JAVA ACADEMY - XIDERAL MONTERREY, N.L

WEEK 4
SPRING BATCH

Made by:
Luis Miguel Sánchez Flores

INTRODUCTION

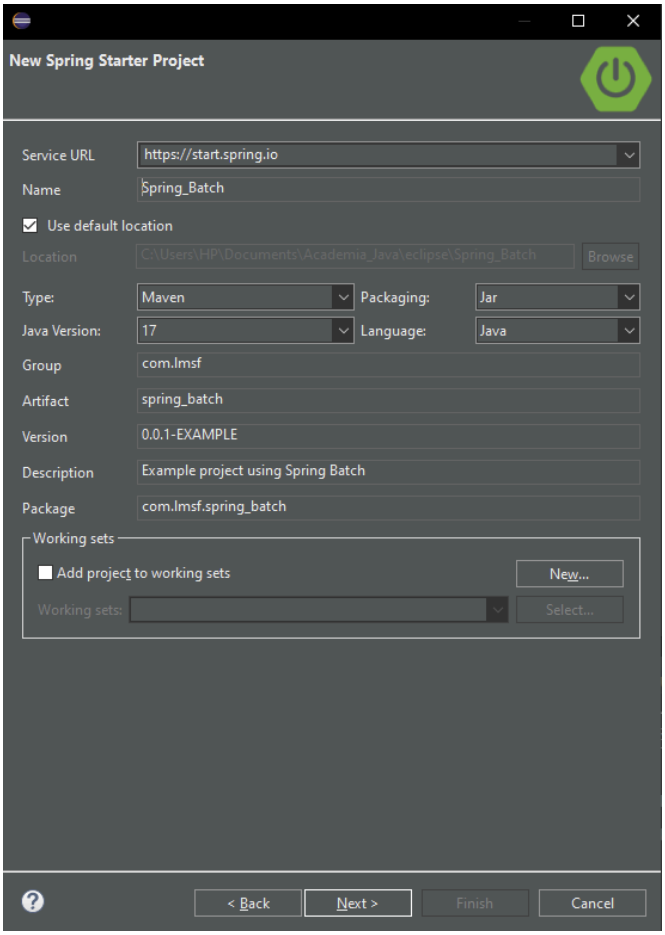
In an era where large amounts of data can be managed in a short period of time is becoming more and more critical, there is a growing demand from organizations for the development of software that is capable of processing large volumes of data in a scheduled manner. This is known as **batch processing,** where repetitive tasks are performed automatically by the computers organized by chunks, i.e, groups of data big enough to be processed by the computer in a timely fashion. Batch processing is usually done during off-peak hours (such as midnight) and when computer resources are available.

One of the more popular frameworks for working with batch processing in Java is **Spring Batch**, a Spring module that provides an extensive set of capabilities and designed to work with a persistent repository or in memory to store processing metadata which reports additional information on the activities executed.
Spring Batch provides built-in support for different types of data sources to work with such as databases, files and message queues, allowing developers to easily integrate existing systems and workflows. It also includes comprehensive error handling to ensure that batch jobs can be recovered from failures and improve the reliability of data processing tasks.

Spring Batch empowers developers to effectively manage large volumes of data while ensuring reliability and scalability, simplifying the development of batch processing workflows and optimizing performance, making it suitable for applications that require high throughput and efficiency.

Setting up Spring Batch

The Spring project was started through the **Spring Tool Suite plugin** available in the Eclipse IDE to quickly generate the project, filling in the project metadata and integrating the required dependencies, especially the Spring Batch dependency for developing a batch application towards automating and optimizing the data processing workflow:



With the project initialized, the `application.properties` file was configured to set up the connection with the MySQL database for this Spring Batch project. A special parameter is specified in the url that instructs the JDBC driver to automatically create the school_batch database in case it does

not yet exist at the time of establishing the connection, which is useful for setting up the new database quickly for the test:

```
# MySQL datasource
# Create 'school_batch' if it doesnt exists already:
spring.datasource.url=jdbc:mysql://localhost:3306/school_batch?createDatabaseIfNotExist=true
spring.datasource.username=springstudent
spring.datasource.password=springstudent
server.port=8101
```

Spring will automatically create a `students_batch` table through the `hibernate.ddl-auto` property, based on the following `Student` entity, which declares the attributes / columns. Lombok is applied to generate the constructor as well as the getters and setters for each attribute:

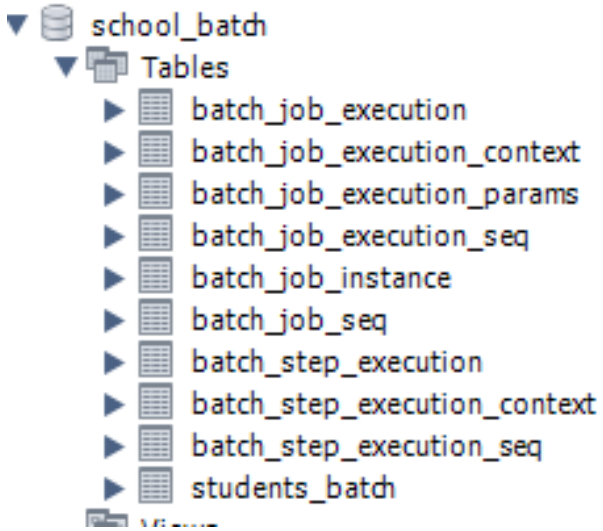
```
15 //Entity
16
17 @Entity
18 @Table(name="students_batch")
19 @Data
20 @AllArgsConstructor
21 @NoArgsConstructor
22 public class Student {
23
```

```
// Attributes / Columns to be generated
@Id
@Column(name="id")
private int id;

@Column(name="first_name", nullable=false)
private String firstName;

@Column(name="last_name", nullable =false)
private String lastName;

@Column(name="date_of_birth", nullable = false)
private String dateOfBirth;

@Column(name="email")
private String email;


@Column(name="gender")
@Enumerated(value = EnumType.STRING)
private Gender gender;

@Column(name="phone_number")
private String phoneNumber;
```

Lastly, the `spring.batch.jdbc.initialize-schema` y `spring.batch.job.enabled` properties exclusive to Spring Batch are established to define if the metadata tables that provide additional information about the jobs and steps that are executed in the application should be initialized automatically, as well as determine if the jobs should be executed when the application is started:

```
16 # Inicialize the Spring Batch metadata tables (job executions, step executions, etc.)
17 spring.batch.jdbc.initialize-schema=ALWAYS
18 # Avoid running the jobs on startup
19 spring.batch.job.enabled=false
```

Running the application with the previous properties will generate the `school_batch` database, along with the `students_batch` table and metadata tables generated by the Spring Batch project, as shown below:



For this Spring Batch example, **a job will be carried out to recover the list of students who do not have any contact information**, that is, those who do not have an email address and a telephone number.

For that, a `students.csv` file was created that contains a thousand rows of student data, including those who don't have either email or phone number, which will be processed by a **step** component:

```
id,firstName,lastName,dateOfBirth,email,gender,phoneNumber
1,Reinaldos,Holwell,15-03-2006,,Male,1434501961
2,Arte,Randerson,24-10-2003,,Male,3017664707
3,Faber,Adolfsen,25-06-2001,fadolfsen2@washingtonpost.com,Male,7256100676
4,Bonnibelle,Spirit,31-07-2004,bspirit3@seesaa.net,Female,4548592154
5,Jaymee,Kosel,01-11-2001,jkosel4@springer.com,Female,
6,Bernie,Buss,29-06-2004,bbuss5@thetimes.co.uk,Female,1945112281
7,Cecil,Knoton,16-10-2003,cknoton6@mozilla.com,Male,1859840173
8,Wilfred,Arnott,14-03-2003,warnott7@noaa.gov,Male,
9,Saba,Fidge,21-07-2002,sfidge8@washingtonpost.com,Female,3484069521
10,Larine,Siman,06-03-2004,lsiman9@state.tx.us,Female,1035859358
11,Paolo,Cowin,19-12-2001,pcowina@ucoz.com,Male,8775764093
12,Louis,Reuter,30-04-2004,,Male,4799309393
13,Georgianne,Squier,10-01-2006,gsquierc@abc.net.au,Female,2335168048
14,Vivian,Extill,19-09-2002,,Female,4106711452
15,Hatti,Masey,21-10-2004,hmaseye@slideshare.net,Female,6718967738
16,Friederike,Corris,22-01-2003,fcorrisf@squidoo.com,Female,6545666663
17,Hew,Upton,20-01-2001,huptong@census.gov,Male,5067847757
18,Elfrieda,Coppeard,10-07-2003,ecoppeardh@harvard.edu,Female,9312189047
19,Ram,Adamczyk,28-06-2005,radamczyki@nytimes.com,Male,5817682609
20,Drake,Ridgewell,26-01-2004,dridgewellj@altervista.org,Male,6183924661
21,Burnaby,Croyden,31-05-2002,,Male,
22,Ingeborg,Bingley,16-09-2005,ibingleyl@fema.gov,Female,
23,Kirsten,Skillen,27-03-2006,kskillenm@topsy.com,Female,
24,Stillman,Drakes,28-06-2001,sdrakesn@cdc.gov,Male,
25,Austen,Jerg,04-03-2005,ajergo@patch.com,Male,9183270593
26,Florian,Yansons,30-01-2003,fyansonsp@squarespace.com,Other,
27,Raddie,Cavell,23-07-2006,rcavellq@vimeo.com,Male,2765255970
28,Tiler,Bottinelli,27-11-2003,tbottinellir@symantec.com,Male,2546340940
29,Lexi,Godsell,30-09-2002,,Other,
30,Brandie,Ambrosch,05-10-2004,bambroscht@blinklist.com,Female,4178579991
31,Rosalinde,Nicklen,04-01-2001,rnicklenu@skype.com,Other,2885487292
32,Doralin,Temblett,18-09-2004,dtemblettv@liveinternet.ru,Female,1646238084
33,Stevana,Binton,08-04-2003,,Female,9666659370
34,Isidor,Chawner,19-02-2005,ichawnerx@fastcompany.com,Male,
```

Along with the `Student` entity shown before, a `StudentRepository` interface was made that extends the `JpaRepository` interface to execute

basic CRUD operations such as the `save()` function, which will be used by the Spring Batch project to gather the students that do not contain contact info.

```java
package com.lmsf.spring_batch.repository;

import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<Student, Integer> {

}
```

A `StudentController` was developed that represents the REST controller of the application, and in which an API endpoint is provided to trigger the Spring Batch job. It is here that a `JobLauncher` (responsible for starting up the jobs with specific parameters) and a `Job` (set of steps that define the work to be done) are injected into the Controller by Spring, and will be executed through a POST request at the `students-without-phone-number` endpoint. Job parameters can be used to customize the job behavior at runtime or, in this case, identify the job instance executed.

```java
@RestController
@RequestMapping("/students/jobs")
public class StudentController {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job job;



    @PostMapping("/students-without-phone-number")
    public void getStudentsWithoutPhoneNumber() throws Exception  {
        JobParameters params = new JobParametersBuilder()
                        .addLocalDateTime("startedAt", LocalDateTime.now())
                        .toJobParameters();

        try {
            jobLauncher.run(job, params);
        } catch (JobExecutionAlreadyRunningException | JobRestartException | JobInstanceAlreadyCompleteException |
            e.printStackTrace();
        }
    }
```

Job Configuration

The magic of this Spring Batch project lies within the `StudentBatchConfig` class in which, as the name implies, the job and step process are configured in order to read and process the `students.csv` file to then add the students without contact information onto the created database. The class has an `@Configuration` annotation that indicates Spring of bean definitions for the Spring Batch components.

The job is defined by the `runJob()` method, where a `JobRepository` is passed for basic CRUD operations of persistent domain objects within the Spring Batch project, as well as the first (and only) `Step` object that will take care of the task. The job is created through a `JobBuilder` class to establish the workflow:

```
@Bean
public Job runJob(JobRepository jobRepository, Step step1) {
    return new JobBuilder("runJob", jobRepository)
            .flow(step1)
            .end()
            .build();
}
```

The step itself is implemented through the `firstStep()` method, which leverages the `StepBuilder` class to construct the steps with its key components in order to successfully perform the task. Such components are:

- **ItemReader:** Used for reading data from sources such as files, databases or other web services. In this case, a `FlatFileItemReader` is provided that reads each row of the

`students_data.csv` file and pass to the **ItemProcessor:**

```java
@Bean
public FlatFileItemReader<Student> reader() {

    // Generate a new FlatFileItemReader to read from:
    return new FlatFileItemReaderBuilder<Student>()
                .name("studentDataReader") // Name of the Reader instance
                .resource(new ClassPathResource("students_data.csv")) // Resource to read from
                .linesToSkip(1) // Skip the first line, as it contains the headers
                .delimited() // returns a DelimitedBuilder to break down the row in indv. fields
                .strict(false)
                .names("id", "firstName", "lastName", "dateOfBirth", "email", "gender", "phoneNumber") // Name of each field
                .fieldSetMapper(new BeanWrapperFieldSetMapper<Student>()) // Bridge between the extracted data and the Java do
                .targetType(Student.class) // Converts the set of fields into a Student object
                .build();
}
```

- **ItemProcessor:** Process each of the items / rows passed by the ItemReader and make the necessary changes. It is here where the students are validated if they contain either a phone number or email to contact. If that's not the case, they're passed to the **ItemWriter:**

```java
@FunctionalInterface
public interface StudentProcessor extends ItemProcessor<Student, Student> {


    Student process(Student student) throws Exception;
}
```

```java
.processor((StudentProcessor)
        student -> {
            // If the student has no contact info, add it to the table...
            if(student.getPhoneNumber().isEmpty() && student.getEmail().isEmpty()) {
                student.setEmail(null);
                student.setPhoneNumber(null);
                return student;
            }
            // If there's at least a contact info, ignore it...
            return null;
        }
)
```

- **ItemWriter:** Writes the processed items to a target location (the MySQL database). Here a `RepositoryItemWriter` is used to set the JPA repository as the destination, and in which it's `save()` function is applied to store the validated student:

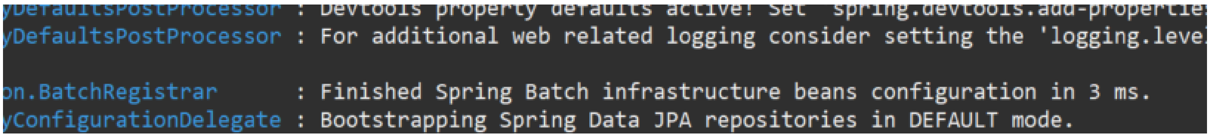
```java
@Bean
public RepositoryItemWriter<Student> writer() {
    return new RepositoryItemWriterBuilder<Student>()
                // Set the JPA repository as the target location
                .repository(studentRepository)
                // Call it's save function to write the student data onto the database
                .methodName("save")
                .build();
}
```

A `TransactionManager` is passed to the `Step` object to ensure that the operations within it are executed atomically

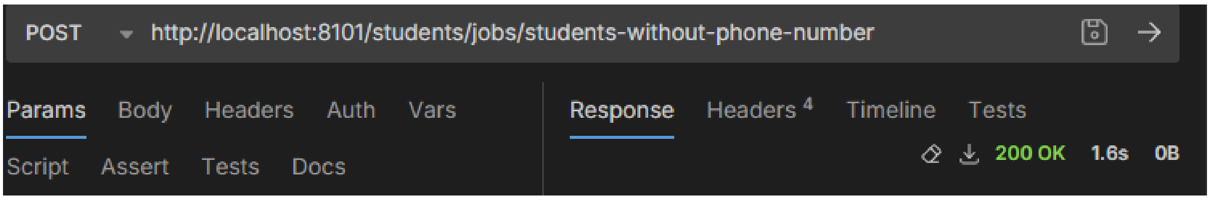
```java
@Bean
public Step firstStep(JobRepository jobRepository, PlatformTransactionManager transactionManager,
        RepositoryItemWriter<Student> writer
        ) {
    return new StepBuilder("firstStep", jobRepository)
            .<Student, Student>chunk(20, transactionManager)
            .reader(reader())
            .processor((StudentProcessor)
                    student -> {
                        // If the student has no contact info, add it to the table...
                        if(student.getPhoneNumber().isEmpty() && student.getEmail().isEmpty()) {
                            student.setEmail(null);
                            student.setPhoneNumber(null);
                            return student;
                        }
                        // If there's at least a contact info, ignore it...
                        return null;
                    }
            )
            .writer(writer)
            .build();
}
```

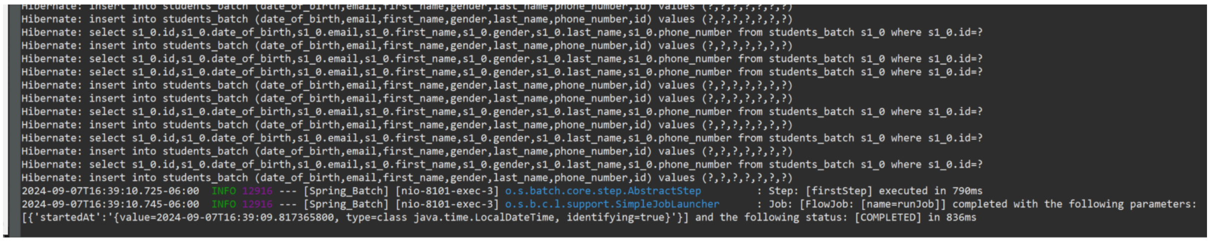
Executing the Job

Once finished with the configuration of the Job and its necessary Steps, the application can be run to execute the Job through the API endpoint established previously. Running the application will prompt Spring to automatically generate the necessary Batch components to carry out the tasks successfully:



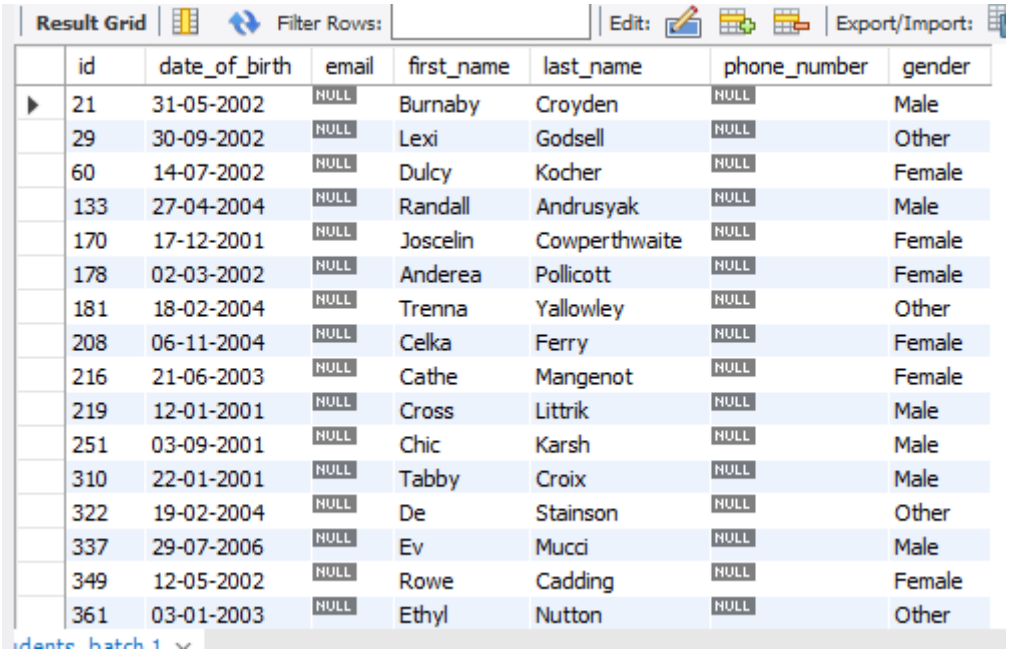
Through the Bruno program, a POST request is sent to the `students-without-phone-number` endpoint of the REST application to execute the Job and obtain the list of students without any contact information, yielding the following result:





The console outputs that both the Step and the Job were able to execute with success, checking out each of the `students_data.csv` rows in groups / chunks of 20 while only adding those onto the database that meet the criteria of having empty values for both the `email` and `phone_number` fields.

Reviewing the table in MySQL Workbench showcases that the Job was able to successfully filter the students without a contact info:

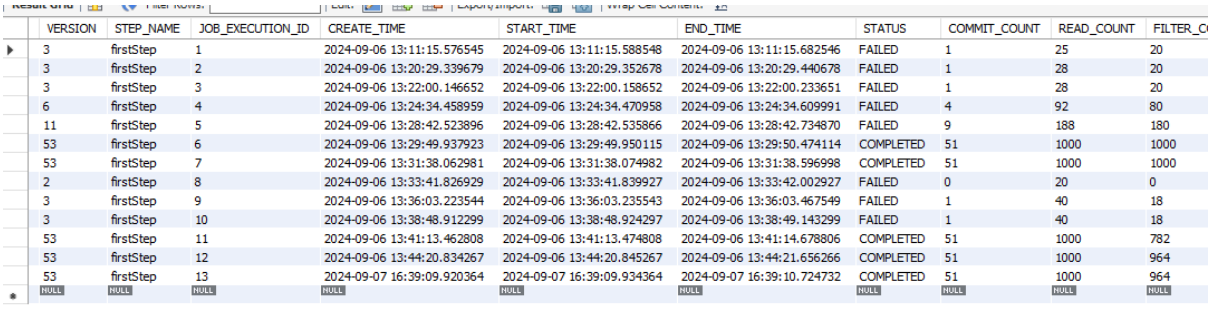


Result Grid | Filter Rows: | Edit: | Export/Import:

id	date_of_birth	email	first_name	last_name	phone_number	gender
21	31-05-2002	NULL	Burnaby	Croyden	NULL	Male
29	30-09-2002	NULL	Lexi	Godsell	NULL	Other
60	14-07-2002	NULL	Dulcy	Kocher	NULL	Female
133	27-04-2004	NULL	Randall	Andrusyak	NULL	Male
170	17-12-2001	NULL	Joscelin	Cowperthwaite	NULL	Female
178	02-03-2002	NULL	Anderea	Pollicott	NULL	Female
181	18-02-2004	NULL	Trenna	Yallowley	NULL	Other
208	06-11-2004	NULL	Celka	Ferry	NULL	Female
216	21-06-2003	NULL	Cathe	Mangenot	NULL	Female
219	12-01-2001	NULL	Cross	Littrik	NULL	Male
251	03-09-2001	NULL	Chic	Karsh	NULL	Male
310	22-01-2001	NULL	Tabby	Croix	NULL	Male
322	19-02-2004	NULL	De	Stainson	NULL	Other
337	29-07-2006	NULL	Ev	Mucci	NULL	Male
349	12-05-2002	NULL	Rowe	Cadding	NULL	Female
361	03-01-2003	NULL	Ethyl	Nutton	NULL	Other

Spring Batch also introduces data onto the generated metadata tables to keep track of the executed jobs and provide metrics like the duration, processed rows and status for both a particular job or step, helping developers to evaluate the performance of the jobs and steps and perform the necessary changes for a more effective batch processing.

The following table showcases the data added to the `batch_step_execution`, demonstrating its name, start and end time, the number of rows, read, committed and filtered, etc:



VERSION	STEP_NAME	JOB_EXECUTION_ID	CREATE_TIME	START_TIME	END_TIME	STATUS	COMMIT_COUNT	READ_COUNT	FILTER_C
3	firstStep	1	2024-09-06 13:11:15.576545	2024-09-06 13:11:15.588548	2024-09-06 13:11:15.682546	FAILED	1	25	20
3	firstStep	2	2024-09-06 13:20:29.339679	2024-09-06 13:20:29.352678	2024-09-06 13:20:29.440678	FAILED	1	28	20
3	firstStep	3	2024-09-06 13:22:00.146652	2024-09-06 13:22:00.158652	2024-09-06 13:22:00.233651	FAILED	1	28	20
6	firstStep	4	2024-09-06 13:24:34.458959	2024-09-06 13:24:34.470958	2024-09-06 13:24:34.609991	FAILED	4	92	80
11	firstStep	5	2024-09-06 13:28:42.523896	2024-09-06 13:28:42.535866	2024-09-06 13:28:42.734870	FAILED	9	188	180
53	firstStep	6	2024-09-06 13:29:49.937923	2024-09-06 13:29:49.950115	2024-09-06 13:29:50.474114	COMPLETED	51	1000	1000
53	firstStep	7	2024-09-06 13:31:38.062981	2024-09-06 13:31:38.074982	2024-09-06 13:31:38.596998	COMPLETED	51	1000	1000
2	firstStep	8	2024-09-06 13:33:41.826929	2024-09-06 13:33:41.839927	2024-09-06 13:33:42.002927	FAILED	0	20	0
3	firstStep	9	2024-09-06 13:36:03.223544	2024-09-06 13:36:03.235543	2024-09-06 13:36:03.467549	FAILED	1	40	18
3	firstStep	10	2024-09-06 13:38:48.912299	2024-09-06 13:38:48.924297	2024-09-06 13:38:49.143299	FAILED	1	40	18
53	firstStep	11	2024-09-06 13:41:13.462808	2024-09-06 13:41:13.474808	2024-09-06 13:41:14.678806	COMPLETED	51	1000	782
53	firstStep	12	2024-09-06 13:44:20.834267	2024-09-06 13:44:20.845267	2024-09-06 13:44:21.656266	COMPLETED	51	1000	964
53	firstStep	13	2024-09-07 16:39:09.920364	2024-09-07 16:39:09.934364	2024-09-07 16:39:10.724732	COMPLETED	51	1000	964