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 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.

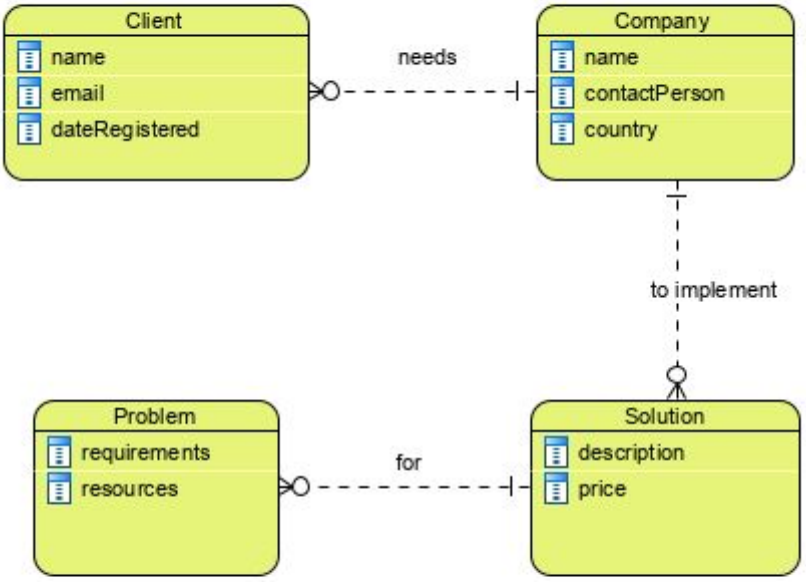

Fig. 1 - Conceptual Data Model

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.

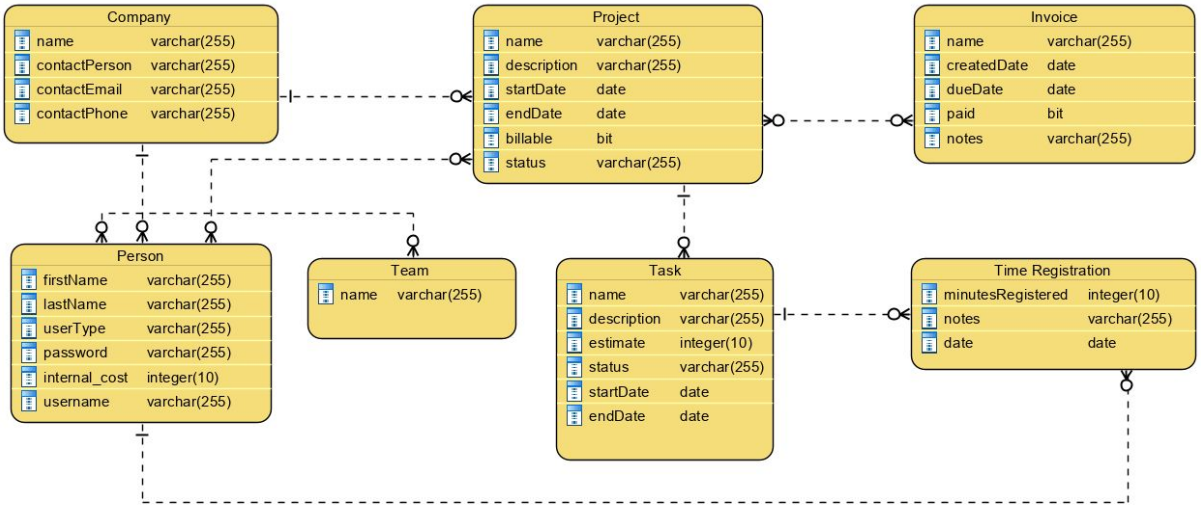

Fig.2 - Logical Data Model

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.

2.2 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:

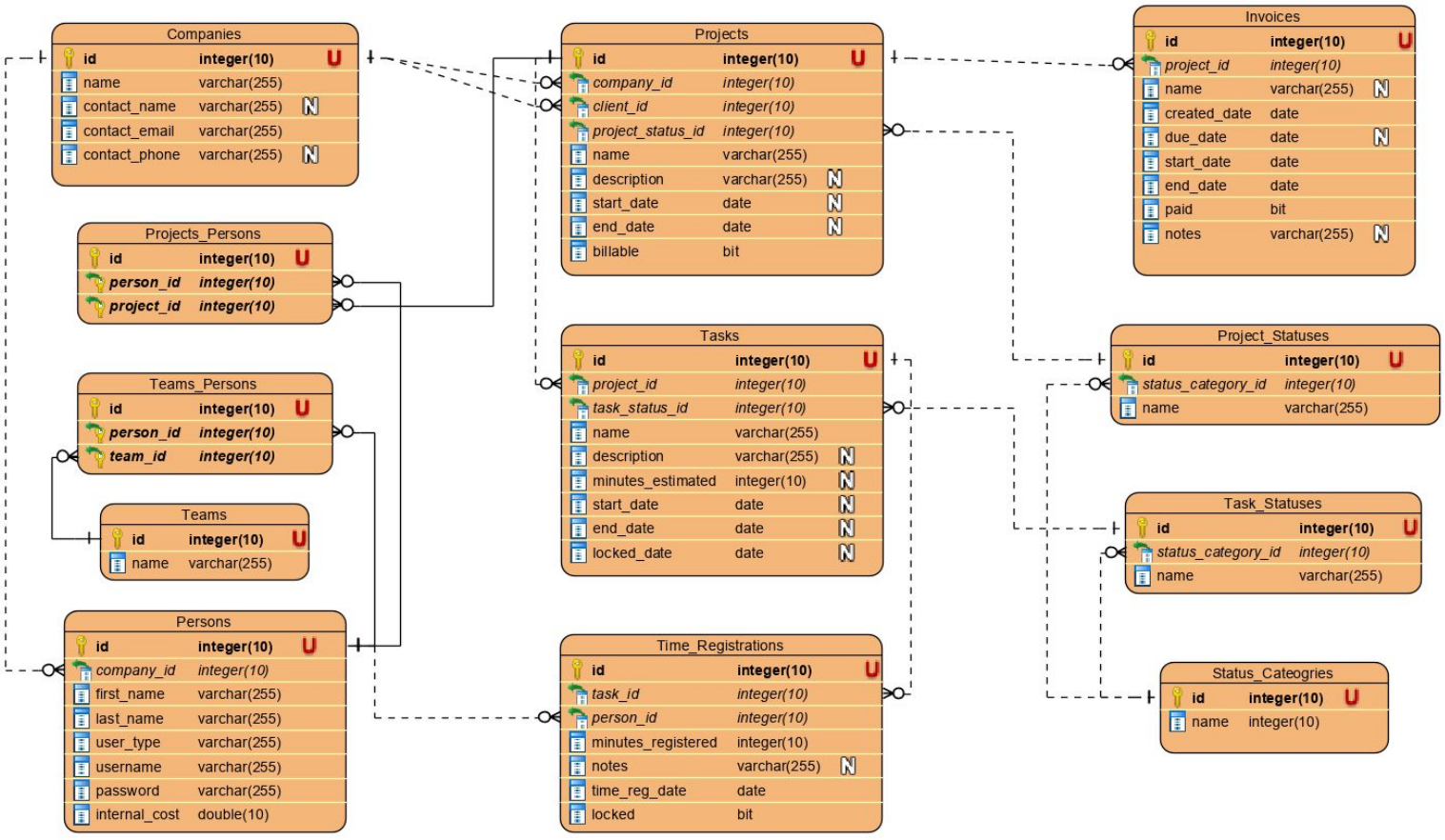

Fig.3 - Physical Data Model

2.2.1. 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:

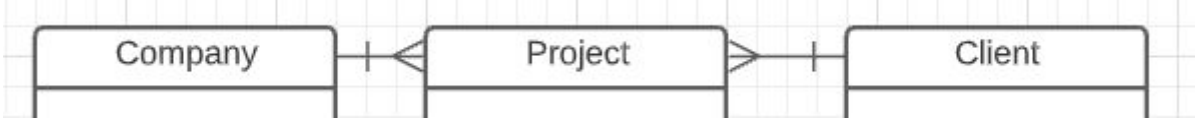

Fig.4 - Short entity diagram showing the initial structure of the project-company-client relationship

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*.

2.2.2 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 be able 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.

2.3 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.