => pIds.map(mapping))
  }

}
```

1.53 models/services/UserService.scala

```
package models.services

import java.util.UUID

import com.mohiva.play.silhouette.api.LoginInfo
import com.mohiva.play.silhouette.api.repositories.
   AuthInfoRepository
import com.mohiva.play.silhouette.api.services.IdentityService
import com.mohiva.play.silhouette.api.util.{
   PasswordHasherRegistry, PasswordInfo}
import com.mohiva.play.silhouette.impl.providers.
   CredentialsProvider
import forms.SignUp
import javax.inject.Inject
import models.daos.MembershipDAO
import models.tables.User.dbUsers
import models.tables.User
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}
import slick.jdbc.JdbcProfile
```

```
import scala.concurrent.{ExecutionContext, Future}
import scala.util.{Failure, Success}

/**
 * User service for authorization.
 * @param dbConfigProvider for db
 * @param passwordHasherRegistry for password
 * @param authInfoRepository authentication
 * @param membershipDAO access to membership db
 * @param ec ExecutionContext
 */
class UserService @Inject() (protected val dbConfigProvider:
   DatabaseConfigProvider,
                             passwordHasherRegistry:
                                PasswordHasherRegistry,
                             authInfoRepository:
                                AuthInfoRepository,
                             membershipDAO: MembershipDAO)(
                                implicit ec: ExecutionContext)
  extends IdentityService[User]
  with HasDatabaseConfigProvider[JdbcProfile]  {

  import profile.api._

  /**
   * Get existing user.
   * @param loginInfo email and password.
   * @return Some if user was found in db. None otherwise.
   */
  override def retrieve(loginInfo: LoginInfo): Future[Option[
     User]] =
    db.run(
      dbUsers
        .filter(user => user.providerKey === loginInfo.
           providerKey && user.providerId === loginInfo.
           providerID)
        .result
        .headOption)

  /**
   * Create user and add to db.
   * @param form sign up form. Data.
   * @return Created user.
   */
  def create(form: SignUp): Future[User] = {
    val newUser = User(
      id = UUID.randomUUID(),
      name = form.name,
      login = form.login,
```

```
      email = form.email,
      providerID = CredentialsProvider.ID,
      providerKey = form.email
    )

    db.run {
      dbUsers += newUser
    } andThen {
      case Failure(e: Throwable) =>
        None

      case Success(id: Int) =>
        val loginInfo: LoginInfo   = LoginInfo(
          CredentialsProvider.ID, form.email)
        val authInfo: PasswordInfo = passwordHasherRegistry.
          current.hash(form.password)
        authInfoRepository.add(loginInfo, authInfo)
    } map { _ => newUser }
  }

}
```

1.54 models/tables/Crawler.scala

```
package models.tables

import java.util.UUID

import io.swagger.annotations.ApiModel
import models.common.PGProfile.api._
import play.api.libs.json.JsValue
import models.tables.Project.dbProjects

@ApiModel(description = "Crawler␣object")
case class Crawler(id: Long = 0,
                   projectId: Long,
                   name: String,
                   settings: Option[JsValue] = None,
                   args: Option[JsValue] = None)

object Crawler {
  class CrawlerTable(tag: Tag) extends Table[Crawler](tag, "
     crawler") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def name = column[String]("name")
    def projectId = column[Long]("project_id")
    def settings = column[Option[JsValue]]("settings")
    def args = column[Option[JsValue]]("args")

    def projectFK = foreignKey("project_fk", projectId,
```

```
      dbProjects)(_.id)

    def * = (id, projectId, name, settings, args) <> ((Crawler.
      apply _)tupled, Crawler.unapply)
  }

  val dbCrawlers = TableQuery[CrawlerTable]
}
```

1.55 models/tables/JobExecution.scala

```
package models.tables

import java.time.Instant
import java.util.UUID

import models.tables.JobInstance.dbJobInstances
import models.common.PGProfile.api._
import models.common.enums.JobExecutionStatus
import models.common.enums.JobExecutionStatus.
   JobExecutionStatus
import play.api.libs.json.JsValue

case class JobExecution(id: Long = 0,
                        scrapydId: UUID = UUID.randomUUID(),
                        jobInstanceId: Long,
                        executionSettings: Option[JsValue] =
                           None,
                        executionArgs: Option[JsValue] = None,
                        createTime: Instant,
                        startTime: Option[Instant] = None,
                        endTime: Option[Instant] = None,
                        status: JobExecutionStatus =
                           JobExecutionStatus.Pending)

object JobExecution {
  class JobExecutionTable(tag: Tag) extends Table[JobExecution
     ](tag, "job_execution") {
    def id = column[Long]("id", O.AutoInc, O.PrimaryKey)
    def scrapydId = column[UUID]("scrapyd_id", O.Unique)
    def jobInstanceId = column[Long]("job_instance_id")
    def executionSettings = column[Option[JsValue]](""
       execution_settings")
    def executionArgs = column[Option[JsValue]]("execution_args
       ")
    def createTime = column[Instant]("create_time")
    def startTime = column[Option[Instant]]("start_time")
    def endTime = column[Option[Instant]]("end_time")
    def status = column[JobExecutionStatus]("status")
```

```
    def jobInstanceFK = foreignKey("job_instance_id_fk",
       jobInstanceId, dbJobInstances)(_.id)

    def * = (id, scrapydId, jobInstanceId, executionSettings,
       executionArgs, createTime, startTime, endTime, status)
       <> ((JobExecution.apply _).tupled, JobExecution.unapply)
  }

  val dbJobExecutions = TableQuery[JobExecutionTable]

}
```

1.56 models/tables/JobInstance.scala

```
package models.tables


import models.common.PGProfile.api._
import models.common.enums.{JobPriority, RunningStatus}
import models.common.enums.JobPriority.JobPriority
import models.common.enums.RunningStatus.RunningStatus
import models.common.enums.RunningType.RunningType
import play.api.libs.json.{JsValue, Json, OFormat}
import models.tables.Project.dbProjects
import models.tables.Crawler.dbCrawlers

case class JobInstance(id: Long = 0,
                       projectId: Long,
                       title: Option[String] = None,
                       description: Option[String] = None,
                       settings: Option[JsValue] = None,
                       args: Option[JsValue] = None,
                       priority: JobPriority = JobPriority.
                          Normal,
                       status: RunningStatus = RunningStatus.
                          Enabled,
                       spider: Long,
                       runType: RunningType,
                       cron: Option[String] = None)

object JobInstance {
  class JobInstanceTable(tag: Tag) extends Table[JobInstance](
     tag, "job_instance") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def projectId = column[Long]("project_id")
    def settings = column[Option[JsValue]]("settings")
    def args = column[Option[JsValue]]("args")
    def priority = column[JobPriority]("priority")
    def status = column[RunningStatus]("status")
    def spiderId = column[Long]("spider_id")
```

```
    def runType = column[RunningType]("run_type")

    def cron = column[Option[String]]("cron")

    def title = column[Option[String]]("title")
    def description = column[Option[String]]("description")

    def projectFK = foreignKey("project_fk", projectId,
       dbProjects)(_.id)
    def spiderFK = foreignKey("spider_fk", spiderId, dbCrawlers
       )(_.id)

    def * = (id, projectId, title, description, settings, args,
      priority, status, spiderId, runType, cron) <> ((
         JobInstance.apply _)tupled, JobInstance.unapply)
  }

  implicit val jobInstanceFormat: OFormat[JobInstance] = Json.
     format[JobInstance]

  val dbJobInstances = TableQuery[JobInstanceTable]
}
```

1.57 models/tables/Membership.scala

```
package models.tables

import java.util.UUID

import models.common.PGProfile.api._
import models.common.enums.MembershipAccessRight.
   MembershipAccessRight

case class Membership(userId: UUID,
                      projectId: Long,
                      accessRight: MembershipAccessRight)

object Membership {

  class MembershipTable(tag: Tag) extends Table[Membership](tag
     , "membership") {
    def userId = column[UUID]("user_id")
    def projectId = column[Long]("project_id")
    def accessRight = column[MembershipAccessRight]("
       access_right")

    def pk = primaryKey("membership_pk", (userId, projectId))
    def userFK = foreignKey("membership_user_fk", userId, User.
       dbUsers)(_.id)
    def projectFK = foreignKey("membership_project_fk",
```

```
      projectId, Project.dbProjects)(_.id, onUpdate=
      ForeignKeyAction.Cascade, onDelete=ForeignKeyAction.
      Cascade)

    def * = (userId, projectId, accessRight) <> ((Membership.
      apply _)tupled, Membership.unapply)
  }

  val dbMembership = TableQuery[MembershipTable]
}
```

1.58 models/tables/Password.scala

```
package models.tables

import models.common.PGProfile.api._

case class Password(key: String,
                    hasher: String,
                    hash: String,
                    salt: Option[String])

object Password {
  class PasswordTable(tag: Tag) extends Table[Password](tag, "
     password") {
    def key = column[String]("provider_key", O.PrimaryKey)
    def hasher = column[String]("hasher")
    def hash   = column[String]("hash")
    def salt = column[Option[String]]("salt")

    def * = (key, hasher, hash, salt) <> ((Password.apply _).
      tupled, Password.unapply)
  }

  val dbPasswords = TableQuery[PasswordTable]
}
```

1.59 models/tables/Project.scala

```
package models.tables

import java.time.Instant
import java.util.UUID

import io.swagger.annotations.ApiModel
import models.common.PGProfile.api._
import play.api.libs.json.JsValue
import models.tables.User.dbUsers

@ApiModel(description = "Project␣object")
```

```
case class Project(id: Long = 0,
                   name: String,
                   description: Option[String],
                   spidersSettings: Option[JsValue] = None,
                   args: Option[JsValue] = None,
                   eggfile: Option[Array[Byte]] = None,
                   ownerId: UUID,
                   createdAt: Instant,
                   changedBy: UUID,
                   changedAt: Instant)

object Project {

  class ProjectTable(tag: Tag) extends Table[Project](tag, "
     project") {
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def name = column[String]("name")
    def description = column[Option[String]]("description")
    def settings = column[Option[JsValue]]("settings")
    def args = column[Option[JsValue]]("args")
    def eggfile = column[Option[Array[Byte]]]("eggfile")
    def owner = column[UUID]("owner")
    def createdAt = column[Instant]("created_at")
    def changedBy = column[UUID]("changed_by")
    def changedAt = column[Instant]("changed_at")

    def ownerFK = foreignKey("owner_fk", owner, dbUsers)(_.id)
    def createdByFK = foreignKey("changed_by_FK", changedBy,
       dbUsers)(_.id)

    def * = (id, name, description, settings, args, eggfile,
       owner, createdAt, changedBy, changedAt) <> ((Project.
       apply _)tupled, Project.unapply)
  }

  val dbProjects = TableQuery[ProjectTable]
}
```

1.60 models/tables/User.scala

```
package models.tables

import java.util.UUID

import com.mohiva.play.silhouette.api.Identity
import models.common.PGProfile.api._

case class User(id: UUID,
                name: String,
                login: String,
```

```
                email: String,
                providerID: String,
                providerKey: String) extends Identity

object User {
  class UsersTable(tag: Tag) extends Table[User] (tag, "user")
     {
    def id = column[UUID]("id", O.PrimaryKey)
    def name = column[String]("name")
    def login = column[String]("login")
    def email = column[String]("email")
    def providerId = column[String]("provider_id")
    def providerKey = column[String]("provider_key")

    def * = (id, name, login, email, providerId, providerKey)
       <> ((User.apply _).tupled, User.unapply)
  }

  val dbUsers = TableQuery[UsersTable]
}
```

1.61 modules/ActorModule.scala

```
package modules

import com.google.inject.AbstractModule
import models.actors.JobSchedulerActor
import models.services.JobService
import play.api.libs.concurrent.AkkaGuiceSupport

class ActorModule extends AbstractModule with AkkaGuiceSupport
   {

  override def configure = {
    bind(classOf[JobService]).asEagerSingleton()
    bindActor[JobSchedulerActor]("job-actor")
  }
}
```

1.62 modules/SilhouetteModule.scala

```
package modules

import com.google.inject.name.Named
import com.google.inject.{AbstractModule, Provides}
import com.mohiva.play.silhouette.api.{Environment, EventBus,
   Silhouette, SilhouetteProvider}
import com.mohiva.play.silhouette.api.actions.{
   DefaultSecuredErrorHandler, DefaultUnsecuredErrorHandler,
   SecuredErrorHandler, UnsecuredErrorHandler}
```

```
import com.mohiva.play.silhouette.api.crypto.{Crypter,
   CrypterAuthenticatorEncoder, Signer}
import com.mohiva.play.silhouette.api.repositories.
   AuthInfoRepository
import com.mohiva.play.silhouette.api.services.
   AuthenticatorService
import com.mohiva.play.silhouette.api.util.{CacheLayer, Clock,
   FingerprintGenerator, IDGenerator, PasswordHasherRegistry,
   PasswordInfo}
import com.mohiva.play.silhouette.crypto.{JcaCrypter,
   JcaCrypterSettings, JcaSigner, JcaSignerSettings}
import com.mohiva.play.silhouette.impl.authenticators.{
   CookieAuthenticator, CookieAuthenticatorService,
   CookieAuthenticatorSettings}
import com.mohiva.play.silhouette.impl.providers.
   CredentialsProvider
import com.mohiva.play.silhouette.impl.util.{
   DefaultFingerprintGenerator, PlayCacheLayer,
   SecureRandomIDGenerator}
import com.mohiva.play.silhouette.password.{
   BCryptPasswordHasher, BCryptSha256PasswordHasher}
import com.mohiva.play.silhouette.persistence.daos.
   DelegableAuthInfoDAO
import com.mohiva.play.silhouette.persistence.repositories.
   DelegableAuthInfoRepository
import models.daos.PasswordDAO
import net.codingwell.scalaguice.ScalaModule
import play.api.Configuration
import play.api.mvc.CookieHeaderEncoding
import models.services.UserService
import utils.DefaultEnv

import scala.concurrent.ExecutionContext
import scala.concurrent.duration.FiniteDuration

class SilhouetteModule extends AbstractModule with ScalaModule
   {

  override def configure(): Unit = {
    bind[Silhouette[DefaultEnv]].to[SilhouetteProvider[
       DefaultEnv]]
    bind[UnsecuredErrorHandler].to[DefaultUnsecuredErrorHandler
       ]
    bind[SecuredErrorHandler].to[DefaultSecuredErrorHandler]
    bind[DelegableAuthInfoDAO[PasswordInfo]].to[PasswordDAO]
    bind[EventBus].toInstance(EventBus())
    bind[Clock].toInstance(Clock())
  }
```

```
@Provides
def provideEnvironment(userService: UserService,
                       authenticatorService:
                          AuthenticatorService[
                          CookieAuthenticator],
                       eventBus: EventBus)(implicit ec:
                          ExecutionContext): Environment[
                          DefaultEnv] =
  Environment[DefaultEnv](
    userService,
    authenticatorService,
    Seq(),
    eventBus
  )

@Provides
def provideFingerprintGenerator(): FingerprintGenerator =
  new DefaultFingerprintGenerator(false)

@Provides
def providesCookieAuthenticatorSettings(configuration:
   Configuration): CookieAuthenticatorSettings =
  CookieAuthenticatorSettings(
    cookieName = configuration.get[String]("silhouette.
       authenticator.cookieName"),
    cookiePath = configuration.get[String]("silhouette.
       authenticator.cookiePath"),
    cookieDomain = None,
    secureCookie = configuration.get[Boolean]("silhouette.
       authenticator.secureCookie"),
    httpOnlyCookie = configuration.get[Boolean]("silhouette.
       authenticator.httpOnlyCookie"),
    useFingerprinting = configuration.get[Boolean]("
       silhouette.authenticator.useFingerprinting"),
    cookieMaxAge = None,
    authenticatorIdleTimeout = None,
    authenticatorExpiry = configuration.get[FiniteDuration]("
       silhouette.authenticator.authenticatorExpiry")
  )

@Provides
def provideAuthenticatorService(
                                  @Named("authenticator-signer
                                     ") signer: Signer,
                                  @Named("authenticator-
                                     crypter") crypter:
                                     Crypter,
                                  settings:
                                     CookieAuthenticatorSettings
```

```
                                        ,
                                    cookieHeaderEncoding:
                                       CookieHeaderEncoding,
                                    fingerprintGenerator:
                                       FingerprintGenerator,
                                    idGenerator: IDGenerator,
                                    configuration: Configuration
                                       ,
                                    clock: Clock)(implicit ec:
                                       ExecutionContext):
                                       AuthenticatorService[
                                       CookieAuthenticator] = {

    val authenticatorEncoder = new CrypterAuthenticatorEncoder(
       crypter)

    new CookieAuthenticatorService(
      settings,
      None,
      signer,
      cookieHeaderEncoding,
      authenticatorEncoder,
      fingerprintGenerator,
      idGenerator,
      clock)
  }

  @Provides
  def provideSecureRandomGenerator()(implicit ec:
     ExecutionContext): IDGenerator =
    new SecureRandomIDGenerator()

  @Provides
  def provideAuthInfoRepository(passwordDAO:
     DelegableAuthInfoDAO[PasswordInfo])(
    implicit ec: ExecutionContext): AuthInfoRepository =
    new DelegableAuthInfoRepository(passwordDAO)

  @Provides
  def providePasswordHasherRegistry(): PasswordHasherRegistry =
    PasswordHasherRegistry(new BCryptSha256PasswordHasher(),
       Seq(new BCryptPasswordHasher()))

  @Provides
  def provideCredentialsProvider( authInfoRepository:
     AuthInfoRepository,
                                  passwordHasherRegistry:
                                     PasswordHasherRegistry)(
                                     implicit ec:
```

```
                                        ExecutionContext):
                                        CredentialsProvider =
    new CredentialsProvider(authInfoRepository,
       passwordHasherRegistry)

  @Provides
  @Named("authenticator-signer")
  def provideAuthenticatorSigner(configuration: Configuration):
      Signer = {
    val config = JcaSignerSettings("SecretKey")

    new JcaSigner(config)
  }

  @Provides
  @Named("authenticator-crypter")
  def provideAuthenticatorCrypter(configuration: Configuration)
     : Crypter = {

    val config = JcaCrypterSettings("SecretKey")

    new JcaCrypter(config)
  }

}
```

1.63 utils/DefaultEnv.scala

```
package utils

import com.mohiva.play.silhouette.api.Env
import com.mohiva.play.silhouette.impl.authenticators.
   CookieAuthenticator
import models.tables.User

trait DefaultEnv extends Env {
  type I = User
  type A = CookieAuthenticator
}
```

1.64 views/index.scala.html

```
@()

@main("Welcome␣to␣Play") {
  <h1>Welcome to Play!</h1>
}
```

1.65 views/main.scala.html

```
@*
 * This template is called from the ‘index‘ template. This
    template
 * handles the rendering of the page header and body tags. It
    takes
 * two arguments, a ‘String‘ for the title of the page and an ‘
    Html‘
 * object to insert into the body of the page.
 *@
@(title: String)(content: Html)

<!DOCTYPE html>
<html lang="en">
    <head>
        @* Here’s␣where␣we␣render␣the␣page␣title␣‘String‘.␣*@
␣␣␣␣␣␣␣␣<title>@title</title>
␣␣␣␣␣␣␣␣<link␣rel="stylesheet"␣media="screen"␣href="@routes.
   Assets.versioned("stylesheets/main.css")">
␣␣␣␣␣␣␣␣<link␣rel="shortcut␣icon"␣type="image/png"␣href="@
   routes.Assets.versioned("images/favicon.png")">

␣␣␣␣</head>
␣␣␣␣<body>
␣␣␣␣␣␣␣␣@*␣And␣here’s where we render the ‘Html‘ object
   containing
         * the page content. *@
        @content

      <script src="@routes.Assets.versioned("javascripts/main.
         js")" type="text/javascript"></script>
    </body>
</html>
```

2 Test

2.1 controllers/ApplicationSpec.scala

```
package controllers


import utils.{AuthSpecification,DatabaseCleaner}
import play.api.test._
import com.mohiva.play.silhouette.test._
import play.api.test.Helpers._



class ApplicationSpec extends AuthSpecification
                      with DatabaseCleaner {
```

```
  import utils.data.UserData._

  private val controller = app.injector.instanceOf[
     ApplicationController]

  "ApplicationController␣Logout␣GET" should {

    "should␣be␣unauthorized␣error" in {
      val result = controller.logout.apply(FakeRequest(method =
          "GET", "/"))
      status(result) mustBe UNAUTHORIZED
    }

    "redicrect␣if␣user␣was␣found" in {
      val request = FakeRequest().withAuthenticator(loginInfo)
      val result = controller.logout(request)

      status(result) mustBe SEE_OTHER
      redirectLocation(result) mustBe Some("/api/helloworld")
    }
  }
}
```

2.2 controllers/AuthorizationSpec.scala

```
package controllers

import akka.stream.Materializer
import com.mohiva.play.silhouette.api.LoginInfo
import forms.SignIn
import play.api.libs.json.{JsString, Json}
import play.api.test.FakeRequest
import play.api.test.Helpers._
import utils.{AuthSpecification, DatabaseCleaner}

class AuthorizationSpec extends AuthSpecification
                        with DatabaseCleaner {

  import utils.data.UserData._

  private val controllerSignUp = app.injector.instanceOf[
     SignUpController]
  private val controllerSignIn = app.injector.instanceOf[
     SignInController]
  implicit lazy val materializer: Materializer = app.
     materializer

  "Authorize␣person" should {
```

```
    "signUp: OK" in {

      val result = controllerSignUp.signUp()(FakeRequest().
         withBody(signUpForm))

      status(result) mustBe OK
      contentAsJson(result) mustEqual Json.toJson(Map("user" ->
          LoginInfo("credentials", authorizationEmail)))

      val resultSignIn = controllerSignIn.signIn()(FakeRequest
         ().withBody(credentials))
      status(resultSignIn) mustBe OK
    }

    "signUp: userAlreadyExists" in {

      val result = controllerSignUp.signUp()(FakeRequest().
         withBody(signUpForm.copy(email = email)))

      status(result) mustBe CONFLICT
      contentAsJson(result) mustBe JsString(controllerSignUp.
         UserAlreadyExistsMessage)
    }

    "signUp: invalid email format" in {
      val invalidCredentials = signUpForm.copy(email = "
         wrongformat@ j j j")
      val result = call(controllerSignUp.signUp(), FakeRequest(
         POST, "/api/auth/signup").withJsonBody(Json.toJson(
         invalidCredentials)))
      status(result) mustBe FORBIDDEN
    }

    "signIn: wrong credentials" in {

      val wrongCredentials = Json.toJson(SignIn("ddd@kkk.d", "
         fff"))
      val result = call(controllerSignIn.signIn(), FakeRequest(
         POST, "/api/auth/signin").withJsonBody(
         wrongCredentials))

      status(result) mustBe FORBIDDEN
    }
  }
}
```

2.3 controllers/CrawlerSpec.scala

```
package controllers
```

```
import models.tables.Crawler
import play.api.libs.json.{Json, OFormat}
import play.api.test.FakeRequest
import play.api.test.Helpers.{contentAsJson, status}
import utils.{AuthSpecification, DatabaseCleaner, TestHelper}
import play.api.test.Helpers._
import utils.data.UserData._
import com.mohiva.play.silhouette.test._
import forms.SpiderChangeForm
import models.services.SecurityService
import utils.data.ProjectTestData

class CrawlerSpec extends AuthSpecification
                  with DatabaseCleaner {

  implicit val crawlerFormat: OFormat[Crawler] = Json.format[
     Crawler]

  private val projectsController = app.injector.instanceOf[
     ProjectsController]
  private val crawlersController = app.injector.instanceOf[
     CrawlersController]

  private val readonlyProject = ProjectTestData.Access.readOnly
  private val readAndWriteProject = ProjectTestData.Access.
     readAndWrite
  private val noAccessProject = ProjectTestData.Access.noAccess

  private def getPutRequest: FakeRequest[SpiderChangeForm] = {

    val changeBody = SpiderChangeForm(settings = Map("
       DOWNLOAD_DELAY" -> "300"))
    FakeRequest().withAuthenticator(loginInfo).withBody(
       changeBody)
  }

  "GET␣crawlers:␣OK" in {

    val projectId = 15
    val deployResult = projectsController.deployProject(
       projectId)(TestHelper.requestWithMetadata("egg1.egg"))

    status(deployResult) mustBe OK
    contentAsJson(deployResult) mustBe Json.toJson(TestHelper.
       crawlers)

    val getResult = crawlersController.listSpiders(projectId)(
       FakeRequest().withAuthenticator(loginInfo))
```

```
    status(getResult) mustBe OK
    contentAsJson(deployResult) mustBe Json.toJson(TestHelper.
      crawlers)
  }

  "GET␣crawlers:␣no␣access" in {

    val getResult = crawlersController.listSpiders(
      noAccessProject)(FakeRequest().withAuthenticator(
      loginInfo))

    status(getResult) mustBe FORBIDDEN
  }

  "PUT␣crawlers" in {

    val putResult = crawlersController.updateSpider(
      readAndWriteProject, 2)(getPutRequest)

    status(putResult) mustBe OK
  }

  "PUT␣crawlers:␣ReadOnly␣access" in {

    val putResult = crawlersController.updateSpider(
      readonlyProject, 1)(getPutRequest)

    status(putResult) mustBe FORBIDDEN
    contentAsString(putResult) mustBe SecurityService.
      UserAccessMessage
  }

  "PUT␣crawlers:␣spider␣not␣found" in {

    val putResult = crawlersController.updateSpider(
      readAndWriteProject, 40)(getPutRequest)

    status(putResult) mustBe FORBIDDEN
    contentAsString(putResult) mustBe SecurityService.
      CrawlerMessage
  }
}
```

2.4 controllers/jobs/JobTestCase.scala

```
package controllers.jobs

import controllers.{JobsController, PeriodicJobsController,
   ProjectsController}
```

```
import models.tables.Crawler
import play.api.libs.json.{Json, OFormat}
import play.api.test.Helpers.{status, _}
import utils.{AuthSpecification, DatabaseCleaner, TestHelper}

class JobTestCase extends AuthSpecification
                  with DatabaseCleaner {

  // MARK: - Controllers

  val periodicJobsController: PeriodicJobsController = app.
     injector.instanceOf[PeriodicJobsController]
  val projectController: ProjectsController = app.injector.
     instanceOf[ProjectsController]
  val jobsController: JobsController = app.injector.instanceOf[
     JobsController]

  implicit val crawlerFormat: OFormat[Crawler] = Json.format[
     Crawler]

  def postEggFile(projectId: Long): Unit = {
    val deployResult = projectController.deployProject(
       projectId)(TestHelper.requestWithMetadata("egg1.egg"))

    status(deployResult) mustBe OK
  }

}
```

2.5 controllers/jobs/specs/JobExecutionSpec.scala

```
package controllers.jobs.specs

import java.util.UUID

import com.mohiva.play.silhouette.test._
import controllers.jobs.JobTestCase
import models.common.enums.JobExecutionStatus
import models.responses.SimpleJob
import models.services.SecurityService
import play.api.libs.json._
import play.api.test.FakeRequest
import play.api.test.Helpers._
import utils.data.UserData._
import utils.TestHelper
import utils.data.{JobTestData, ProjectTestData}

class JobExecutionSpec extends JobTestCase {

  private val projectId: Long = JobTestData.OnetimeJob.Owner.
```

```
    projectId
  private val jobId: Long = JobTestData.OnetimeJob.ids.length
  private val scheduledJobId: Long = JobTestData.OnetimeJob.ids
    .length + 1

  private val request = FakeRequest().withAuthenticator(
    loginInfo)

  private val readAndWriteProject = ProjectTestData.Access.
    readAndWrite
  private val readonlyProject = ProjectTestData.Access.readOnly

  private def scheduleJob(projectId: Long, scheduledJobId: Long
    ): UUID = {
    /** Post egg file with crawlers for project */
    postEggFile(projectId)

    /** Schedule job */
    val request = FakeRequest().withAuthenticator(loginInfo).
      withBody(TestHelper.onetimeJobForm)
    val result = jobsController.schedule(projectId)(request)

    val responseObject = contentAsJson(result).as[SimpleJob]

    responseObject.id mustBe scheduledJobId
    status(result) mustBe OK

    responseObject.scrapydId
  }

  "JobsController" should {

    "GET␣jobs" in {

      val limit = 10
      val result = jobsController.getJobsExecutions(limit,
        JobExecutionStatus.Finished)(request)

      status(result) mustBe OK
      contentAsJson(result) match {
        case JsArray(value) =>
          value must have length limit
          (value(0) \ "id").get must equal(JsNumber(20))
        case _ => assertTypeError("Error␣type")
      }

      val resultRunning = jobsController.getJobsExecutions(
        limit, JobExecutionStatus.Running)(request)
```

```
    status(resultRunning) mustBe OK
    contentAsJson(resultRunning) match {
      case JsArray(value) => value must have length 0
      case _ => assertTypeError("Error␣type")
    }
  }

  "GET␣jobs:␣pagination" in {

    val limit = 10
    val result = jobsController.getJobsExecutions(limit,
       JobExecutionStatus.Finished, Some(3))(request)

    status(result) mustBe OK
    status(result) mustBe OK
    contentAsJson(result) match {
      case JsArray(value) =>
        value must have length 2
        (value(0) \ "id").get must equal(JsNumber(2))
      case _ => assertTypeError("Error␣type")
    }
  }

  "POST␣schedule:␣ordinary" in {

    scheduleJob(projectId, scheduledJobId)

    /** Get 1 running/pending job execution */
    val getResult = jobsController.getJobsExecutions(10,
       JobExecutionStatus.Pending)(request)

    status(getResult) mustBe OK
    contentAsJson(getResult) match {
      case JsArray(value) =>
        value must have length 1
        (value(0) \ "id").get must equal(JsNumber(
           scheduledJobId))
        (value(0) \ "status").get must equal(JsString("
           Pending"))
      case _ => assertTypeError("Error␣type")
    }
  }


  "POST␣schedule:␣ReadOnly␣access" in {

    /** Schedule job with ReadOnly access */
    val request = FakeRequest().withAuthenticator(loginInfo).
       withBody(TestHelper.onetimeJobForm)
```

```
    val result = jobsController.schedule(readonlyProject)(
      request)

    status(result) mustBe FORBIDDEN
    contentAsString(result) mustBe SecurityService.
      UserAccessMessage
  }

  "POST␣schedule:␣ProjectId␣doesn't␣match␣CrawlerId" in {

    val request = FakeRequest().withAuthenticator(loginInfo).
      withBody(TestHelper.onetimeJobForm)
    val result = jobsController.schedule(readAndWriteProject)
      (request)

    status(result) mustBe FORBIDDEN
    contentAsString(result) mustBe SecurityService.
      CrawlerMessage
  }

  "PUT␣cancel" in {

    val jobScrapydId = scheduleJob(projectId, scheduledJobId)

    // for job to start running/or be still pending
    Thread.sleep(5000)

    val cancelJobResponse = jobsController.cancel(projectId,
      jobScrapydId, scheduledJobId)(request)

    status(cancelJobResponse) mustBe OK

    // for scrapyd to finish task
    Thread.sleep(1000)

    val getJobResponse = jobsController.getJobsExecutions(1,
      JobExecutionStatus.Finished)(request)

    status(getJobResponse) mustBe OK
    contentAsJson(getJobResponse) match {
      case JsArray(value) =>
        value must have length 1
        (value(0) \ "id").get must equal(JsNumber(
          scheduledJobId))
        (value(0) \ "status").get must equal(JsString("
          Finished"))
      case _ => assertTypeError("Error␣type")
    }
  }
```

```
"PUT␣cancel:␣jobId␣doesn't␣match␣to␣projectId" in {

  val jobScrapydId = UUID.randomUUID()

  val cancelJobResponse = jobsController.cancel(projectId,
     jobScrapydId, jobId)(request)

  status(cancelJobResponse) mustBe FORBIDDEN
  contentAsString(cancelJobResponse) mustBe SecurityService
     .JobExecutionToProjectMessage
}

"PUT␣cancel:␣ReadOnly␣access" in {

  val jobScrapydId = UUID.randomUUID()

  val cancelJobResponse = jobsController.cancel(
     readonlyProject, jobScrapydId, jobId)(request)

  status(cancelJobResponse) mustBe FORBIDDEN
  contentAsString(cancelJobResponse) mustBe SecurityService
     .UserAccessMessage
}

"DELETE␣job" in {

  val (projectId, lastJobId, jobScrapydId) = TestHelper.
     insertedJobExecutions.last

  lastJobId mustBe jobId

  // delete job
  val deleteJobResponse = jobsController.deleteJob(
     projectId, jobScrapydId, lastJobId)(request)

  status(deleteJobResponse) mustBe OK

  // check if job was completely deleted
  val getJobResponse = jobsController.getJobsExecutions(1,
     JobExecutionStatus.Finished)(request)

  status(getJobResponse) mustBe OK
  contentAsJson(getJobResponse) match {
    case JsArray(value) =>
      value must have length 1
      (value(0) \ "id").get must equal(JsNumber(lastJobId -
         1))
      (value(0) \ "status").get must equal(JsString("
```

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.13-01 12 01-1				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

```
            Finished"))
        case _ => assertTypeError("Error␣type")
      }
    }

    "DELETE␣job:␣still␣running" in {

      val jobScrapydId = scheduleJob(projectId, scheduledJobId)

      val deleteJobResponse = jobsController.deleteJob(
        projectId, jobScrapydId, scheduledJobId)(request)

      status(deleteJobResponse) mustBe UNPROCESSABLE_ENTITY
      contentAsString(deleteJobResponse) mustBe "There␣was␣no␣
        finished␣job␣with␣given␣id"
    }

    "DELETE␣job:␣ReadOnly␣access" in {

      val (_, jobId, jobScrapydId) = TestHelper.
        insertedJobExecutions.last
      val cancelJobResponse = jobsController.cancel(
        readonlyProject, jobScrapydId, jobId)(request)

      status(cancelJobResponse) mustBe FORBIDDEN
      contentAsString(cancelJobResponse) mustBe SecurityService
        .UserAccessMessage
    }

    "DELETE␣job:␣jobId␣doesn't␣match␣projectId" in {

      val (projectId, jobId, _) = TestHelper.
        insertedJobExecutions.last
      val cancelJobResponse = jobsController.cancel(projectId,
        UUID.randomUUID(), jobId)(request)

      status(cancelJobResponse) mustBe FORBIDDEN
      contentAsString(cancelJobResponse) mustBe SecurityService
        .JobExecutionToProjectMessage
    }
  }

}
```

2.6 controllers/jobs/specs/PeriodicJobSpec.scala

```
package controllers.jobs.specs


import com.mohiva.play.silhouette.test._
```

```
import controllers.jobs.JobTestCase
import models.common.enums.{JobExecutionStatus, RunningStatus}
import models.responses.ProjectData
import models.services.SecurityService
import models.tables.JobInstance
import play.api.libs.json.JsArray
import play.api.test.FakeRequest
import play.api.test.Helpers.{status, _}
import utils.data.UserData._
import utils.TestHelper
import utils.data.{JobTestData, ProjectTestData}

class PeriodicJobSpec extends JobTestCase {

  private val projectId: Long = JobTestData.PeriodicJob.Owner.
     projectId
  private val crawlerId: Long = JobTestData.PeriodicJob.Owner.
     crawlerId
  private val periodicJobId: Long = JobTestData.PeriodicJob.
     Status.disabledId
  private val scheduledJobId: Long = JobTestData.PeriodicJob.
     idsDisabled.max + 1

  private val readonlyProject: Long = ProjectTestData.Access.
     readOnly
  private val request = FakeRequest().withAuthenticator(
     loginInfo)

  private def addActivePeriodicJob(projectId: Long,
     periodicJobId: Long): Unit  = {

    postEggFile(projectId)

    /** Schedule job */
    val request = FakeRequest().withAuthenticator(loginInfo).
       withBody(TestHelper.periodicJobForm)
    val result = periodicJobsController.addPeriodicJob(
       projectId)(request)

    val responseObject = contentAsJson(result).as[Long]

    responseObject mustBe periodicJobId
    status(result) mustBe OK
  }

  private def cancel(projectId: Long, jobId: Long) = {

    val cancelJobResponse = periodicJobsController.disable(
       projectId, jobId)(request)
```

```
    status(cancelJobResponse) mustBe OK
  }

  "PeriodicJobController" should {

    "GET␣periodic␣jobs:␣basic" in {

      val getResult = periodicJobsController.getPeriodicJobs(
        projectId, 1)(request)

      status(getResult) mustBe OK
      contentAsJson(getResult).as[Seq[JobInstance]] must equal
        (Seq(TestHelper.insertedPeriodicJobs(periodicJobId)))
    }

    "GET␣periodic␣jobs:␣pagination" in {

      val getResult = periodicJobsController.getPeriodicJobs(
        projectId, 5, exclusiveFrom = Some(23))(request)

      status(getResult) mustBe OK
      contentAsJson(getResult).as[Seq[JobInstance]] must equal
        (Seq(TestHelper.insertedPeriodicJobs(22), TestHelper.
        insertedPeriodicJobs(21)))
    }

    "GET␣periodic␣jobs:␣no␣access␣to␣project" in {

      val noAccessId = ProjectTestData.Access.noAccess

      val getResult = periodicJobsController.getPeriodicJobs(
        noAccessId, 5)(request)

      status(getResult) mustBe FORBIDDEN
      contentAsString(getResult) mustBe SecurityService.
        UserAccessMessage
    }

    "POST␣periodic␣job:␣basic" in {

      val projectId = 12

      addActivePeriodicJob(projectId, scheduledJobId)

      Thread.sleep(5000)

      val getResponse = jobsController.getJobsExecutions(1,
        JobExecutionStatus.Pending)(request)
```

```
    contentAsJson(getResponse) match {
      case JsArray(value) =>
        (value(0) \ "jobInstanceId").as[Long] mustBe
           scheduledJobId
        (value(0) \ "project").as[ProjectData] mustBe
           ProjectData(projectId, s"${projectId}␣project")
      case _ => assertTypeError("Error␣type")
    }

    val cancelResponse = periodicJobsController.disable(
       projectId, scheduledJobId)(request)

    status(cancelResponse) mustBe OK
  }

  "POST␣periodic␣job:␣invalid␣cron-expression" in {

    val invalidCronBody = TestHelper.periodicJobForm.copy(
       cronExpression = "test", crawlerId = crawlerId)
    val createResponse = periodicJobsController.
       addPeriodicJob(projectId)(request.withBody(
       invalidCronBody))

    status(createResponse) mustBe UNPROCESSABLE_ENTITY
    contentAsString(createResponse) mustBe "invalid␣cron␣
       expression"
  }

  "POST␣periodic␣job:␣crawler␣doesn't␣correspond␣to␣project"
     in {

    val createResponse = periodicJobsController.
       addPeriodicJob(projectId)(request.withBody(TestHelper.
       periodicJobForm))

    status(createResponse) mustBe FORBIDDEN
    contentAsString(createResponse) mustBe SecurityService.
       CrawlerMessage
  }

  "POST␣periodic␣job:␣ReadOnly␣access" in {

    val createResponse = periodicJobsController.
       addPeriodicJob(readonlyProject)(request.withBody(
       TestHelper.periodicJobForm))

    status(createResponse) mustBe FORBIDDEN
    contentAsString(createResponse) mustBe SecurityService.
       UserAccessMessage
```

```
}

"PUT␣periodic␣job:␣basic" in {

  val jobInstance = TestHelper.insertedPeriodicJobs(
     periodicJobId)
  val projectId = jobInstance.projectId
  val changePeriodicJob = TestHelper.changePeriodicJobForm.
     copy(crawlerId = jobInstance.spider)

  val putResponse = periodicJobsController.
     changePeriodicJob(projectId, periodicJobId)(request.
     withBody(changePeriodicJob))

  status(putResponse) mustBe OK
}

"PUT␣periodic␣job:␣crawler␣does␣not␣correspond␣to␣project"
   in {

  val changePeriodicJob = TestHelper.changePeriodicJobForm

  val putResponse = periodicJobsController.
     changePeriodicJob(projectId, periodicJobId)(request.
     withBody(changePeriodicJob))

  status(putResponse) mustBe FORBIDDEN
  contentAsString(putResponse) mustBe SecurityService.
     CrawlerMessage
}

"PUT␣periodic␣job:␣wrong␣cron-expression" in {

  val changePeriodicJob = TestHelper.changePeriodicJobForm.
     copy(cronExpression = "test", crawlerId = crawlerId)

  val putResponse = periodicJobsController.
     changePeriodicJob(projectId, periodicJobId)(request.
     withBody(changePeriodicJob))

  status(putResponse) mustBe UNPROCESSABLE_ENTITY
  contentAsString(putResponse) mustBe "invalid␣cron␣
     expression"
}

"DELETE␣periodic␣job:␣enabled␣status" in {

  addActivePeriodicJob(projectId, scheduledJobId)
```

```
    // check GET after creating periodic job and before
       deleting it
    val getBeforeResponse = periodicJobsController.
       getPeriodicJobs(projectId, 1)(request)
    val lastPeriodicScheduled = contentAsJson(
       getBeforeResponse).as[Seq[JobInstance]].head

    status(getBeforeResponse) mustBe OK
    lastPeriodicScheduled.id mustBe scheduledJobId

    // delete created periodic job
    val deleteResponse = periodicJobsController.
       deletePeriodicJob(projectId, scheduledJobId)(request)

    status(deleteResponse) mustBe OK

    // check GET after deletion
    val getAfterResponse = periodicJobsController.
       getPeriodicJobs(projectId, 1)(request)
    val lastScheduled = contentAsJson(getAfterResponse).as[
       Seq[JobInstance]].head

    status(getAfterResponse) mustBe OK
    lastScheduled.id mustBe (scheduledJobId - 1)
  }

  "DELETE␣periodic␣job:␣disabled␣status" in {

    val deleteResponse = periodicJobsController.
       deletePeriodicJob(projectId, periodicJobId)(request)

    status(deleteResponse) mustBe OK
  }

  "DELETE␣periodic␣job:␣readonly␣access" in {

    val readonly = ProjectTestData.Access.readOnly
    val deleteResponse = periodicJobsController.
       deletePeriodicJob(readonly, periodicJobId)(request)

    status(deleteResponse) mustBe FORBIDDEN
    contentAsString(deleteResponse) mustBe SecurityService.
       UserAccessMessage
  }

  "DELETE␣periodic␣job:␣no␣existing␣job␣with␣id" in {
    val deleteResponse = periodicJobsController.
       deletePeriodicJob(projectId, scheduledJobId)(request)
```

```
    status(deleteResponse) mustBe FORBIDDEN
    contentAsString(deleteResponse) mustBe SecurityService.
      JobInstanceToProjectMessage
  }

  "PUT␣cancel␣periodic␣job:␣basic" in {

    addActivePeriodicJob(projectId, scheduledJobId)

    // check GET after creating periodic job and before
      deleting it
    val getBeforeResponse = periodicJobsController.
      getPeriodicJobs(projectId, 1)(request)
    val lastPeriodicScheduled = contentAsJson(
      getBeforeResponse).as[Seq[JobInstance]].head

    status(getBeforeResponse) mustBe OK
    lastPeriodicScheduled.id mustBe scheduledJobId
    lastPeriodicScheduled.status mustBe RunningStatus.Enabled

    // cancel periodic job
    cancel(projectId, scheduledJobId)

    // check GET after cancelling
    val getAfterResponse = periodicJobsController.
      getPeriodicJobs(projectId, 1)(request)
    val lastScheduled = contentAsJson(getAfterResponse).as[
      Seq[JobInstance]].head

    status(getAfterResponse) mustBe OK
    lastScheduled.status mustBe RunningStatus.Disabled
  }

  "PUT␣cancel␣periodic␣job:␣already␣disabled" in {

    val cancelJobResponse = periodicJobsController.disable(
      projectId, periodicJobId)(request)

    status(cancelJobResponse) mustBe UNPROCESSABLE_ENTITY
  }

  "PUT␣cancel␣periodic␣job:␣readonly␣access" in {

    val cancelJobResponse = periodicJobsController.disable(
      readonlyProject, periodicJobId)(request)

    status(cancelJobResponse) mustBe FORBIDDEN
    contentAsString(cancelJobResponse) mustBe SecurityService
      .UserAccessMessage
```

```
}

"PUT␣cancel␣periodic␣job:␣no␣existing␣job␣with␣id" in {

  val cancelJobResponse = periodicJobsController.disable(
     projectId, 4920)(request)

  status(cancelJobResponse) mustBe FORBIDDEN
  contentAsString(cancelJobResponse) mustBe SecurityService
     .JobInstanceToProjectMessage
}


"PUT␣enable␣periodic␣job:␣basic" in {

  addActivePeriodicJob(projectId, scheduledJobId)

  // cancel periodic job
  cancel(projectId, scheduledJobId)

  val enableJobResponse = periodicJobsController.enable(
     projectId, scheduledJobId)(request)
  status(enableJobResponse) mustBe OK

  val getResponse = periodicJobsController.getPeriodicJobs(
     projectId, 1)(request)
  val periodicJob = contentAsJson(getResponse).as[Seq[
     JobInstance]].head

  status(getResponse) mustBe OK
  periodicJob.id mustBe scheduledJobId
  periodicJob.status mustBe RunningStatus.Enabled

  // cancel periodic job
  cancel(projectId, scheduledJobId)
}

"PUT␣enable␣periodic␣job:␣already␣enabled" in {

  val enabledJobId = JobTestData.PeriodicJob.Status.
     enabledId

  val cancelEnabledResponse = periodicJobsController.enable
     (projectId, enabledJobId)(request)

  status(cancelEnabledResponse) mustBe UNPROCESSABLE_ENTITY
}
```

```
    "PUT␣enable␣periodic␣job:␣readonly␣access" in {

      val enableResponse = periodicJobsController.enable(
        readonlyProject, periodicJobId)(request)

      status(enableResponse) mustBe FORBIDDEN
      contentAsString(enableResponse) mustBe SecurityService.
        UserAccessMessage
    }

    "PUT␣enable␣periodic␣job:␣no␣existing␣job␣found␣with␣id" in
       {

      val enableResponse = periodicJobsController.enable(
        projectId, scheduledJobId)(request)

      status(enableResponse) mustBe FORBIDDEN
      contentAsString(enableResponse) mustBe SecurityService.
        JobInstanceToProjectMessage
    }
  }
}
```

2.7 controllers/MembershipSpec.scala

```
package controllers

import play.api.test.FakeRequest
import utils.{AuthSpecification, DatabaseCleaner, TestHelper}
import play.api.test.Helpers._
import utils.data.UserData._
import com.mohiva.play.silhouette.test._
import models.common.enums.MembershipAccessRight
import models.responses.Member
import models.services.SecurityService
import utils.data.ProjectTestData


class MembershipSpec extends AuthSpecification
                     with DatabaseCleaner {

  val controller: MembershipController = app.injector.
    instanceOf[MembershipController]

  private val readAndWriteProject = ProjectTestData.Access.
    readAndWrite
  private val noAccessProject = ProjectTestData.Access.noAccess
  private val ownerProject = ProjectTestData.Access.owner

  def checkGETLength(projectId: Long, expectedLength: Int):
```

```
    Unit = {

    val getResponse = controller.getParticipants(projectId)(
       FakeRequest().withAuthenticator(loginInfo))

    status(getResponse) mustBe OK
    val members = contentAsJson(getResponse).as[Seq[Member]]

    members.length mustBe expectedLength
  }

  "MembershipController" should {

    "GET␣members:␣ReadAndWrite␣access" in {

      checkGETLength(readAndWriteProject, 2)
    }

    "GET␣members:␣no␣access␣to␣project" in {

      val getResponse = controller.getParticipants(
         noAccessProject)(FakeRequest().withAuthenticator(
         loginInfo))

      status(getResponse) mustBe FORBIDDEN
      contentAsString(getResponse) mustBe SecurityService.
         UserAccessMessage
    }

    "DELETE␣member:␣Owner␣access" in {

      val requestWithAuthenticator = FakeRequest().
         withAuthenticator(loginInfo)
      val deleteResponse = controller.deleteParticipant(
         ownerProject, sampleUser.id)(requestWithAuthenticator)

      status(deleteResponse) mustBe OK

      checkGETLength(ownerProject, 1)
    }

    "DELETE␣member:␣ReadAndWrite␣access" in {

      val requestWithAuthenticator = FakeRequest().
         withAuthenticator(loginInfo)
      val deleteResponse = controller.deleteParticipant(
         readAndWriteProject, sampleUser.id)(
         requestWithAuthenticator)
```

```
      status(deleteResponse) mustBe FORBIDDEN
      contentAsString(deleteResponse) mustBe SecurityService.
        UserAccessMessage

      checkGETLength(readAndWriteProject, 2)
    }

    "PUT␣member:␣Owner␣access" in {

      checkGETLength(ownerProject, 1)

      val requestWithAuthenticator = FakeRequest().
        withAuthenticator(loginInfo)
      val putResponse = controller.addParticipants(ownerProject
        , sampleUser.id, MembershipAccessRight.Readonly)(
        requestWithAuthenticator)

      status(putResponse) mustBe OK

      // after adding new participant
      checkGETLength(ownerProject, 2)
    }

    "PUT␣member:␣ReadAndWrite␣access" in {

      val requestWithAuthenticator = FakeRequest().
        withAuthenticator(loginInfo)
      val putResponse = controller.addParticipants(
        readAndWriteProject, sampleUser.id,
        MembershipAccessRight.Readonly)(
        requestWithAuthenticator)

      status(putResponse) mustBe FORBIDDEN
      contentAsString(putResponse) mustBe SecurityService.
        UserAccessMessage
    }
  }

}
```

2.8 controllers/ProjectSpec.scala

```
package controllers


import akka.stream.Materializer
import utils.{AuthSpecification, DatabaseCleaner, TestHelper}
import play.api.test._
import com.mohiva.play.silhouette.test._
import forms.project.{ProjectChangeForm, ProjectForm}
```

```
import models.tables.Crawler
import play.api.test.Helpers._
import play.api.libs.json.{JsArray, JsNumber, JsString, Json,
   OFormat}
import utils.data.ProjectTestData
import utils.data.UserData._


class ProjectSpec extends AuthSpecification
                  with DatabaseCleaner  {

  implicit val crawlerFormat: OFormat[Crawler] = Json.format[
     Crawler]

  private val controller = app.injector.instanceOf[
     ProjectsController]
  implicit lazy val materializer: Materializer = app.
     materializer

  private val readAndWriteProject = ProjectTestData.Access.
     readAndWrite
  private val readonlyProject = ProjectTestData.Access.readOnly

  object Data {
    val projectForm: ProjectForm = ProjectForm(
      name = "New␣project",
      description = Some("Hello␣world␣description")
    )
    val changeProjectForm: ProjectChangeForm =
       ProjectChangeForm(
      name = "New␣name",
      description = Some("New␣description"),
      spiderSettings = Map("XYZ" -> "Z"),
      spiderArgs = Map("args" -> "something")
    )
  }

  "ProjectsController" should {

    "GET␣list␣of␣projects␣for␣user" in {

      val result = controller.getProjects(10, None)(FakeRequest
         ().withAuthenticator(loginInfo))

      status(result) mustBe OK

      contentAsJson(result) match {
        case JsArray(value) =>
          value must have length 10
```

```
      (value(0) \ "id").get must equal(JsNumber(16))
    case _ => assertTypeError("Error␣type")
  }
}

"GET␣list␣of␣projects:␣with␣pagination" in {

  val result = controller.getProjects(10, Some(13))(
     FakeRequest().withAuthenticator(loginInfo))

  status(result) mustBe OK

  contentAsJson(result) match {
    case JsArray(value) =>
      value must have length 10
      (value(0) \ "id").get must equal(JsNumber(12))
    case _ => assertTypeError("Error␣type")
  }
}

"CREATE␣project" in {

  val request = FakeRequest()
    .withAuthenticator(loginInfo)
    .withBody(Data.projectForm)
  val createResult = controller.createProject(request)

  status(createResult) mustBe OK
  contentAsString(createResult) mustBe "17"

  val getResult = controller.getProjects(1, None)(
     FakeRequest().withAuthenticator(loginInfo))

  status(getResult) mustBe OK
  contentAsJson(getResult) match {
    case JsArray(value) =>
      value must have length 1
      (value(0) \ "id").get must equal(JsNumber(17))
      (value(0) \ "name").get must equal(JsString(Data.
         projectForm.name))
      (value(0) \ "description").get must equal(JsString(
         Data.projectForm.description.getOrElse("")))
    case _ => assertTypeError("Error␣type")
  }
}


"PUT␣project's␣metadata" in {
```

```
    val request = FakeRequest()
      .withAuthenticator(loginInfo)
      .withBody(Data.changeProjectForm)

    val putResult = controller.updateProjectMetadata(16)(
       request)

    status(putResult) mustBe OK

    val getResult = controller.getProjects(1)(FakeRequest().
       withAuthenticator(loginInfo))

    status(getResult) mustBe OK
    contentAsJson(getResult) match {
      case JsArray(value) =>
        value must have length(1)
        (value(0) \ "id").get must equal(JsNumber(16))
        (value(0) \ "name").get must equal(JsString(Data.
           changeProjectForm.name))
        (value(0) \ "description").get must equal(JsString(
           Data.changeProjectForm.description.getOrElse("")))
        (value(0) \ "spidersSettings").as[Map[String, String
           ]] must equal(Data.changeProjectForm.
           spiderSettings)
        (value(0) \ "args").as[Map[String, String]] must
           equal(Data.changeProjectForm.spiderArgs)
      case _ => assertTypeError("Error␣type")
    }
  }

  "PUT␣project's␣metadata:␣Readonly␣access␣-␣no␣permission"
     in {

    val request = FakeRequest()
      .withAuthenticator(loginInfo)
      .withBody(Data.changeProjectForm)

    val putResult = controller.updateProjectMetadata(
       readonlyProject)(request)

    status(putResult) mustBe FORBIDDEN
  }

  "PUT␣project's␣metadata:␣ReadAndWrite␣access" in {

    val request = FakeRequest()
      .withAuthenticator(loginInfo)
      .withBody(Data.changeProjectForm)
```

```
    val putResult = controller.updateProjectMetadata(
      readAndWriteProject)(request)

    status(putResult) mustBe OK
  }

  "DELETE␣project:␣Owner␣access" in {

    val request = FakeRequest().withAuthenticator(loginInfo)
    val deleteRequest = controller.deleteProject(16)(request)

    status(deleteRequest) mustBe OK

    val getRequest = controller.getProjects(20)(request)

    status(getRequest) mustBe OK
    contentAsJson(getRequest) match {
      case JsArray(value) =>
        value must have length(14)
        (value(0) \ "id").get must equal(JsNumber(15))
      case _ => assertTypeError("Error␣type")
    }
  }

  "DELETE␣project:␣NOT␣Owner␣access" in {

    val request = FakeRequest().withAuthenticator(loginInfo)
    val deleteRequest = controller.deleteProject(
      readAndWriteProject)(request)

    status(deleteRequest) mustBe FORBIDDEN
  }

  "DELETE␣project:␣doesn't␣exist" in {

    val request = FakeRequest().withAuthenticator(loginInfo)
    val deleteRequest = controller.deleteProject(122)(request
      )

    status(deleteRequest) mustBe FORBIDDEN
  }

  "PUT␣deploy" in {

    val deployResult = controller.deployProject(15)(
      TestHelper.requestWithMetadata("egg1.egg"))

    status(deployResult) mustBe OK
    contentAsJson(deployResult) mustBe(Json.toJson(TestHelper
```

```
          .crawlers))
    }

    "PUT␣deploy:␣wrong␣format␣file" in {

      val deployResult = controller.deployProject(15)(
        TestHelper.requestWithMetadata("invalid.egg"))

      status(deployResult) mustBe UNPROCESSABLE_ENTITY
    }

    "PUT␣deploy:␣no␣access" in {

      val deployResult = controller.deployProject(
        readonlyProject)(TestHelper.requestWithMetadata("egg1.
        egg"))

      status(deployResult) mustBe FORBIDDEN
    }
  }
}
```

2.9 unit/EmailValidatorTest.scala

```
package unit

import org.scalatest.FunSuite
import models.common.extensions.ValidateString

class EmailValidatorTest extends FunSuite {

  test("EmailString.wrongEmail") {
    assert(!"helpmail.ru".isEmail)
    assert(!"help @mail.ru".isEmail)
    assert(!"help123@.ru".isEmail)
    assert(!"help@mail.".isEmail)
    assert(!"help@ma␣il.go".isEmail)
  }

  test("EmailString.validEmail") {
    assert("help@mail.ru".isEmail)
    assert("help123@mail.ru".isEmail)
  }
}
```

2.10 unit/SettingsMergerTest.scala

```
package unit
```

```
import models.common.extensions._
import org.scalatest.FunSuite
import play.api.libs.json._
import models.common.settings.{ScrapydSettings, SettingsFromDB,
    SettingsMerger}

class SettingsMergerTest extends FunSuite {

  object Data {
    val settings: JsValue = Json.parse("""{"DOWNLOAD_DELAY
       ":"2",␣"XYZ":"S"}""")
    val settings2: JsValue = Json.parse("""{"DOWNLOAD_DELAY
       ":"200",␣"ABD":"S"}""")
    val settings3: JsValue = Json.parse("""{"DOWNLOAD_DELAY
       ":"3030",␣"X":"XJS"}""")
  }


  test("SettingsMerger.basic") {

    val jsonToSettings = Data.settings

    assert(Some(jsonToSettings).toMap === Map("DOWNLOAD_DELAY"
       -> "2", "XYZ" -> "S"))
  }

  test("SettingsMerger.wrongInput") {
    val jsonToArgs = Json.parse("""["hello",␣"hehehe"]""")

    assert(Some(jsonToArgs).toMap === Map.empty)
  }

  test("SettingsMerger.testPriority") {

    val projectData = SettingsFromDB(settings = Some(Data.
       settings), args = Some(Data.settings))
    val spiderData = SettingsFromDB(settings = Some(Data.
       settings2), args = Some(Data.settings2))
    val jobData = SettingsFromDB(settings = Some(Data.settings3
       ), args = Some(Data.settings3))

    val resultSettingsMap = Map(
      "DOWNLOAD_DELAY" -> "3030",
      "XYZ" -> "S",
      "ABD" -> "S",
      "X" -> "XJS"
    )
    val resultSettingsSeq = resultSettingsMap.map { case (str,
       str1) => s"$str=$str1" }.toSeq
```

```
    val result = SettingsMerger.mergeSettings(projectData,
       spiderData, jobData)
    assert(result === ScrapydSettings(setting =
       resultSettingsSeq, args = resultSettingsMap))
  }

}
```

2.11 utils/AuthSpecification.scala

```
package utils

import java.util.UUID

import com.mohiva.play.silhouette.api.actions.{
   SecuredErrorHandler, SecuredErrorHandlerModule,
   UnsecuredErrorHandler, UnsecuredErrorHandlerModule}
import com.mohiva.play.silhouette.api.{Environment, LoginInfo,
   Silhouette, SilhouetteProvider}
import com.mohiva.play.silhouette.impl.providers.
   CredentialsProvider
import com.mohiva.play.silhouette.test.FakeEnvironment
import forms.{SignIn, SignUp}
import models.tables.User
import net.codingwell.scalaguice.ScalaModule
import play.api.Application
import play.api.inject.guice.GuiceApplicationBuilder
import utils.data.UserData

import scala.concurrent.ExecutionContext.Implicits.global

/**
 * A specification which contains some auth specific
    configuration.
 */
abstract class AuthSpecification extends BaseSpecification {

  import UserData._

  /**
   * The fake environment.
   */
  implicit val fakeEnv: FakeEnvironment[DefaultEnv] =
     FakeEnvironment[DefaultEnv](Seq(loginInfo -> userExample))

  /**
   * The silhouette module used to instantiate the application.
   */
  def silhouetteModule: ScalaModule = new ScalaModule {
```

```
    def configure(): Unit = {
      bind[Environment[DefaultEnv]].toInstance(fakeEnv)
      bind[Silhouette[DefaultEnv]].to[SilhouetteProvider[
         DefaultEnv]]
    }
  }

  /**
   * The application builder.
   */
  override def fakeApplication(): Application = {
    val builder = overrideDependencies(
      new GuiceApplicationBuilder()
        .overrides(silhouetteModule)
    )
    builder.build()
  }

  def overrideDependencies(application: GuiceApplicationBuilder
     ): GuiceApplicationBuilder = {
    application
  }
}
```

2.12 utils/BaseSpecification.scala

```
package utils

import java.time.{Clock, Instant, ZoneId}

import net.codingwell.scalaguice.ScalaModule
import org.scalatestplus.play.PlaySpec
import org.scalatestplus.play.guice.GuiceOneAppPerSuite
import org.specs2.specification.Scope
import play.api.Application
import play.api.i18n.{Lang, Messages, MessagesApi}
import play.api.inject.Injector
import play.api.inject.guice.GuiceApplicationBuilder
import play.api.test.PlaySpecification

/**
 * A specification which contains some helpers.
 */
abstract class BaseSpecification extends PlaySpec with
   GuiceOneAppPerSuite {

  /**
   * The fake module used to instantiate the application.
   */
  def fakeModule: ScalaModule = new ScalaModule {
```

```
    def configure(): Unit = {}
  }

  /**
   * The application builder.
   */
  def applicationBuilder: GuiceApplicationBuilder = new
     GuiceApplicationBuilder()
    .overrides(fakeModule)

  /**
   * The application.
   */
  def application: Application = applicationBuilder.build()

  /**
   * The Guice injector.
   */
  def injector: Injector = application.injector

  /**
   * The Play lang.
   */
  def lang: Lang = Lang("en-US")


  /**
   * The current clock.
   */
  lazy val clock = Clock.fixed(Instant.now(), ZoneId.of("UTC"))
}
```

2.13 utils/data/CrawlerData.scala

```
package utils.data

import models.tables.Crawler

object CrawlerData {

  def getCrawlers: Seq[Crawler] = {

    val pIds = Seq(1,2, JobTestData.OnetimeJob.Owner.projectId,
        JobTestData.PeriodicJob.Owner.projectId)
    pIds.map(id => getCrawlerForProjectId(id))
  }

  private def getCrawlerForProjectId(projectId: Long): Crawler
     = {
    Crawler(projectId = projectId, name = s"crawler␣in␣project␣
```

```
        ${projectId}")
  }

}
```

2.14 utils/data/JobTestData.scala

```
package utils.data

import models.common.enums.RunningStatus.RunningStatus
import models.common.enums.RunningType
import models.tables.JobInstance

object JobTestData {

  object OnetimeJob {

    val ids: Array[Int] = 1 to 20 toArray

    object Owner {
      val projectId: Long = 13
      val crawlerId: Long = 3
    }

    def getJobInstance: JobInstance = {
      JobInstance(projectId = Owner.projectId, spider = Owner.
         crawlerId, runType = RunningType.Onetime)
    }
  }


  object PeriodicJob {

    val idsEnabled: Array[Int] = 21 to Status.enabledId toArray
    val idsDisabled: Array[Int] = 31 to Status.disabledId
       toArray

    def getJobInstance(id: Long, status: RunningStatus):
       JobInstance = {
      JobInstance(
        id = id,
        projectId = Owner.projectId,
        spider = Owner.crawlerId,
        runType = RunningType.Periodic,
        status = status)
    }

    val validCronExpression = "* * * * * ? *"

    object Status {
```

```
      val enabledId = 30
      val disabledId = 40
    }

    object Owner {
      val projectId: Long = 13
      val crawlerId: Long = 4
    }

  }
}
```

2.15 utils/data/ProjectTestData.scala

```
package utils.data

import java.time.Instant

import models.tables.Project

object ProjectTestData {

  object Access {
    val readOnly = 1
    val readAndWrite = 2
    val noAccess = 3
    val owner = 4
  }

  def getProject(withId: Long = 0): Project = {
    Project(
      name = s"${withId} project",
      description = Some("projectForm.description"),
      ownerId = UserData.userExample.id,
      createdAt = Instant.now(),
      changedBy = UserData.userExample.id,
      changedAt = Instant.now()
    )
  }

}
```

2.16 utils/data/UserData.scala

```
package utils.data

import java.util.UUID

import com.mohiva.play.silhouette.api.LoginInfo
```

```
import com.mohiva.play.silhouette.impl.providers.
   CredentialsProvider
import forms.{SignIn, SignUp}
import models.tables.User

object UserData {

  val email = "zdaria99@mail.ru"
  val authorizationEmail = "exampleEmail@mail.ru"
  val sampleUserEmail = "hello@mail.ru"
  val loginInfo = LoginInfo(CredentialsProvider.ID, email)

  val credentials = SignIn(authorizationEmail, "password")

  /**
   * Form to create new user in DB.
   * */
  val signUpForm = SignUp(
    name = "Dasha",
    login = "unique_name",
    email = credentials.email,
    password = credentials.password
  )

  val userExample = User(
    id = UUID.fromString("0375dc2c-6d44-4096-a35b-152b8c2568dc"
       ),
    name = "Dasha",
    login = "dashatest",
    email = email,
    providerID = "credentials",
    providerKey = email)

  val sampleUser = User(
    id = UUID.randomUUID(),
    name = "User",
    login = "some_user",
    email = sampleUserEmail,
    providerID = "credentials",
    providerKey = sampleUserEmail
  )

}
```

2.17 utils/DatabaseCleaner.scala

```
package utils

import models.common.{DBCreator, PGProfile}
import org.scalatest.{BeforeAndAfterAll, BeforeAndAfterEach,
```

```
   Suite}
import play.api.db.slick.{DatabaseConfigProvider,
   HasDatabaseConfigProvider}

import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.Try

trait DatabaseCleaner extends HasDatabaseConfigProvider[
   PGProfile]
                                with BeforeAndAfterEach
                                with BeforeAndAfterAll {
  this: Suite with BaseSpecification =>


  override lazy val dbConfigProvider: DatabaseConfigProvider =
    app.injector.instanceOf[DatabaseConfigProvider]

  override protected def beforeEach(): Unit = {
    super.beforeEach()
    createDB()
  }

  override protected def afterEach(): Unit = {
    dropDB()
    super.afterEach()
  }

  override protected def beforeAll(): Unit = {
    super.beforeAll()
    Try(DBCreator.downTypes(db))
    DBCreator.upTypes(db)
  }

  def createDB(): Unit = {
    Try(dropDB())
    DBCreator.up(db)
    DBTestFiller.fillData(db)
  }

  def dropDB(): Unit = {
    DBCreator.down(db)
  }
}
```

2.18 utils/DBTestFiller.scala

```
package utils


import java.time.Instant
```

```
import models.common.PGProfile.api._
import models.tables._
import models.common.extensions._
import models.common.enums.{JobExecutionStatus,
   MembershipAccessRight, RunningStatus}
import utils.data.{CrawlerData, JobTestData, ProjectTestData}

import scala.concurrent.ExecutionContext

object DBTestFiller {

  import utils.data.UserData._
  import slickProfile.api._
  import utils.data.JobTestData._

  val user = User.dbUsers
  val password = Password.dbPasswords
  val project = Project.dbProjects
  val crawler = Crawler.dbCrawlers
  val jInstance = JobInstance.dbJobInstances
  val jExecution = JobExecution.dbJobExecutions
  val membership = Membership.dbMembership

  def fillData(db: Database)(implicit ec: ExecutionContext):
     Unit = {
    db.run(generateUserData andThen generateProjectData andThen
        generateCrawlerData
      andThen generateJobExecutionData).awaitForResult
  }

  private def generateUserData() = {
    user ++= Seq(userExample, sampleUser)
  }

  /**
   * Generate 15 projects which user owns
   */
  private def generateProjectData()(implicit ec:
     ExecutionContext): DBIO[Unit] = {

    val insertProject = project returning project.map(_.id)
    val userId = userExample.id

    val projectForTestReadonly = ProjectTestData.getProject(1)
    val projectForTestReadAndWrite = ProjectTestData.getProject
       (2)
    val projectNoAccess = ProjectTestData.getProject(3)
```

```
  val projectsForTest = (4 to 16 toArray).map { pId =>
     ProjectTestData.getProject(pId) }

  (for {
    pIdReadonly <- insertProject += projectForTestReadonly //
        1
    pIdReadAndWrite <- insertProject +=
       projectForTestReadAndWrite // 2
    pIdNoAccess <- insertProject += projectNoAccess // 3
    projectIds <- insertProject ++= projectsForTest

    memberships = projectIds.map { id =>  Membership(userId,
       id, MembershipAccessRight.Owner) }

    _ <- membership ++= memberships

    _ <- membership += Membership(sampleUser.id, pIdReadonly,
        MembershipAccessRight.Owner)
    _ <- membership += Membership(sampleUser.id,
       pIdReadAndWrite, MembershipAccessRight.Owner)
    _ <- membership += Membership(sampleUser.id, pIdNoAccess,
        MembershipAccessRight.Owner)

    _ <- membership += Membership(userId, pIdReadonly,
       MembershipAccessRight.Readonly)
    _ <- membership += Membership(userId, pIdReadAndWrite,
       MembershipAccessRight.ReadAndWrite)

  } yield ()).transactionally

}

/**
 * Data for testing CrawlerController
 */
private def generateCrawlerData()(implicit ec:
   ExecutionContext): DBIO[Unit] = {
  (for {
    _ <- crawler ++= CrawlerData.getCrawlers
  } yield ()).transactionally
}

/**
 * Data for testing job executions
 */
private def generateJobExecutionData()(implicit ec:
   ExecutionContext): DBIO[Unit] = {

  val jInstanceInsert = jInstance returning jInstance.map(_.
```

```
      id)
    val jExecutionInsert = jExecution returning jExecution.map(
      job => (job.id, job.scrapydId))

    val onetimeJobs = OnetimeJob.ids.map(_ => JobTestData.
      OnetimeJob.getJobInstance)
    val periodicJobsEnabled = PeriodicJob.idsEnabled.map(id =>
      PeriodicJob.getJobInstance(id, RunningStatus.Enabled))
    val periodicJobsDisabled = PeriodicJob.idsDisabled.map(id =
      > PeriodicJob.getJobInstance(id, RunningStatus.Disabled)
      )

    val periodicJobs = (periodicJobsEnabled ++
      periodicJobsDisabled)

    for {
      ids1 <- jInstanceInsert ++= onetimeJobs
      ids2 <- jInstanceInsert ++= periodicJobs

      jobExecutions = ids1.map { id =>
        JobExecution(jobInstanceId = id, createTime = Instant.
          now(), status = JobExecutionStatus.Finished)
      }
      insertedJobExecutions <- jExecutionInsert ++=
        jobExecutions
    } yield {
      TestHelper.insertedJobExecutions = insertedJobExecutions.
        map { case (l, uuid) => (JobTestData.OnetimeJob.Owner.
        projectId, l, uuid) }
      TestHelper.insertedPeriodicJobs = ids2.zip(periodicJobs).
        toMap
    }
  }

}
```

2.19 utils/TestHelper.scala

```
package utils

import java.nio.file.Paths
import java.util.UUID

import com.mohiva.play.silhouette.api.Environment
import models.tables.{Crawler, JobInstance}
import play.api.libs.Files
import play.api.libs.Files.SingletonTemporaryFileCreator
import play.api.libs.json.{Json, OFormat}
import play.api.mvc.MultipartFormData
import play.api.mvc.MultipartFormData.FilePart
```

```
import play.api.test._
import utils.data.UserData._
import com.mohiva.play.silhouette.test._
import forms.OnetimeJobForm
import forms.periodic.{PeriodicJobChangeForm,
   PeriodicJobCreateForm}
import models.common.enums.JobPriority
import utils.data.JobTestData


object TestHelper {

  private val baseDir = "test/data/egg/"

  val projectWithCrawlers = 15

  var insertedJobExecutions: Seq[(Long, Long, UUID)] = Seq.
     empty
  var insertedPeriodicJobs: Map[Long, JobInstance] = Map.empty

  implicit val crawlerFormat: OFormat[Crawler] = Json.format[
     Crawler]

  val crawlers: Seq[Crawler] = Seq(
    Crawler(5, projectWithCrawlers, "toscrape-css"),
    Crawler(6, projectWithCrawlers, "toscrape-xpath")
  )

  val onetimeJobForm = OnetimeJobForm(
    crawlerId = crawlers.head.id,
    priority = JobPriority.Normal,
    settings = Map("DOWNLOAD_DELAY" -> "100")
  )

  val periodicJobForm = PeriodicJobCreateForm(
    title = "Periodic␣Job",
    crawlerId = crawlers.head.id,
    cronExpression = JobTestData.PeriodicJob.
       validCronExpression
  )

  val changePeriodicJobForm = PeriodicJobChangeForm(
    title = "Periodic␣Job",
    crawlerId = crawlers.head.id,
    cronExpression = JobTestData.PeriodicJob.
       validCronExpression
  )

  def requestWithMetadata(fileName: String)(implicit env:
```

```
     Environment[DefaultEnv]): FakeRequest[MultipartFormData[
     Files.TemporaryFile]] = {
    val filePart = FilePart(
      "eggFile",
      baseDir + fileName,
      Some("text/plain;charset=UTF-8"),
      SingletonTemporaryFileCreator.create(Paths.get(baseDir +
         fileName))
    )

    FakeRequest()
      .withAuthenticator(loginInfo)
      .withBody(MultipartFormData(
        dataParts = Map.empty,
        files = Seq(filePart),
        badParts = Nil))
  }
}
```

Лист регистрации изменений									
Изм.	Номера листов				Всего листов в документе	№ документа	Входящий № сопроводит. докум. и дата	Подпись	Дата
	измененных	замененных	новых	аннулированных					