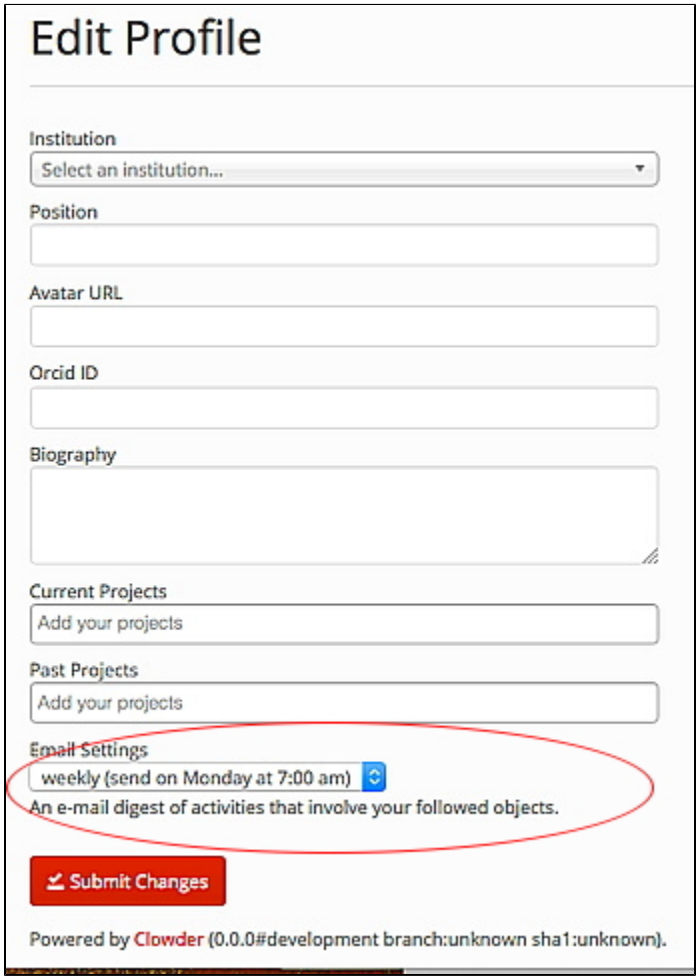
Timer job

Clowder supports scheduling of repetitive tasks by executing a job registered in MongoDB `jobs` Collection. The only implemented job as of August 2016 is **`EmailDigest`** triggered by setting an e-mail option in user's profile page as seen in Figure below. Selecting hourly, daily or monthly option in the pull down menu creates a job called `Digest('userId')` in the database which is then executed at pre-defined times, currently at the top of an hour (hourly), at 7:00 am (daily) or every Monday at 7:00 am (weekly).



Implementation

A `jobTimer` in the Clowder calls `JobsScheduler.runScheduledJobs()` every minute. This is done from `Akka.system().scheduler.schedule` in `app/Global`.
This time is then split into `minute`, `hour`, `day_of_week`, `day_of_month` variables in the `JobsScheduler` (`app/models/JobsScheduler.scala`) for further use in the code and for jobs' execution and maintenance.
The time variables are compared with a set of integers stored in every job object in the MongoDB Collection (`jobs`). In other words the time comparison is done by integer equality. A job gets fired hourly when `minute=minute`$_{job}$ if no other values of `hour`, `day_of_week`, `day_of_month` are defined. Similarly, the job gets executed with minutes and hours set (daily) when `minute=minute`$_{job}$ and `hour=hour`$_{job}$ (no other values of `day_of_week`, `day_of_month` are defined) etc. Note that there is no verification of time inserted in the system, nor there is time/date object comparison.

A job model in Clowder is called **`TimerJob`** (`app/models/TimerJob.scala`). A programmer can create different job schema but the `TimerJob` is sufficient for the most repetitive tasks.
A full time, or only subset of it is set in the `TimerJob` with `minute` (0-59), `hour` (0-23), `day_of_week` (1-7 for Monday-Sunday) and `day_of_month` (1-31) precision. An option `frequency` is meant to be 'hourly', 'daily', 'weekly', 'monthly' but can be any descriptive string. The `lastJobTime` field is useful for getting the time interval since the last job call (set by `scheduler.updateLastRun('jobName')`). Additionally, `parameters` can be used for any object id, `function` is a string describing action (e.g. `EmailDigest`).

Creating and calling a new timer job

Create and update job

1. A new `TimerJob` job is created and set in the MongoDB `Collection` by newly coded functions in `app/services/ScheduleService` and `app/services/mongoldb/MongoDBSchedulerService.scala` called for example `updateMyJob()`:

```
def updateMyJob(id: UUID, name: String, setting: String)
```

and

```
def updateMyJob(id: UUID, name: String, setting: String) = {
  if (jobExists(name) == false) {
      Jobs.insert(new TimerJob(name, None, None, None, None, Option('function'), Option(id), None, Option
(new Date())))
  }
  if (setting == "hourly"){
    updateJobTime(name, Option(0), None, None, Option(setting))
  }
  else if (setting == "daily"){
    updateJobTime(name, Option(0), Option(7), None, Option(setting))
  }
  else if (setting == "weekly"){
    updateJobTime(name, Option(0), Option(7), Option(1), Option(setting))
  }
  else {
    deleteJob(name)
  }
}
```

This is similar to a function `updateEmailJob()` already implemented in the Clowder where `updateJobTime(name, Option(0), Option(7), Option(1), Option(setting))` refers tot the time set to minute=0, hour=7 and day_of_week=1 (Monday) in the database as mentioned above. When Clowder time matches the values the job is returned from the database and Action is fired.
You can either change time directly in the code or pass the time values from a Play template as additional parameters such as:

```
def updateMyJob(id: UUID, name: String, setting: String, minute: Integer)= {
    updateJobTime(name, Option(minute), None, None, Option(setting))
}
```

2. from the Play `request` or directly in Scala on the server side call your job update:
`scheduler.updateMyJob(id, name, setting)`
here the `id` is a parameters id (`parameters: Option[UUID]`, see `TimerJob` model), `name` is the job's name and `settings` are used to distinguish time frequency (from options of pull down menus for example - hourly, weekly etc.)

Call and get job

1. Create function `getMyJobs()` in `app/models/JobsScheduler.scala` to get your job (`TimerJob`) at a certain time

```
def getMyJobs (minute: Integer, hour: Integer, day_of_week: Integer) = {
   var myJobs = scheduler.getJobByTime(minute, hour, day_of_week)
   myJobs
}
```

and register it with `runScheduledJobs()`

```
var myJobs = getMyJobs(minute.toInt, hour.toInt, day_of_week.toInt)
myAction.myActionJob(myJobs)
```

2. Create a new class `myAction.scala` for example in a package `models` (see `models/Event.scala` as an example)

```
package models

import java.util.Date
import services.SchedulerService
import services.DI

object myAction {
val scheduler: SchedulerService =  DI.injector.getInstance(classOf[SchedulerService])
val objects: ObjectService =  DI.injector.getInstance(classOf[ObjectService])
 /**
  * ‘Do something’ for each job returned by getMyJobs
  */
 def myActionJob(listJob: List[TimerJob]) = {
   for (job <- listJob){
     job.parameters match {
       case Some(id) => {
         objects.findById(id) match {
           case Some(object) => {
             job.lastJobTime match {
               case Some(date) => {
                 ‘Do something’
               }
               case None => Logger.debug("LastJobTime not found")
             }
           }
           case None => Logger.debug(“Object not found")
         }
         scheduler.updateLastRun(‘jobName’) //sets job’s name for example ”myJob[“ + id + "]")
       }
       case None => Logger.debug("Parameters not found")
     }
   }
 }
```

The `ObjectService` above, for example called `UserService` in the case of sending email digest and the user id was used as a parameter in the `TimerJob` and MongoDB job objects.

3. implement **`'Do something'`** in `myActionJob()` 🙂

Note

The `day_of_month` variable is part of the `TimerJob` model but it is not used in the `scheduler.getJobByTime()`
Adding it is straightforward:

- `app/services/SchedulerService.scala`

```
def getJobByTime(minute: Integer, hour: Integer, day_of_week: Integer, day_of_month: Integer): List
[TimerJob]
```

- `app/services/mongoldb/MongoDBSchedulerService.scala`

```
def getJobByTime(minute: Integer, hour: Integer, day_of_week: Integer, day_of_month: Integer): List
[TimerJob] ={
    val jobs = Jobs.find(
      $and(
        // either day_of_month exists AND the value is 'day_of_month' OR day_of_month does not exist
        $or($and("day_of_month" $exists true, MongoDBObject("day" -> day_of_month)), "day_of_month"
$exists false),
        // either day_of_week exists AND the value is 'day_of_week' OR day_of_week does not exist
        $or($and("day_of_week" $exists true, MongoDBObject("day_of_week" -> day_of_week)), "day_of_week"
$exists false),
        // either hour exists AND the value is 'hour' OR hour does not exist
        $or($and("hour" $exists true, MongoDBObject("hour" -> hour)), "hour" $exists false),
        // either minute exists AND the value is 'minute' OR minute does not exist
        $or($and("minute" $exists true, MongoDBObject("minute" -> minute)), "minute" $exists false)
      )
    )
```

```
def updateJobTime(name: String, minute: Option[Integer], hour: Option[Integer], day_of_week: Option
[Integer], day_of_month: Option[Integer], freq: Option[String]) = {
    if (minute == None){
      Jobs.dao.update(MongoDBObject("name" -> name), $unset("minute"))
    }
    else {
      Jobs.dao.update(MongoDBObject("name" -> name), $set("minute" -> minute))
    }

    if (hour == None){
      Jobs.dao.update(MongoDBObject("name" -> name), $unset("hour"))
    }
    else {
      Jobs.dao.update(MongoDBObject("name" -> name), $set("hour" -> hour))
    }

    if (day_of_week == None){
      Jobs.dao.update(MongoDBObject("name" -> name), $unset("day_of_week"))
    }
    else {
      Jobs.dao.update(MongoDBObject("name" -> name), $set("day_of_week" -> day_of_week))
    }

    if (day_of_month == None){
      Jobs.dao.update(MongoDBObject("name" -> name), $unset("day_of_month"))
    }
    else {
      Jobs.dao.update(MongoDBObject("name" -> name), $set("day_of_month" -> day_of_month))
    }

    Jobs.dao.update(MongoDBObject("name" -> name), $set("frequency" -> freq))
  }
```

- add extra parameter (`None`) to `updateEmailJob()`

Testing

1. Disable `scheduler.updateLastRun('jobName')` in `myAction.scala` (or in `models/Event.scala` for sending email digests) by commenting it out. Your events (followed objects for example) will become 'permanent' and the timer job will always execute since there is no update of the `lastJobTime` variable in the MongoDB job object. Don't forget to enable the `updateLastRun()` when you are done debugging.
2. Set the time variables in `getJobByTime() in app/services/mongoldb/MongoDBSchedulerService.scala` to those saved in the MongoDB job object. The job action will fire every minute since the integer equality is always true. For example for the pre-defined e-mail times (`minute=0`, `hour=7` and `day_of_week=1`) set `getJobByTime()`

```
def getJobByTime(minute: Integer, hour: Integer, day_of_week: Integer, day_of_month: Integer): List
[TimerJob] ={
    val jobs = Jobs.find(
      $and(
        $or($and("day_of_week" $exists true, MongoDBObject("day_of_week" -> 1)), "day_of_week" $exists
false),
        $or($and("hour" $exists true, MongoDBObject("hour" -> 7)), "hour" $exists false),
        $or($and("minute" $exists true, MongoDBObject("minute" -> 0)), "minute" $exists false)
      )
    )
```

Again, don't forget to reverse changes when you are done debugging.

3. **Testing e-mail digest**
    - Add functioning e-mail in `app/util/Mail.scala` if you use a 'fake' e-mail in your local Clowder developmental branch

```
def sendEmail(subject: String, user: Option[User], recipient: User, body: Html) {
        if (recipient.email.isDefined) {
            Logger.debug("Subject:" + subject + ", From:" + emailAddress(user) + ", Recipient: " +
emailAddress(Some(recipient)) + ", Body:")
                //sendEmail(subject, emailAddress(user), emailAddress(Some(recipient))::Nil, body)
                sendEmail(subject, emailAddress(user), List("yourEMail@illinois.edu"), body)
        }
}
```

    - Use functioning `smtp` in `securesocial.conf` or override it by setting `smtp.host` and (optional) `smtp.from` in your `custom.conf`

```
smtp.host=smtp.ncsa.illinois.edu
smtp.from=yourEMail@illinois.edu
```

Note that the host above can be used only within the NCSA's network.