Project Time Tracking and Invoicing Platform

SD21w2

Doroftei, Cristina
Mocanu, Gheorghe Marian

March 09, 2021

1.Introduction

1.1 Purpose

The purpose of this document is to encapsulate the functionalities of the system we have been implementing and to describe the structure of our solution, along with the technologies used.

1.2 Glossary

Project: A venture to produce a new deliverable.

User: A person who can log in to the platform.

Task: A piece of work that needs to be completed.

Being assigned to a project: Being a member of a project.

Team: A group of individuals in an organization. A person can be part of several teams.

1.3 Problem definition

According to the most recent Billentis Study, 55 billion paperless invoices have already been sent out around the world. But that's just a tenth of the total. For processing purposes, this means that the remaining 90% must be entered manually – or the invoices must be scanned and digitized. This adds up to a significant extra cost for the receiver and leads to errors.

Techture, a software company based in Copenhagen, wants to solve this problem by developing a project time tracking and invoicing platform.

The system is to, first of all, allow users to create projects, link clients and assign people to them. Since working in teams is a common practise, the system should allow users to create and manage their company's teams.

Users should be able to add tasks to projects and estimate how long it will take until the task is considered to be done. There are three main types of status categories: to do, in progress and done. The user can create new project statuses and task statuses, define their category and set them to projects and tasks.

Secondly, the system is to allow users to create invoices and track payments and get an overall overview of what was invoiced to the clients, what was paid and what is still outstanding.

2. Technical Analysis

2.1 Explanation of choices for RDBMS, ORM and programming language

While developing the project, we used the following technologies:

- **MySQL:** MySQL is an open-source relational database management system. We mainly chose this RBMS because it is robust and compatible with almost all types of programming languages. It has a big community of developers using it, so we can always find many technical and non technical documentation online

- **NodeJS:** Node.js is an open-source, cross-platform, back-end JavaScript runtime environment**.** Since Javascript has proven to be one of the most popular programming languages and we both wanted to try it for the backend side, Node.js seemed like the perfect choice. Thus, we saw the huge potential of using Node.js from its rich ecosystem, consisting of npm modules.

- **Sequelize:** Sequelize is a promise-based Node.js ORM for multiple RDBMS, including MySQL. It is a great ORM in the NodeJS space with an active community on Github. Sequelize is very easy to learn and its interesting features, like magic methods, helped us a lot.

2.2 Conceptual and Logical Data Models

The purpose of the conceptual data model is to define *what* the system business scope contains and it hardly has any detail on how the actual database structure is. The logical data model defines *how* the system should be implemented, regardless of the DBMS.

We find these 2 models crucial in the process of developing the database structure, because they provide a better foundation and focus for determining scope.

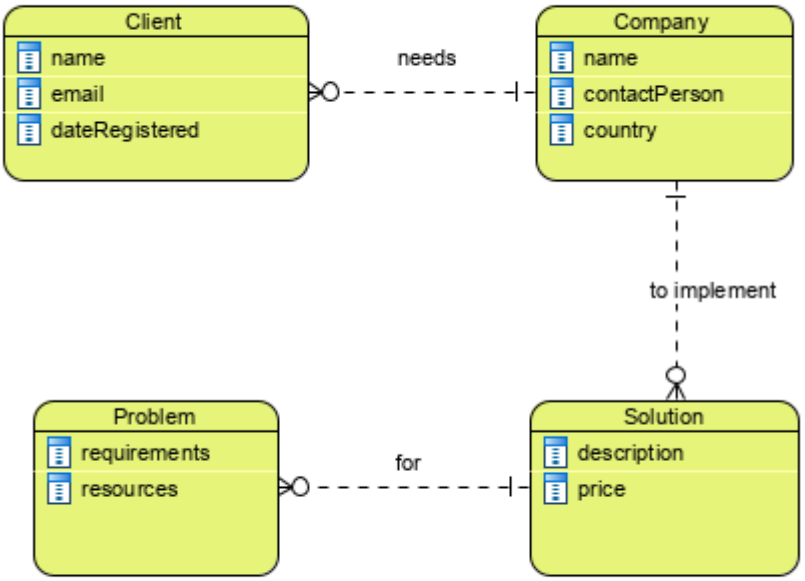


Fig. 1 - Conceptual Data Model

As it is illustrated in *Fig.1*, the main business model of the system is: a client needs a company to implement a solution for a problem, and the scope of the platform is to help the hired company to easily manage the time registrations of the developing process of the solution and create invoices given to the client.

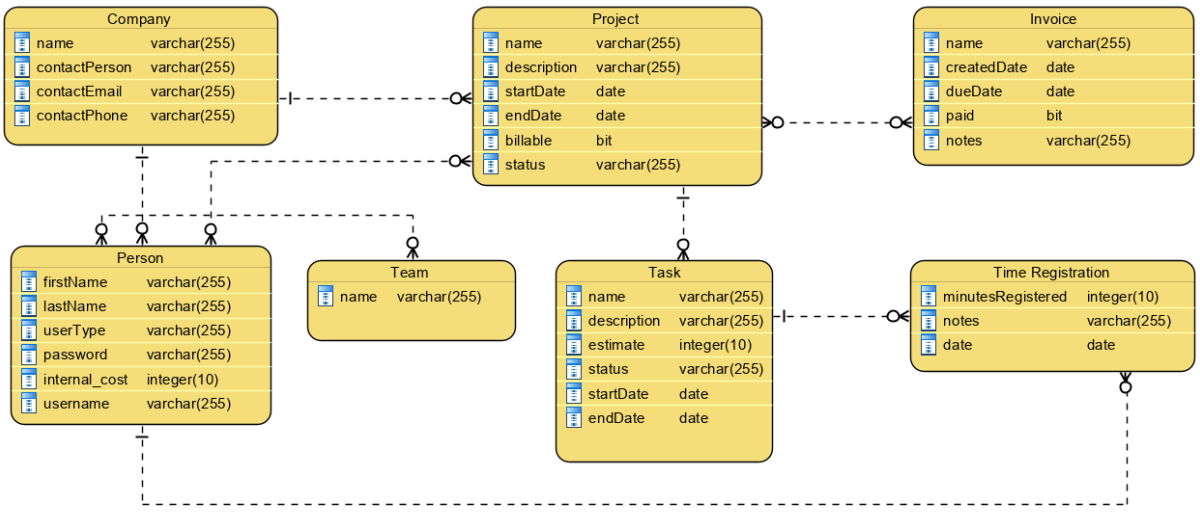


Fig.2 - Logical Data Model

A company can have many persons/employees, each of which can be part of multiple teams and projects. Since teams can also have multiple people in them, there is a many-to-many relationship between persons and teams. Additionally,

projects can also have multiple people assigned to them, so there is a many-to-many relationship between persons and projects.

A company can manage many projects and users can create multiple invoices on a project. A project can have multiple tasks that people can register time on.

2.3 Physical Data Model

A physical data model represents how the structure of the tables will be in the database. We chose to implement our database in the following way:
Fig.3 - Physical Data Model

2.3.1. Normalization and denormalization decisions

The system needs to handle projects that are required by a client-company and solved by a hired-company. Our initial idea was to create two separate tables: Clients table and Companies table, so that the company_id and client_id foreign keys inside of the Project table will point to two different tables:

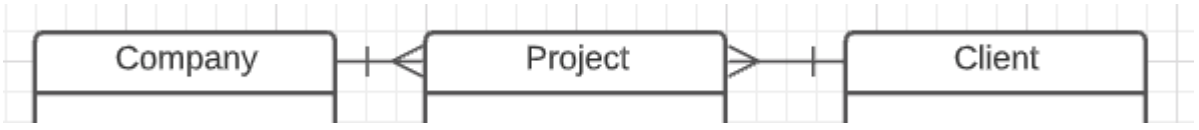
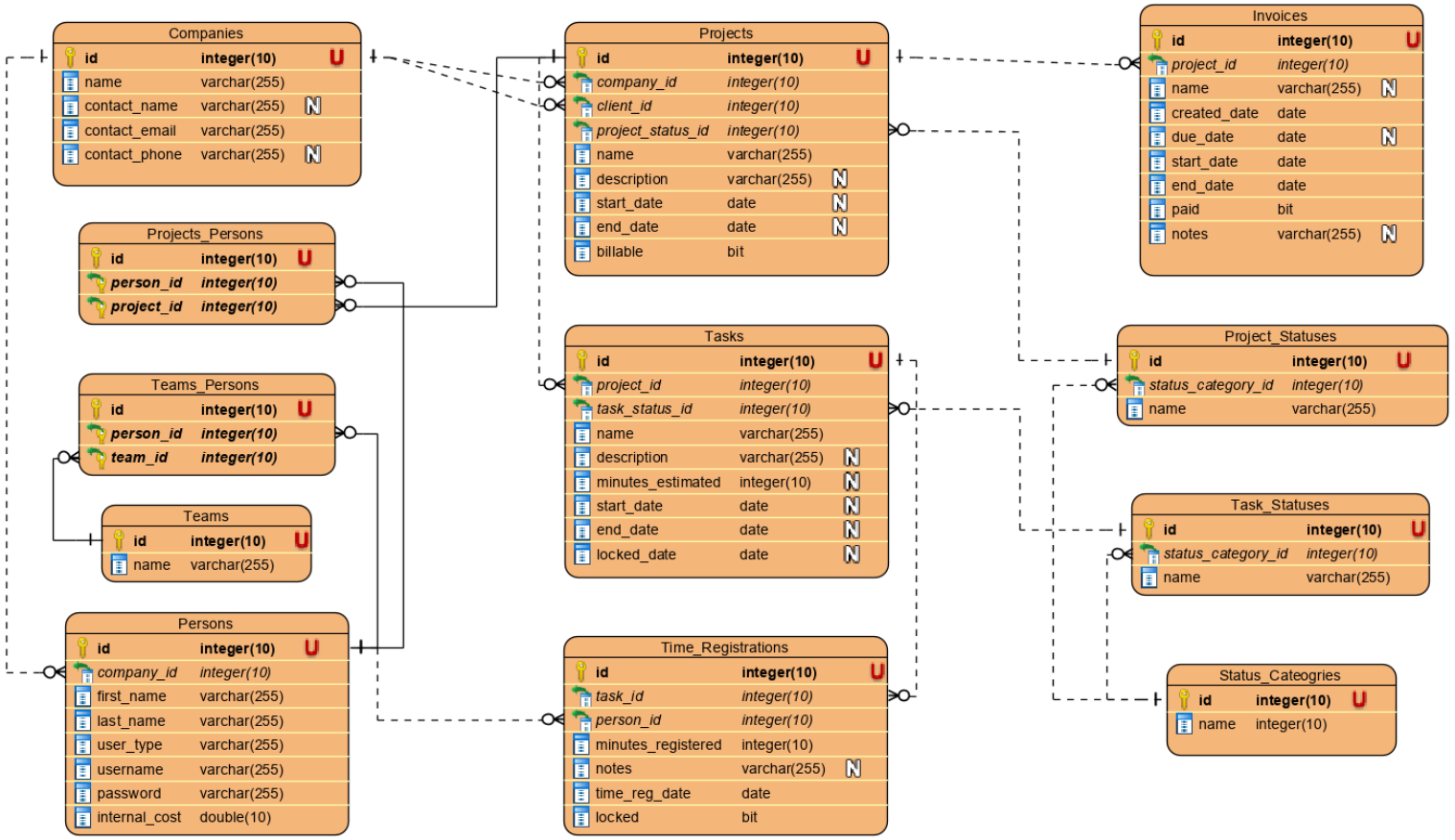


Fig.4 - Short entity diagram showing the initial structure of the project-company-client relationship

Initially, the approach from *Fig.4* seemed like a clean and understandable structure. Conversely, if there will be cases when we want to get information about a project regarding its client-company and hired-company, we will need to execute two joins, which would definitely be a performance issue. Furthermore, a client-company



and a hired-company are both companies, which means that they can share the same attributes.

Hence, we came to the conclusion that we can denormalize this structure by having the company_id and client_id foreign keys pointing to the same table: the Companies table.

There are three main status categories: to do, in progress and done. Since the user can create different project statuses and task statuses which are defined by these status categories, we decided to normalize the database by creating a Status_Categories table, which will always contain three values: to do, in progress and done. The Project_Statuses and Task_Statuses tables will make use of the Status_Categories. Even if we will have to execute two joins when we want to get the status category of a project or a task, the Status_Categories table has an insignificant amount of rows, so we did not consider it a performance issue. Thus, we kept the normalization.

The entity Teams_Persons is an associative entity, implementing a many-to-many relationship between person and team, outlined in the description of *Fig.2*. In the same manner, the entity Projects_Persons is an associative entity, implementing a many-to-many relationship between person and project, also outlined in the description of *Fig.2*.

2.3.2 Explanation of choices for data types and keys

The Companies table contains all the data necessary for describing a real company within the system. Besides the id which is an integer primary key, all the other entries representing the contact information are held in varchar fields.

Representing the users of the system, the Persons table has fields like, *username* and *password* which are the credentials that a user will use to access the system, *internal_cost* being the amount one user is paid per hour and of course a foregin key, *company_id*, pointing to the company that the user works for.

Users can create projects which will be further stored in the Projects table having three foreign keys: *company_id* and *client_id* being a record saved previously in Companies table, and *project_status_id* defining its status. In addition to *name* and *description*, there are two date fields representing the start and deadline of a project: *start_date* and *end_date*. Finally, we have a bit type of attribute called *billable* that will have the value *0* when the project is non-billable and *1* otherwise.

After having created a project, the system's user should create tasks. These are stored under the Tasks table that has two foreign keys: *project_id* pointing to its project and *task_status_id* representing its status. The two critical needed fields in this table are *minutes_estimated* which has the scope of letting any assigned person to know in an estimative manner how much time should one spend working on it, and *locked_date* describing the date up to which a user cannot register time in case there has been an invoice issued. Furthermore, a task has 2 date fields, *start_date* and *end_date*, that are initially inherited from its project if not provided.

At any time, the user can create different statuses for both projects and tasks. When the user creates a status for a project it will be stored in *Projects_Statuses* table. Likewise, a task's status is saved in *Tasks_Statuses*. Both tables have a foreign key, *status_category_id*, pointing to a category defined in *Status_Categories* table.

In order to create invoices, a user needs to have registered time on tasks. An instance of registered time is saved in the Time_Registrations table having two foreign keys: *task_id* defining the task on which the user has registered time, and *person_id* describing the person that has registered time. A time registration cannot be created without having supplied an amount of time represented by *minutes_registered* and a date represented by *time_reg_date*. There is also a *notes* field of type *text* serving the scope of saving any details that the user might find useful. A *bit* type of data in this table is the *locked* field which is initially recorded as 0, and only changed into 1 whenever an invoice has been issued within a timeframe including *time_reg_date*.

The next logical step for a user is to create an invoice. Its details are saved in the Invoices table having only one foreign key, *project_id*, pointing to the project an invoice is issued on. In this table, there are four date entries stored:

- *created_date*: when the invoice was created;
- *due_date*: up to when the invoice needs to be paid;

- *start_date*: the start of the daterange the user wants to create invoices on
- *end_date*: the end of the daterange the user wants to create invoices on

Moreover, there is a *notes* field which has the scope of letting the user to save any additional details as a text and, lastly, a *paid* field described by *0* when the invoice has not been paid and *1* otherwise.

2.4 Description of used stored objects

```
DELIMITER $$
CREATE TRIGGER after_invoice
AFTER INSERT ON invoices FOR EACH ROW
BEGIN
    DECLARE inserted BOOLEAN;
    SELECT 1 INTO @inserted
        FROM
            invoices
        WHERE
            invoices.id = NEW.id;
    IF @inserted = 1 THEN
        UPDATE tasks
            INNER JOIN projects
                ON projects.id = tasks.project_id
            INNER JOIN time_registrations
                ON tasks.id = time_registrations.task_id
        SET
            time_registrations.locked = b'1'
        WHERE
            projects.id = NEW.project_id
                AND
            time_registrations.time_reg_date
                BETWEEN NEW.start_date AND NEW.end_date;
        UPDATE
            tasks
        SET
            tasks.locked_date = NEW.end_date
        WHERE
            tasks.project_id = NEW.project_id;
    END IF;
END $$
```

Fig.5 - Trigger executed after inserting an invoice

Fig.5 represents the syntax of a trigger which updates two fields in two different tables, both fields being updated after an invoice is inserted. In the *time_registrations* table, the *locked* field gets updated to 1 only for the records used in the invoice. In the *tasks* table, *locked_date* gets updated to the *end_date* of the invoice only for the tasks used in the invoice.

First we start by specifying the name of the trigger, followed by the AFTER INSERT clause. Since we need to execute multiple statements in the trigger body, we need to use the BEGIN END block and change the default delimiter to $$. We create a boolean variable that will have a true state only if the newly inserted record has been successfully added. Based on this variable, the trigger will firstly update all

the entries from the *time_registrations* table identified by joining *tasks* with *time_registrations* and *projects* whose *time_reg_date* is BETWEEN the date range of the invoice. Secondly, the trigger will update the *locked_date* attribute of all *tasks*, with invoice's *end_date*, *tasks* identified by the invoice's project.

```
CREATE VIEW invoices_amounts AS
    SELECT
        invoices.id,
        invoices.project_id,
        sum(time_registrations.minutes_registered) as minutes_registered,
        round(sum(time_registrations.minutes_registered / 60 * persons.internal_cost), 2) as total,
        invoices.start_date,
        invoices.end_date
    FROM
        time_registrations
            INNER JOIN tasks ON time_registrations.task_id = tasks.id
            INNER JOIN projects ON tasks.project_id = projects.id
            INNER JOIN invoices ON invoices.project_id = projects.id
            INNER JOIN persons ON persons.id = time_registrations.person_id
    WHERE
        time_registrations.locked = b'1'
    GROUP BY
        invoices.id;
```

Fig.6 View calculating total amount for each invoice

Another stored object that the system is making use of, is this view in which we calculate an invoice's total amount to be paid. Since each person in each company is getting paid with a different amount per hour that is stored in *persons.internal_cost* attribute, the view *invoices_amounts* calculates the total amount to be paid for each person and sums it into *total*. We can make use of this view any time we would want to query all the invoices for a project or even when we want to query all the invoices within a timeframe by simply executing a select statement.

```
DELIMITER $$
CREATE trigger after_project
    AFTER insert
    on projects
    for each row
begin
    select 1 into @inserted from projects where projects.id = NEW.id;
    if @inserted = 1 then
        insert into task_statuses (status_category_id, name, project_id)
        VALUES (1, 'To do', NEW.id),
               (2, 'In progress', NEW.id),
               (2, 'Ready for test', NEW.id),
               (3, 'Done', NEW.id);
    end if;
end $$
```

Fig.7 Trigger for assigning default task statuses

Fig. 7 represents the syntax of a trigger which inserts 4 new entries in table *task_statuses*. It checks whether a project entry was successfully created and inserts 4 new rows into task_statuses:

- To do
- In progress
- Ready for test
- Done

Each row has a *status_category_id* as foreign key, pointing to a status category, *name* that defines the name of a status and *project_id* as a foreign key representing the newly inserted project. Thus, each project can have different statuses for its tasks.

```
DELIMITER $$
CREATE trigger after_company
    AFTER insert
    on companies
    for each row
begin
    select 1 into @inserted from companies where companies.id = NEW.id;
    if @inserted = 1 then
        insert into project_statuses (status_category_id, name, company_id)
        VALUES (1, 'Opportunity', NEW.id),
               (2, 'Planning', NEW.id),
               (2, 'Running', NEW.id),
               (3, 'Completed', NEW.id);
    end if;
end $$
```

Fig.8 Trigger for assigning default project statuses

Very similar to fig.7, in fig.8 is presented the source code of a trigger that creates 4 default project statuses when a company is created.

- Opportunity
- Planning
- Running
- Completed

The only major difference between these 2 triggers is that the second one (fig.8) supplies the id of the newly inserted company under the *company_id* field, which references the company that can make use and manage these project statuses in their unique way.

2.5 Explanation of the implementation of transactions

A transaction is a sequential group of statements, queries, or database manipulation operations, which is performed as if it were one single work unit. Transactions are not considered complete unless each statement/query/operation within the group is successful.

We used transactions when performing our database migrations, in order to ensure that all instructions are successfully executed or rolled back in case of failure.

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    return queryInterface.sequelize.transaction((t) => {
      return Promise.all([
        queryInterface.addColumn(
          "project_statuses",
          "company_id",
          {
            type: Sequelize.INTEGER,
            allowNull: true,
            references: {
              model: "companies",
              key: "id",
            },
          },
          { transaction: t }
        ),
      ]);
    });
  },

  down: async (queryInterface, Sequelize) => {
    return queryInterface.sequelize.transaction((t) => {
      return Promise.all([
        queryInterface.removeColumn("project_statuses", "company_id", {
          transaction: t,
        }),
      ]);
    });
  },
};
```

A managed transaction is started by passing a callback to `sequelize.transaction`. Then, the next steps are:

- Sequelize starts the transaction and obtain the transaction object `t`
- Sequelize executes the callback provided, in this case adding the `company_id` column to `project_statuses` table, and passes `t` into it
- If the callback throws, Sequelize will rollback the transaction
- If the callback succeeds, Sequelize will commit the transaction

2.6 Authentication

TODO Explain successful registration scenario. Mention that at signup we create an admin account.

2.6.1 Sign up

The sign up post request is executed once the `/signup` endpoint is reached. We decided that, when a user wants to sign up, he/she also needs to create his/her own company.

```
router.post("/signup", validate, authController.registerCompany);
```

Before executing the function that will register the company, we firstly need to validate the username and password, by executing the `validate` middleware.

```
const validate = (req, res, next) => {
  Persons.findOne({ where: { username: req.body.username } }).then((person) => {
    if (person) {
      return res.sendStatus(409);
    }
  });
  if (req.body.password.length < 5) {
    return res.sendStatus(401);
  }
  next();
};
```

The `validate` function is looking in the persons table in the database for a person with the same username. If that person was found, the function will return the status code 409, meaning that the request could not be completed and the user will have to try again with another username. Besides that, if the password is less than 5 characters long, the middleware will return status 401. If both username and password are valid, the `next()` function will pass control to the next middleware function, `registerCompany.`

```
exports.registerCompany = (req, res, next) => {      Marian Mocanu, 3 wee
  Companies.create({
    name: req.body.name,
    contact_name: req.body.contact_name,
    contact_email: req.body.contact_email,
    contact_phone: req.body.contact_phone,
  }).then((company) => {
    company
      .createPerson({
        first_name: req.body.first_name,
        last_name: req.body.last_name,
        user_type: "ADMIN",
        username: req.body.username,
        password: bcrypt.hashSync(req.body.password, 12),
        internal_cost: req.body.internal_cost,
      })
      .then((person) => res.send({ company: company, person: person }));
  });
};
```

The `registerCompany` function is firstly creating a company with the relevant fields from the request body. After the company was successfully created, sequelize will return an object which holds the newly made company. By using the magic method called `createPerson`, which was automatically created by sequelize, the user that just signed up will be registered as an admin by default.

2.6.2 Login

```
router.post("/login", authController.login);
```

The login post request is executed once the `/signup` endpoint is reached.

```
exports.login = (req, res, next) => {
  const { username, password } = req.body;
  Persons.findOne({ where: { username: username } }).then((person) => {
    if (!person) {
      return res.status(401).send("Invalid username!");
    }

    if (bcrypt.compareSync(password, person.password)) {
      req.session.person = person;      Marian Mocanu, 3 weeks ago via PR #1
      return req.session.save((err) => {
        if (err) {
          console.log("Error!", err);
          next(err);
        } else {
          res.sendStatus(200);
        }
      });
    } else {
      return res.status(401).send("Invalid password!");
    }
  });
};
```

The `login` function takes the username and password from the request body and firstly checks if there is any person in the database with that username. If the person is not found, the function returns an error is the message “Invalid username”.

If the user was found, we compare the password from the request body and the actual person’s hashed password by using bcrypt. If the values are not matching, we return an error with the message “Invalid password!”.

If they are matching, we add the person’s fields to the req.session and save the request session. If the saving fails, we pass the error to the appropriate error handling middleware.

Finally, if everything was successful, we return a 200 status code.

2.6.3 Logout

The logout post request is executed once the `/logout` endpoint is reached.

```
exports.logout = (req, res, next) => {
  req.session.destroy((err) => {
    if (err) {
      console.log("Error when logging out!");
      return res.sendStatus(404);
    } else {
      return res.sendStatus(200);
    }
  });
};
```

This short function is deleting the request session by calling `.destroy()`. If any error occurs, we send a 404 status code. If the action was successful, we return a 200 status code.

2.7 Security

TODO Backend request validator

2.7.1 Explanation of users and privileges

Name	Description
Admin	Have limitless access to the entire platform and can configure everything from the admin panel and throughout the platform.
Controller	Have similar rights to Admins, except access to the admin panel. They can configure anything related to projects, tasks, invoices, task statuses and project statuses, but not teams or persons.

Fig.9 - Diagram that describes all the user types that interact with the Time Registrations Report

We decided to have two types of users, as described in *Fig.9.* In the server-side code, we added a middleware called `isAdmin` to all the endpoints that can only be accessed by admins, which verifies if the request was sent by an admin. Some examples of how this middleware is used are shown below:

```
router.put("/persons/:id", isAdmin, personsController.updatePerson);      You, 2 days ago vi

router.put("/time_registrations/:id", timeRegistrationsController.updateTimeRegistration);

router.put("/task_statuses/:id", taskStatusesController.updateTaskStatus);

router.put("/companies/:id", isAdmin, companiesController.updateCompany);

router.put("/teams/:id", isAdmin, teamsController.updateTeam);
```

This function performs the following steps:

```
module.exports = (req, res, next) => {
  const userType = req.person.user_type;
  if (!userType === "ADMIN") return res.sendStatus(400);
  next();
};
```

- Take the usertype from the person stored in the request object
- If the usertype is not admin, the function will return a response with status code 400.
- If the user is an admin, the `next()` function will pass control to the next middleware function

2.7.2 Middleware

When it comes to security, one way to determine whether or not a user is logged in is to examine the request.session variable. As a result, when a user logs in, the server creates a session and saves it in the database.

We checked for the previously stored *req.session.person* in our implementation of this middleware. This object used to exist as a sequelize type of object during the login process. Its type and references were lost because it was saved in the database. As a result, it is very simple and useful to recreate it, allowing its magical methods to be used whenever they are required.

```
exports.isAuthenticated = (req, res, next) => {
  if (req.session.person) {
    Persons.findOne({ where: { username: req.session.person.username } })
      .then((person) => {
        req.person = person;
        next();
      })
      .catch((err) => {
        console.log(err);
      });
  }
  if (!req.session.person) {
    return res.sendStatus(400);
  }
};
```

If the *req.session.person* does not exist or is not defined, the current user is not logged in the system. As a result, the server responds with a status code of 400 (actual 403), indicating that the user is not authorized to access that particular resource.

3. Server Routing Structure

The Model View Controller (MVC) pattern has grown in popularity among web developers. People liked it because it is a very simple abstraction that primarily aids in code organization and provides a dynamic response to HTTP requests.
While the JavaScript frameworks we use today are not true MVC, we can apply what we've learned from MVC to our large JavaScript applications. Below is an example of how the routes are structured in the application.

HTTP Verb	Path	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

Fig.10 Routing convention

3.1 Authentication

- Sign up
    - Method: POST /signup
    - Request body:
        - name: string, required - company name
        - contact_name: string - company's delegated contact person's name
        - contact_email: string, required - company's contact email
        - contact_phone: string - company's contact phone number
        - first_name: string, required - registering user's first name
        - last_name: string, required - registering user's last name
        - username: string, required - chosen username used to log in
        - password: string, required - chosen password used to log in
        - internal_cost: double, required - salary payment per hour
    - Response: "response" - object containing the newly created *company* and *person* with ADMIN permissions.

- Log in
    - Method: POST /login
    - Request body:
        - username: string, required - part of login credentials
        - password: string, required - part of login credentials
    - Response: “status” - 200, representing successful log in
- Log out
    - Method: POST /logout
    - Response: “status” - 200, representing successful log out

3.2 Projects

- Get all projects for company
    - Method GET /projects
    - Request person (retrieved from session)
    - Response: “projects” - an array containing Project items
- Get project by id
    - Method GET /projects/:id
    - Route parameters:
        - id: number - a project’s specific id
    - Response: “project” - a Project item
- Create project
    - Method POST /projects
    - Request body:
        - client_id: number - the project’s client company
        - name: string - the project’s name
        - billable: boolean - defines whether the project is billable
    - Response: “project” - the newly created project
- Update project
    - Method PUT /projects:id
    - Route parameters:
        - Id: number - a project’s specific id
    - Request body:
        - client_id: number - the project’s client company
        - name: string - the project’s name
        - billable: boolean - defines whether the project is billable
    - Response: “project” - the newly updated project
- Delete project
    - Method DELETE /projects:id
    - Route parameters:
        - Id: number - a project’s specific id
    - Response: “200” - status indicating a successful action

3.3 Tasks

3.4 Time registrations

3.5 Invoices

3.6 Persons

3.7 Statuses