

TECHNISCHE UNIVERSITÄT BERLIN

INTERNET OF SERVICES

---

## **GroupBy**

---

*Authors:*

**Bram LEENDERS** 367999

**Georgi KRASTEV** 366576

**Juan ROCCA** 367991

**Marc ROMEYN** 368036

August 12, 2015

## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Current solutions</b>	<b>3</b>
<b>3</b>	<b>Requirements</b>	<b>4</b>
<b>4</b>	<b>Algorithm</b>	<b>5</b>
4.1	Tabu Search . . . . .	5
4.2	Tabu criteria . . . . .	5
4.3	Objective Function . . . . .	5
4.4	Components . . . . .	6
4.5	Moves . . . . .	7
4.6	Models . . . . .	9
<b>5</b>	<b>Implementation</b>	<b>11</b>
5.1	High-level architecture . . . . .	11
5.2	Server-side . . . . .	12
5.3	Algorithm-side . . . . .	13
5.4	Hapi on NodeJS vs. Ruby on Rails . . . . .	13
<b>6</b>	<b>Deployment</b>	<b>14</b>
6.1	Starting Docker . . . . .	14
6.2	Stopping Docker . . . . .	15
6.3	Deploying new versions . . . . .	15
6.4	Access to the VMs . . . . .	16
6.5	Docker networking . . . . .	17
<b>7</b>	<b>Continuous Integration</b>	<b>17</b>
<b>8</b>	<b>Result Evaluation</b>	<b>18</b>
<b>9</b>	<b>Group Work Evaluation</b>	<b>18</b>
<b>10</b>	<b>Future Work</b>	<b>19</b>

# 1 INTRODUCTION

The objective of this project is to solve the following problem statement:

How can one distribute a set of students over a set of project groups in an optimal way?

Teachers face this question every time a new semester start, when they have to distribute the newly enrolled students over project groups. Our aim is to make this easier for the teachers by providing an open and simple to use platform for distributing students over groups.

We can break down this main problem into various sub-problems. Two fundamental sub-problems we already address here, and reference to in later sections. These are:

- What is an optimal distribution?
- How do we reach this distribution?

This first question is non-technical, but pedagogical. This is not our area of expertise, so we used literary study to find an answer to this question.

The second question is more in the machine learning area. This problem is in general an NP-complete problem (as shown by E. Ronn [4]): brute-forcing an optimal solution would involve evaluating all possible assignments, which makes it unfeasible even for small problem sizes. To illustrate: even for medium sized courses (like Internet of Services) with an estimated 30 students and 6 equally sized groups, there are approximately  $5 \cdot 10^{26}$  possible distributions<sup>1</sup>. For a computer that can test a billion combinations per second, it would take over 16 billion years to test these combinations. This is just a back-of-the-envelope calculation, but it illustrates that we need a more efficient way to search.

We take a heuristic approach by trying to optimize an objective function with taboo search. The resulting solution might not be globally optimal, but it should be close to optimal and at least better than a random assignment.

Our objective is to design and implement a matching algorithm for assigning students to project groups based on various constraints and optimization criteria, e.g. even distribution of skills, group diversity, student preferences for groups and friends/preferred partners.

The design of our algorithm is based on and similar to the solution proposed by Hübscher [3] with some modifications and extensions. In the referenced paper, he describes both the pedagogical and technical aspect of how to distribute students over groups.

# 2 CURRENT SOLUTIONS

This project was created to replace the current solution of first come first serve (FCFS) approach being handled in the Internet of Services course. This solution was not optimal since it is a mandatory course for a select number of students and an elective for the rest. So if the mandatory students were not the first to sign up they were forced into a group which may not

---

<sup>1</sup>If we start assigning first for group 1 place 1, then group 1 place 2, etc., then group 2 place 1, etc., we have  $30!$  possibilities, but many are doubles. To be precise, there are  $6!$  ways to order groups (all cause doubles), and per group  $5!$  ways to order the students in the group (all cause doubles). That means that there are  $\frac{30!}{6! \cdot 6 \cdot 5!} = 5 \cdot 10^{26}$  ways to distribute 30 students over 6 groups (excluding doubles).

suit their skills. Since there is a number of groups with a different range of skills requirements and a large number of students each with a unique skill set, FCFS could end up matching students to groups where they do not meet the requirements.

Also, giving mandatory students first choice would give them an unfair advantage over other students and might put non-mandatory students in suboptimal groups.

Seeing these obstacles and experiencing them ourselves, we wanted to find a solution that takes into account the student's group preferences and the skill set requirements of the group to determine an optimal solution without excluding any of the mandatory students. Our approach was to create a tool that integrates with ISIS for a smooth application process.

### 3 REQUIREMENTS

The following list documents requirements we had during the implementation of the project:

- The distribution algorithm must be transparent (e.g. open source).
- Students that mandatorily follow the course must be assigned to a group.
- Except for the previous requirement, mandatory students and non-mandatory students are treated equally. Mandatory students do not get first pick.
- The resulting distribution is stable, meaning that no two students want to switch groups<sup>2</sup>.
- Teachers can register new courses.
- Teachers can set the following settings for the course:
  - A name, semester
  - The groups in the course
  - Important skills for this course
- Teachers can start a distribution. Optionally, they can then change parameters and re-start the distribution.
- Students can enroll for courses.
- Students can set the following parameters for the algorithm:
  - Their friends, choosing from a list of enrolled students.
  - How much they like each group, on a scale of 1 to 10.
  - What they find more important: topics (groups) or friends. This is be on a 1 to 10 scale, not binary.

---

<sup>2</sup>This requirement has been adapted: the supervisor can set global weights in such a way that this constraint is violated. Instead, the distribution is stable in the sense that no single switch results in a higher overall value of the objective function. Only if the supervisor does not change the default settings, we can still guarantee the original constraint of stable pairs.

- Authentication is be done through ISIS, so all and only TU Berlin students can enroll.
- The information must be confidential: only the student and possibly the supervisor can see a student's settings

## 4 ALGORITHM

### 4.1 TABU SEARCH

Tabu search is a pluggable heuristic search algorithm that can be successfully applied as a basis for many domain-specific optimization problems [1, 2, 3]. Since this is a heuristic approach, it can not guarantee that a globally optimal solution will always be found, but it will always find at least a local optimum of the objective function.

The basic idea of the search routine is that it climbs towards a local maximum and keeps a limited number of traversed states as history, in order to reduce the memory requirements of the program. The history is then used in a domain-specific way to prohibit (make tabu) similar states to those already visited, in order to keep exploring the search space instead of revisiting old solutions. Also, aspiration criteria have to be defined for lifting this restriction whenever it would prevent the algorithm from finding a better solution. The last component of tabu search is to define transitions (moves) between different states in the domain and an objective function to evaluate those states, that will be maximized by the algorithm.

### 4.2 TABU CRITERIA

We considered two options for the tabu criteria. The first one is to keep timestamps for every student that track the latest movement. Then moving a student would be prohibited for the next  $n$  steps. This is the strategy used by R. Hübscher [3]. However, this has have a prohibitive effect on dropping and filling groups (see moves below), because these moves involve a whole set of students. Thus, if even one of these students had been moved in any of the previous  $n$  steps, the dropping would be an illegal move.

The other option which we use in the final implementation, is to build a bounded tabu queue that contains the latest  $n$  moves and prohibits similar moves, e.g. swapping the same 2 students (for a full list of moves, see section 4.5).

### 4.3 OBJECTIVE FUNCTION

This section explains in more detail the composition of the objective function that the matching algorithm tries to maximize.

First, let us define the following notations:

$I_G$  : Index of group IDs

$I_S$  : Index of student IDs

$G = \{G_i | i \in I_G\}$  : Set of groups

$S = \{S_i | i \in I_S\}$  : Set of students

$K^*$  : Set of all skills

$K_g \subseteq K^*$  : Set of global skills (per course)

$K = \{K_i | i \in IG, K_i \subseteq K^*\}$  : Set of skills per group  
 $M = \{M_{sk} | s \in I_S, k \in K^*\}$  : Skill matrix (sparse)  
 $P = \{P_{sg} | s \in I_S, g \in I_G\}$  : Preference matrix (sparse)

#### 4.4 COMPONENTS

So far we have the following components to the objective function out of the box. More can be added later on. Every component is calculated separately over the current assignment, and then the values are combined into a total score, by multiplying them with the global weights provided in the algorithm configuration. A global weight of 0 will turn off the respective criterion altogether.

Note that some criteria are global (i.e. calculated over the whole assignment, e.g. distribution in a diverse way) whereas others are local (i.e. calculated per student, e.g. friends of a student). The objective function is well defined only when a single global criterion is used. On the other hand, local criteria allow for specifying fine grained per student weights to improve the performance of the solver.

##### 4.4.1 MAXIMALLY DIVERSE (GLOBAL)

Optimizes for groups with maximal diversity with respect to available skills. The assumption here is that knowledge sharing between peers improves with increasing difference in experience and expertise across subjects. Note that this is a global criteria. Only one of those should be used at a time.

##### 4.4.2 EVENLY SKILLED (GLOBAL)

Optimizes for groups with evenly distributed skills. The assumption here is, that in order to maximize the learning objective, all relevant skills in the groups need to be well represented. In this way, students are likely to pick up the skills that the course was designed to convey. Note that this is a global criteria. Only one of those should be used at a time.

##### 4.4.3 FRIENDS AND FOES (LOCAL)

We also want to allow students to specify friends, with which they would like to be in the same group. For completeness, we show the formula that works for friends and foes, since the algorithm supports it, but the frontend does not expose this function.

$$f(s_1, s_2) = \begin{cases} -1 & \text{if } s_2 \text{ is a foe of } s_1 \\ 0 & \text{if } s_2 \text{ is neither friend nor foe of } s_1 \\ 1 & \text{if } s_2 \text{ is a friend of } s_1 \end{cases}$$

$$friendsFoes = w \cdot \sum_{i \in I_G} \sum_{s_1 \in G_i} \sum_{s_2 \in G_i} f(s_1, s_2)$$

where  $w$  is the local weight of student  $s_1$  on this criterion.

#### 4.4.4 GROUP PREFERENCES (LOCAL)

One of the classic optimization criteria for solving the assignment problem is letting students express their preferences for a group. In this implementation, we also consider this factor, but it's not a strict requirement to have stable pairs. Also, we opted for a weighted map of preferences instead of an ordered list for more flexibility. In this setup, stable pairs don't make sense any more. Note that this is a local criterion, so per student weights are supported.

$$groupPrefs = w \cdot \sum_{i \in I_G} \sum_{s \in G_i} P_{si}$$

where  $w$  is the local weight of student  $s$  on this criterion.

#### 4.5 MOVES

In our implementation of the tabu search algorithm, we allow the following moves (transitions) between valid student-group assignments:

- **Swap:** swaps two students.
- **Switch:** move a student to another group.
- **Drop:** remove a group.
- **Fill:** put a group of students from the waitinglist in a group.

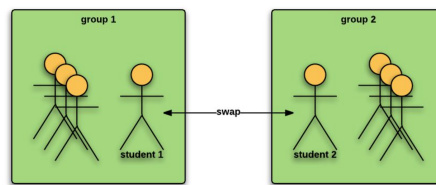
We will now explain these moves in more detail:

##### 4.5.1 SWAP

Function signature:

```
swap(s1: Student, s2: Student)
```

Figure 4.1: Swapping two students



Swap the (current) groups of two students; if student1 is in group A, and student2 in group B, switching them places student1 in B and student2 in A.

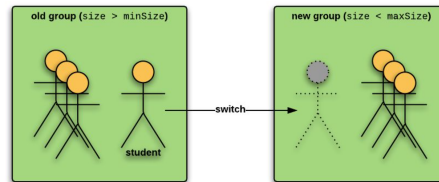
It is also a valid move to swap a student in a group with a student in the waiting list. However, this gives an extra requirement: in this case the course must not be mandatory for the student going to the waiting list. This constraint helps ensure that we never end up with a distribution that has mandatory students in the waiting list.

#### 4.5.2 SWITCH

Function signature:

```
switch(s: Student, g: Group)
```

Figure 4.2: Switching a student



Move a student from his current group to another one. After the move has been performed, the minimal and maximal group size requirements of both the old and the new group have to be fulfilled.

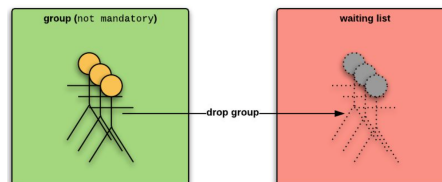
It is also valid to switch a non-mandatory student to the waiting list and a student from the waiting list to a regular group, but again with the restriction that the course must not be mandatory for the student. Again: this constraint helps ensure that we never end up with a distribution that has mandatory students in the waiting list.

#### 4.5.3 DROP GROUP

Function signature:

```
dropGroup(g: Group)
```

Figure 4.3: Dropping a group



Drop a group from the course and move all students currently assigned to it to the waiting list. This move is only valid if the group to be dropped is not mandatory for the course.

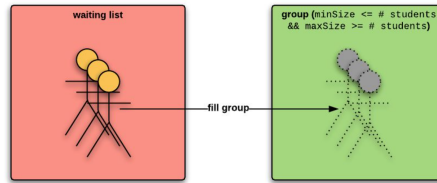
#### 4.5.4 FILL GROUP

Function signature:

```
fillGroup(g: Group, ss: Set[Student])
```



Figure 4.4: Filling a group



Fill an empty group with a set of students from the waiting list. The number of students should be equal or greater than the minimum size and equal or less than the maximum size of the group.

It is comparable to repeated switches from the waitinglist to a group, but allows a previously dropped groups to exist again. There is one important difference: if the previously empty group has a minimum size, this method allows the group to be filled again without violating the minimum size constraint; switching checks for the minimum size constraint on every move, so a group with minimum size 5 would never be filled (since it requires 5 moves, but the first 4 are illegal).

## 4.6 MODELS

In this section we describe the data schema of the algorithm. Most of the input is immutable as the actual assignment is based only on references. The data has a hierarchical structure. Each instance of the algorithm deals with one course at a time, and each course is a separate run of the solver.

### 4.6.1 COURSE

---

```
1 type Course {
2     jobId:      Long,
3     settings:   Settings, // See the settings model
4     endpoints: Endpoints, // See the Endpoints model
5     students:  List[Student], // The students in this course
6     groups:    List[Group], // The groups in this course
7     skills:    Set[String], // List of skills, optional
8     weights:   {
9         maximallyDiverse: Float,
10        evenlySkilled: Float,
11        friendsAndFoes: Float,
12        groupsPreferences: Float
13    }
14 }
```

---

#### 4.6.2 SETTINGS

The supervisor can specify settings to affect the behaviour of the algorithm. These are stored in the following model:

---

```
1 type Settings {
2   // number of iterations without improvement before termination
3   iterations: Int,
4   // Number of initial positions the algorithm tries
5   initialMoves: Int,
6   // Number of starting points
7   startingPoints: Int
8   // size of the tabu list / queue
9   tabuSize: Int,
10  // should the algorithm optimize for diversity or equality
11  diverse: Boolean
12 }
```

---

#### 4.6.3 ENDPOINTS

The Rails server and Scala algorithm communicate asynchronously. The Rails server submits a job, and after some time the algorithm will post the results back to a special endpoint on the Rails server. This datastructure contains the URLs where the Rails server expects to receive the results.

---

```
1 type Endpoints {
2   success: String, // URL to success endpoint on Rails server
3   failure: String  // URL to failure endpoint on Rails server
4 }
```

---

#### 4.6.4 STUDENT

---

```
1 type Student{
2   id: Long
3   name: String           // Optional, default empty
4   mandatory: Boolean     // Optional, default false
5   skills: Map[String => Float], // Optional, default empty
6   weights: {
7     friendsAndFoes: Float, // Optional, default 0.5
8     groupPreferences: Float // Optional, default 0.5
9   },
```

```

10 // Maps groups to how much a student likes it (stored as ints
    ↳ referencing group ids):
11 preferences: Map{Group => Float},
12 // Optional (uses ints to store student ids):
13 friends: List[Student],
14 // Optional, unused in the current version (stored as ints
    ↳ referencing student ids):
15 foes: List[Student]

```

---

#### 4.6.5 GROUP

```

1 type Group {
2   id: Long,
3   minSize: Int,
4   maxSize: Int,
5   mandatory: Boolean, // Optional, default false
6   name: String,       // Optional
7   skills: Set[String] // Optional
8 }

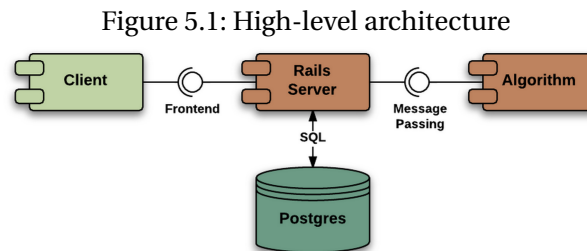
```

---

## 5 IMPLEMENTATION

### 5.1 HIGH-LEVEL ARCHITECTURE

The following image shows a UML diagram of the components:



Our application has two main parts: a Rails server that processes and stores the input from the client(s). When a supervisor (client) requests a distribution to be made, the Rails server sends the data to the Algorithm server that computes a distribution and returns it to the Rails server. The Rails server stores the distribution and shows it to the clients.

This architecture allows us to have a clear separation between the simple CRUD (create, read, update, delete) application running on Rails, and the more complicated business logic in the distribution algorithm.

## 5.2 SERVER-SIDE

### 5.2.1 RUBY ON RAILS

Ruby on Rails<sup>3</sup> (RoR) is the framework we chose for our webserver. It is a model-view-controller framework to use the Ruby programming language for displaying a site. This makes it similar to Django for Python and Spring for Java.

It was decided to use RoR because this community is very active at the time of writing. This community helps with documentation, questions and plug-ins (called gems). These plug-ins will be explained later on, and for us were a large pro for RoR, because things like authentication can be easily implemented with this. Because of the active community there are a lot of plugins available. Furthermore, it has a clear Model View Controller structure making it easy to write structured code, which helps while doing teamwork.

Furthermore, for the views we are using a language called HAML. This language is a mixture of HTML and Ruby code. It has the advantage that it works with indentations in order to define blocks does not need to be written so that an open and close tag, such as in normal HTML does have to. An example is:

#### HTML

---

```
1 <div>
2   <p>
3     foo
4   </p>
5   <p>
6     bar
7   </p>
8 </div>
```

---

#### HAML

---

```
1 %div
2   %p foo
3   %p bar
```

---

### 5.2.2 PHUSION PASSENGER

It is a web and application server design with to be robust lightweight and fast. It reduces the friction of deploying RoR apps and have an extensive number of enterprise grade features which become extremely useful in production. Some of its key features are the ability to handle more traffic(up to 4x more in RoR), its reduced need of maintenance and improved security.

---

<sup>3</sup><http://rubyonrails.org/>

### 5.2.3 POSTGRES

Rails has an ORM called Active Record through which it can connect to various databases (e.g. Postgres, MySQL, SQLite, etc.). We choose to connect it to a Postgres database, because this is a very active and widely used database. This means there are many tools that connect with the database and we can be sure of future maintenance. Also, Postgres is more performant than SQLite. If future developers want to run a different database, they can switch without any trouble. Setting up the configuration file is sufficient, since our code is database-agnostic.

## 5.3 ALGORITHM-SIDE

### 5.3.1 SCALA

For the core of the algorithm we chose to work with Scala<sup>4</sup> for a couple of reasons:

- It is a JVM-based language and can seamlessly interact with Java and thus we could make good use of OpenTS<sup>5</sup>, an open source tabu search implementation in Java.
- It supports higher order functions and persistent immutable data structures, which play together nicely to ease the processing and transformation of nested data.
- Since the algorithm implementation is stateless, we can use the rich support in Scala for multithreaded and distributed systems to easily scale it out.
- We are familiar with Scala from our previous work.

### 5.3.2 SBT

SBT<sup>6</sup> naturally goes with Scala as the dependency management and build tool for the job.

### 5.3.3 SPRAY

Spray<sup>7</sup> is a lightweight HTTP/REST layer built with Akka<sup>8</sup> and is also easily scalable. Its features include automatic content type and error handling and composable routes, built of nested and reusable declaratives.

## 5.4 HAPI ON NODEJS VS. RUBY ON RAILS

In this project our initial attempt to build the backend server was done with a mixture of Node.js using the Hapi framework and Bookshelf.js which is built on top of the Knex.js library for Postgres SQL. This decision was made since Node has been already proven to be able to

---

<sup>4</sup><http://www.scala-lang.org/>

<sup>5</sup><http://www.coin-or.org/0ts/>

<sup>6</sup><http://www.scala-sbt.org/>

<sup>7</sup><http://spray.io/>

<sup>8</sup><http://akka.io/>

function as a robust web server implementation, and also since it was a technology which most of the group felt very comfortable working with. When designing our database we opted for a relational database as Postgres instead of MongoDB since it fitted best with our design criteria.

However, even though the Node community is very active and there is a lot of hype and development around it, our development process was slower than initially planned. At this point we decided to switch from Node to RoR since Rails is more of an opinionated framework and has a lot of out of the box functionality built in to speed up prototyping. Also we would get the advantage of the ActiveRecord ORM when handling nested relationships. This as opposed to implementing the relationships ourselves using Bookshelf.js library as we would have had to do in Hapi.

Once we changed our framework we split up the backend responsibilities so one team member would handle the Rails implementation while the other would fully Dockerize deployment and handle DevOps.

## 6 DEPLOYMENT

There are two ways to run the project: either by running a regular installation on the local machine or by using Docker. We added support for Docker to allow for easier deploys: if a version runs on our development computers (in Docker), then it will also run on the production server.

We recommend Docker to be used for the production, and normal installations of Rails/Postgres/SBT for development. The rationale behind this recommendation is that we tested our work on Docker, but not on every other platform.

In this section, we first give some description on how to use Docker in this specific setting. We also give some details about the virtualized networking, which may be useful for future developers.

### 6.1 STARTING DOCKER

In this section, we show how to do a quick installation of Docker and Docker Compose, and how to build and run the images of this project. Note that this is a short (and by the time you read this possibly outdated) installation guide: for the newest installation of Docker (Compose), we refer to their site: <https://docs.docker.com/compose/install/>. Also, some of these commands need root permission: if you get an error, prepend the command with "sudo", and try again.

---

```
1 # Install on Linux (Ubuntu)
2
3 # (If you don't have curl, first install curl)
4 $ curl -sSL https://get.docker.com/ | sh
5
```

```

6 # Beware: fixed version number. This one we tested on and worked for us.
  ↳ Future versions may have breaking changes.
7 $ curl -L
  ↳ https://github.com/docker/compose/releases/download/1.3.3/docker-compose-`uname
  ↳ -s`-`uname -m` > /usr/local/bin/docker-compose
8 $ chmod +x /usr/local/bin/docker-compose
9
10 # If you want to run Docker as a normal user (i.e. not prepending sudo
    ↳ everytime)
11 $ sudo usermod -aG docker <your username>
12 # Now logout and login again to update the group changes
13
14 # Install on OSX
15 $ brew install docker
16 $ brew install boot2docker
17
18 # Download and run the project
19 # Fetch repository: only the first time
20 $ git clone https://github.com/juancroca/ios.git .
21
22 # Build containers and run the images
23 $ docker-compose build
24 $ docker-compose up -d

```

---

When issuing these commands, the latest versions of the server and algorithm are fetched from Github, installed in Docker images and started. The first time one runs these commands, all dependencies have to be downloaded, which can take quite a while. After that, the cache is used and everything runs faster.

Notice that we pass a -d flag; this starts Compose in daemon mode, meaning that the process will continue after the current user has logged out.

## 6.2 STOPPING DOCKER

To stop docker, simply issue the following command:

---

```

1 $ docker-compose stop

```

---

Note that this only works in the root directory of the project. Docker-compose requires the configuration files (docker-compose.yml and the Dockerfile files) to work, which it can't find if these commands are issued from another folder.

### 6.3 DEPLOYING NEW VERSIONS

To fetch a new version of the server, we have to fetch and merge the new version from github:

---

```
1 $ git fetch && git pull
```

---

To run it, execute:

---

```
1 $ docker-compose build
2 $ docker-compose up
```

---

This will automatically kill any older versions that may be running and replace them with the updated versions. Also, this will automatically update the algorithm: our startup scripts always fetch the latest version.

View running instances To see whether all containers are up and running:

---

```
1 $ docker ps
2 CONTAINER ID   IMAGE                COMMAND                  CREATED    STATUS    PORTS
   ↪ NAMES
3 e2361743cf36   ios_ruby:...         \"/sbin/my_init\"      ... ago   Up       ...
   ↪ 80->80/tcp    ios_ruby_1
4 305afbcf932e   ios_scala:...        \"/sbt run re-start\"   ... ago   Up       ...
   ↪ 8080/tcp      ios_scala_1
5 9993ee424f3b   postgres:...         \"/docker-entrypoint.   ... ago   Up       ...
   ↪ 5432/tcp      ios_db_1
```

---

### 6.4 ACCESS TO THE VMs

We choose not to expose SSH ports on any of the hosts for security reasons. We can, however, still access the terminal of running instances through Docker. This is only possible from the host machine.

---

```
1 $ docker ps
2 CONTAINER ID   IMAGE                ...
3 e2361743cf36   ios_ruby:latest     ...
4 $ docker exec -t -i e2361743cf36 /bin/bash
5 root@e2361743cf36:/home/app/webapp# whoami
6 # just to prove it: who am i?
7 root
8 root@e2361743cf36:
```

---



## 6.5 DOCKER NETWORKING

There are two ways of opening ports in Docker: opening ports internally and externally. To specify which ports are open, one can use the docker-compose.yml file:

---

```
1 ruby:
2   build: .
3   ports:
4     - "80:80"
5   links:
6     - db
7     - scala
```

---

In this example, we define the behaviour of a host labelled `ruby`. It exposes port 80 to the outside world, and has access to the hosts labelled `db` (the Postgres server) and `scala` (which runs the algorithm). It cannot access all ports of `db` and `scala`, only the ports exposed by these images.

Since `outsiders` do not need direct access to the database or algorithm server, we only expose the ports of these images internally. For example, the `pg` host is defined by the following configuration:

---

```
1 db:
2   image: postgres
3   expose:
4     - "5432"
```

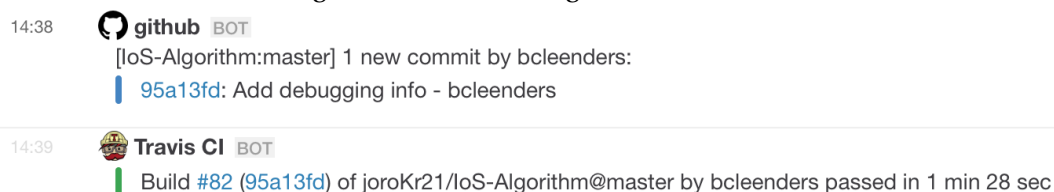
---

Notice that we only specify the port Postgres uses (5432), without specifying a corresponding outer port.

## 7 CONTINUOUS INTEGRATION

We used Travis to do continuous integration. Combined with Slack, this gives us feedback on our code quality on every git push. It monitors our GitHub repository and every time it detects a change, it runs test against the latest version. In Slack, this looks as following:

Figure 7.1: Slack message on Travis tests



We agreed on the convention that in a pull-request to the master branch all tests must pass before the code is merged. Because of this you can ensure that at all times there is a working master version (or at least one that passes the tests). This method of continuous testing makes it possible to add new code and verify that all the existing code is still functioning. The continuous integration server makes it possible to deploy new code several times a day because everything is tested thoroughly.

A problem we ran into, is that Travis only runs tests for one specific language. To be able to run tests both for Scala and for Ruby, we had to split the project into two separate git repositories, with two separate Travis test suites. This did have the consequence that although each might be working separately, the combination of the two might fail without us knowing. We found no solution for this, other than doing manual tests.

## 8 RESULT EVALUATION

TODO  
TODO  
TODO  
TODO  
TODO  
TODO

## 9 GROUP WORK EVALUATION

The construction of the previously described app was not the only result expected for this course. There were other objectives like how the group is managed, how are obstacles tackled and how are decisions and compromises are made. It was undoubtedly a great learning experience, as you can see we did have some hiccups during the development of this solution. Initially finding the correct algorithm to solve this specific problem was not easy it involved a lot of research from some of our team members and some heated discussions. After debating several options we could finally agree on a path we all found viable.

On the other hand, on the backend server side we encounter a major bottleneck with our initial setup so we had to quickly adapt to be able to solve this problem in the expected time-frame, so that is why the change in stack occurred mid development. We were able to accomplish this due to our well-managed communication. We had a weekly face-to-face hacking day during which the entire team got together and devoted its entire time to this project.

Besides this, we started a group chat in Slack which was fully integrated with our git and continuous integration server. We had two repositories corresponding to each of the different components of the solution, this would give the team members the ability to track the project status as a whole as well as the progress of each component, we could see at all times who was developing which features and also which were completely done.

We did notice that, as there were more and more features, it became more difficult to maintain a good overview of the exact status of every independent feature. If we had continued for

a longer time and added more features, we would have had to invest (more) time in a feature tracking system.

After we felt confident with the solution we had built we ran tests comparing our implementation to a brute force solution, we found out that when having a small test dataset the results were 98% accurate and it always found a local optimum. Even though tests with bigger datasets could not be run due to the very rapidly increasing complexity, we are confident that if this test would be feasible the accuracy of our results would not vary that much than the ones obtained in small data sets.

## 10 FUTURE WORK

In the future we would like to see some additional work done in the next areas:

**I18n<sup>9</sup>:** It would be nice if the app would take into account internalization(not only the front end), so the tool could be used in all different courses for TU Berlin where English is not the expected language. Rails is already setup to handle this kind of feature is just a matter of editing the front end and setup the proper yaml files. This tweak would also involve some work regarding the skill selection since currently we are using a predefined skillset in English.

**Integration with more services:** It would be useful to have a more advanced authentication mechanism in place when communicating with ISIS through the LTI protocol. This protocol already supports OAuth, and we would recommend to also have this type of authentication between the algorithm API and the RoR app. Versioning the algorithm API would be a good idea so it would be able to function when the json format evolves to satisfy more features.

**Standalone API:** Currently there are some endpoints who are able to receive a request and respond all in json format but we would like to extend this functionality so all of the endpoints are able to function this way and you can have different front end solutions built on top of the functioning RoR app.

**Extending Algorithm with more criteria:** Currently the only way we can see if a student is the correct fit for a course or not is based on the score he grants himself regarding the skillset of each group. This is a bit too subjective and if we add something like students rating each other this could improve the selection criteria on the objective function.

**Linking history:** If we integrate results of previous courses, we might be able to give a more accurate description of what students work well together. For example: at the end of the semester, students could be offered the option to rank how well they liked their other team-members. This could then be taken into account for future distributions for next courses.

## REFERENCES

- [1] F. Glover and G. A. Kochenberger. *Handbook of metaheuristics*. Springer Science & Business Media, 2003.
- [2] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

---

<sup>9</sup><http://guides.rubyonrails.org/i18n.html>

- [3] R. Hübscher. Assigning students to groups using general and context-specific criteria. *Learning Technologies, IEEE Transactions on*, 3(3):178–189, 2010.
- [4] E. Ronn. Np-complete stable matching problems. *Journal of Algorithms*, 11(2):285–304, 1990.