UNESP – UNIVERSIDADE ESTADUAL PAULISTA
CAMPUS DE PRESIDENTE PRUDENTE
FACULDADE DE CIÊNCIAS E TECNOLOGIA - FCT

Giovanna Carreira Marinho, 181250951

Marcelo Eduardo Rodrigues da Silva Filho, 181251973

Rafael Araujo Chinaglia, 181253992

Eventum

Trabalho da disciplina Banco de Dados I e II do curso de Ciência da Computação da FCT/UNESP.

Prof. Ronaldo Celso Messias Correia.

Presidente Prudente

2021

Índice

Especificação do problema

O meio acadêmico é permeado de vários eventos, indo desde congressos, convenções e seminários até saraus, eventos culturais e desportivos, e para gerenciar as inscrições, vagas e pagamentos, muitas vezes se faz necessário mobilizar pessoas e entidades para cuidar manualmente deste processo, o que gera confusões nos controles, dados, confirmações de pagamentos e controle de caixa.

Uma aplicação que gerencie as inscrições desses eventos se mostra, portanto, uma ótima tática para aliviar o trabalho manual com processos repetitivos e que podem ser automatizados. Com a aplicação, remove-se completamente a necessidade de empresas e aplicações terceiras para o controle das inscrições, o que poderia acarretar em taxas e aumentos nos valores de inscrição, utilizando o conhecimento produzido dentro da universidade para servir toda a sociedade com tecnologia de livre acesso e qualidade. Além disso, um dos maiores problemas com os costumeiros sistemas criados como projetos dentro da universidade é o abandono dos projetos após a formação dos alunos responsáveis, deixando um sistema que muitas vezes carece de boa documentação e de manutenção para continuar relevante e útil.

Dessa forma, se faz necessário o desenvolvimento de uma aplicação cujo objetivo seja prover aos usuários do sistema um ambiente de fácil uso, de forma que consigam gerenciar seus eventos criados ou inscritos. Além disso, uma boa documentação e implementação se faz necessária para que o mesmo possa sofrer manutenções a qualquer momento por outros desenvolvedores e que o sistema não fique inativo.

A aplicação Eventum deverá ser capaz de registrar usuários com seus dados pessoais, bem como eventos que contenham atividades ligadas a ele. Cada atividade, por sua vez, deve ter os dados de data e horário, descrição, custo e demais detalhes importantes para descrevê-las no processo de inscrição. A inscrição, que será uma relação entre os usuários e os eventos, também precisarão registrar detalhadamente quais atividades de um dado evento o usuário optou por inscrever-se.

Sendo assim, podemos definir as seguintes entidades:

- *User*, representa o usuário do sistema, cujos atributos são: *id, user_name, first_name, last_name, email, password, phone, address* (dado pelos atributos *street, house_number, district, complement, CEP*), *CPF, is_staff, is_active, is_trusty, date_joined*; Esta entidade terá uma especialização do tipo total com restrição de sobreposição, sendo que as entidades de nível inferior serão:
    - *Organizer*, representa um usuário do tipo organizador do evento, dessa forma, seus atributos, além daqueles de *User*, serão: *link*.
    - *Participant*, representa um usuário do tipo participante do evento, dessa forma, seus atributos, além daqueles de *User*, serão: *profile_picture*.
- *Event*, representa o evento do sistema, cujos atributos são: *id, title, start_datetime, end_datetime, activity_limit*;
- *Activity*, representa cada atividade que compõem um evento, cujos atributos são: *id, title, location, start_datetime, end_datetime, cost, description, subscription_limit*;

- *Subscription*, representa a inscrição de um participante em um evento, a escolha de tipo de inscrição (entrada normal, meia-entrada, etc) e as atividades nas quais ele optou inscrever-se. Possui os atributos: *id, date, cost.*
- *Subscription_type*, que representa o tipo de inscrição escolhido para o evento. Possui os atributos: *id, description, cost.*

Dessa forma, podemos definir os seguintes relacionamentos entre essas entidades:

- Um evento pode possuir várias atividades e uma atividade pertence a um evento: relacionamento 1:N **Has** entre *Event* e *Activity.*
- Um organizador pode organizar vários eventos e um evento pode ser organizado por vários organizadores: relacionamento N:N **Organize** entre *Organizer* e *Event*.
- Um participante pode fazer várias inscrições e uma inscrição pertence a um participante: relacionamento 1:N **Has** entre *Participant* e *Subscription.*
- Cada inscrição pode conter várias atividades e cada atividade pode estar contida em várias inscrições: relacionamento N:N **Has** entre *Subscription* e *Activity*.
- Uma inscrição contém um tipo de inscrição, e cada tipo pode estar contido em várias inscrições: relacionamento N:1 **Has** entre *Subscription* e *Subscription_type*.
- Uma inscrição referencia um evento e cada evento pode ser referenciado por várias inscrições: relacionamento N:1 **Reference** entre *Subscription* e *Event*, com o atributo *cost* para definir o histórico do custo no ato na inscrição.
- Um evento possui várias formas de inscrição, e uma forma de inscrição possui um evento: 1:N **Has** entre *Event* e *Subscription_type.*

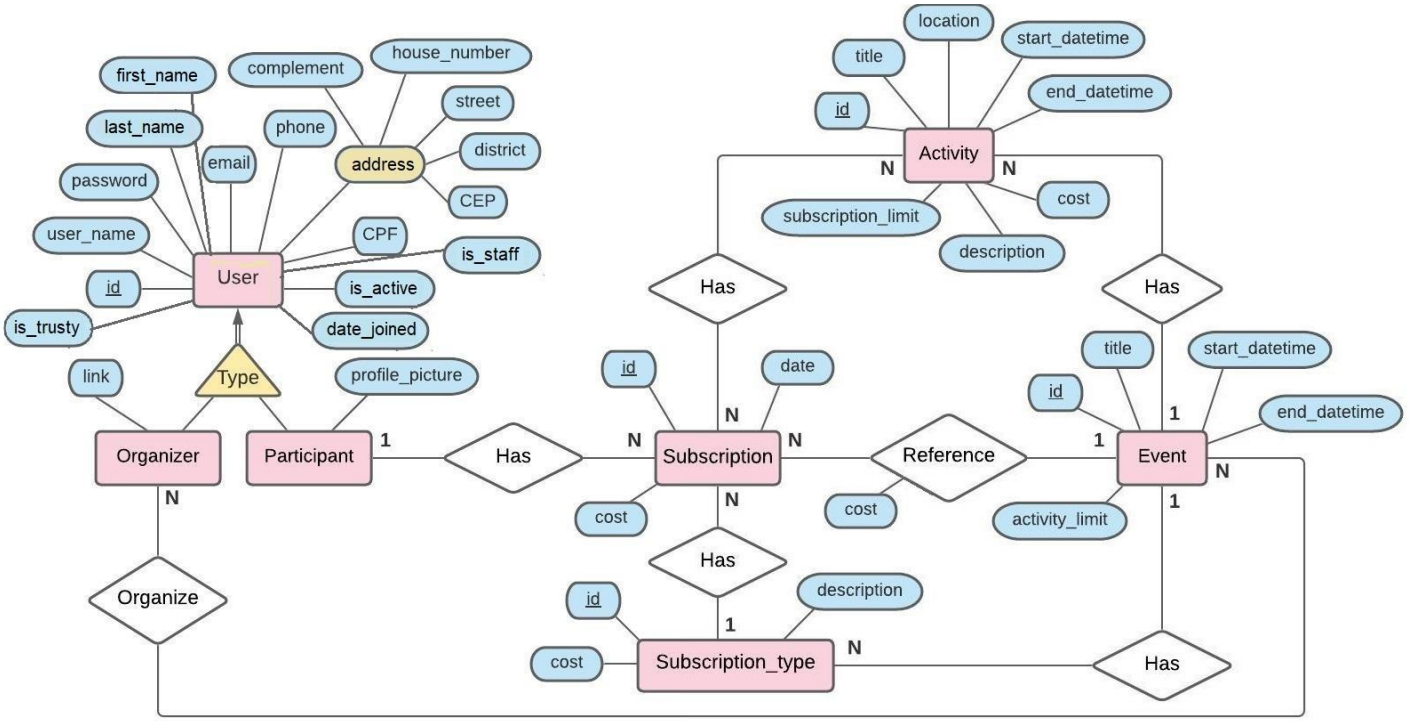
Por fim, podemos definir as seguintes consultas:

- Listar eventos disponíveis;
- Listar eventos que estão ocorrendo em um determinado período;
- Listar eventos de um determinado organizador;
- Listar eventos que um usuário está inscrito;
- Listar participantes de um evento;
- Listar atividades dos eventos que um participante está participando;
- Listar participantes de uma atividade;
- Verificar histórico de eventos participados;

Esquema Conceitual

Uma vez definidos os objetivos do sistema, assim como as entidades, atributos e relacionamentos envolvidos na modelagem, podemos obter o Modelo Entidade-Relacionamento apresentado na Figura abaixo.

Figura 1: Modelo Entidade-Relacionamento.



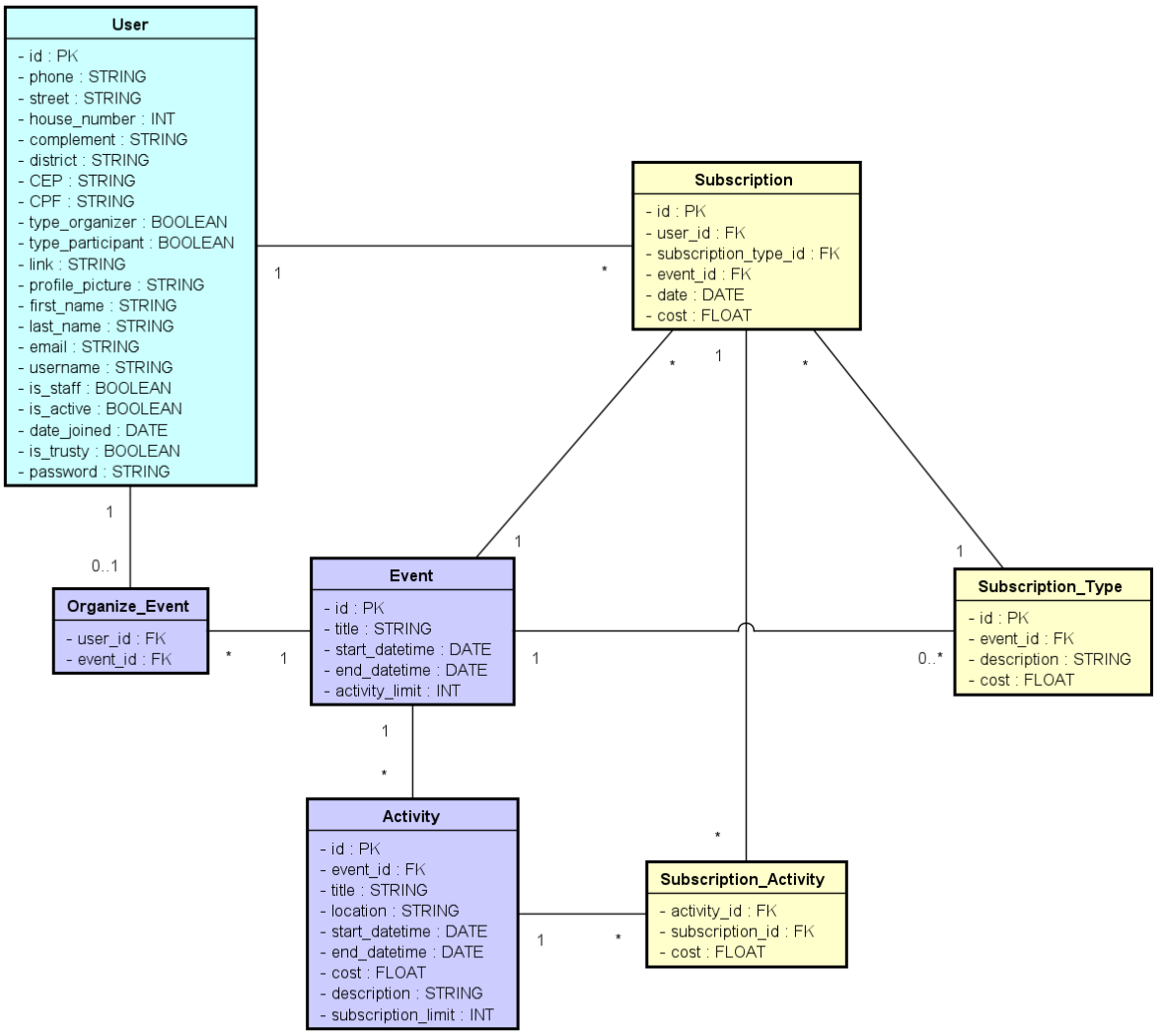
Esquema Relacional

O mapeamento do Modelo Entidade-Relacionamento para o modelo Relacional resultaram nas seguintes relações:

- User(id, phone, street, house_number, district, complement, CEP, CPF, type_organizer, link, type_participant, profile_picture, first_name, last_name, username, password, email, is_staff, is_active, date_joined, is_trusty);
- Event(id, title, start_datetime, end_datetime, activity_limit);
- Activity(id, event_id (FK), title, location, start_datetime, end_datetime, cost, description, subscription_limit);
- Subscription(id, user_id (FK), subscription_type_id (FK), event_id (FK), date, cost);
- Subscription_type(id, event_id (FK), description, cost);
- Organize_Event(user_id (FK), event_id (FK));
- Subscription_Activity(subscription_id (FK), activity_id (FK), cost);

Com isso, o seguinte esquema relacional pode ser criado:

Figura 2: Diagrama para o Modelo Relacional.



Normalização

- **Primeira forma normal (1FN):** todas as relações encontram-se na 1NF, já que todos os atributos são atômicos e monovalorados;
- **Segunda forma normal (2FN):** as relações com chave primária composta tem todos os seus atributos não-chave dependentes de todos os atributos que compõem a chave primária. Dessa forma, todas as relações encontram-se na 2NF;
- **Terceira forma normal (3FN):** todas as relações que estão na 3NF, não tem nenhum atributo não chave de nenhuma das relações depende de outro atributo não chave.

Especificação de Consultas em Álgebra Relacional e SQL

Vale destacar que valores no formato `:variavel` representam dados de entrada.

SQL	Álgebra Relacional
Listar eventos disponíveis	
`SELECT * FROM event WHERE event.end_datetime > NOW`	$\sigma_{\text{end\_datetime} > \text{NOW}}(\text{event})$
Listar eventos que estão ocorrendo em um determinado período	
`SELECT * FROM event WHERE event.start_datetime >= :start AND event.end_datetime <= :end`	$\sigma_{\text{start\_datetime} >= :\text{start} \wedge \text{end\_datetime} <= :\text{end}}(\text{event})$
Listar eventos de um determinado organizador	
`SELECT * FROM event LEFT JOIN organize_event on (organize_event.event_id = event.id AND organize_event.user_id = :organizer_id)`	$\sigma_{\text{organize\_event.event\_id} = \text{event.id} \wedge \text{organize\_event.user\_id} = :\text{organizer\_id}}(\text{event} \times \text{organize\_event})$
Listar eventos que um usuário está inscrito	
`SELECT * FROM event LEFT JOIN subscription on (subscription.event_id = event.id AND subscription.user_id = :participant_id)`	$\sigma_{\text{subscription.event\_id} = \text{event.id} \wedge \text{subscription.user\_id} = :\text{participant\_id}}(\text{event} \times \text{subscription})$
Listar participantes de um evento	
`SELECT * FROM user LEFT JOIN subscription on (user.id = subscription.user_id) WHERE subscription.event_id = :event_id`	$\sigma_{\text{user.id} = \text{subscription.user\_id} \wedge \text{subscription.event\_id} = :\text{event\_id}}(\text{event} \times \text{subscription})$
Listar participantes de uma atividade	
`SELECT user.id, user.show_name, user.profile_picture FROM user LEFT JOIN subscription on (subscription.user_id = user.id) LEFT JOIN subscription_activity on (subscription_activity.subscription_id = subscription.id) WHERE subscription_activity.activity_id = :activity_id AND user.active = 1`	$\pi_{\text{user.id, user.show\_name, user.profile\_picture}}(\sigma_{\text{subscription.user\_id} = \text{user.id} \wedge \text{subscription\_activity.activity\_id} = :\text{activity\_id} \wedge \text{subscription\_activity.subscription\_id} = \text{subscription.id}}(\text{user} \times \text{subscription} \times \text{subscription\_activity}))$

Listar atividades dos eventos que um participante está participando	
SELECT activity.id, activity.title, activity.location, activity.start_datetime, activity.end_datetime, activity.description, activity.subscription_limit, subscription_activity.cost, event.id, event.title FROM activity LEFT JOIN subscription on (subscription.user_id = :user_id) LEFT JOIN subscription_activity on (subscription_activity.activity_id = activity.id AND subscription_activity.subscription_id = subscription.id) LEFT JOIN event on (event.id = subscription.event_id) ORDER BY event.id	$\pi_{\text{activity.id, activity.title, activity.location, activity.start\_datetime, activity.end\_datetime, activity.description, activity.subscription\_limit, subscription\_activity.cost, event.id, event.title}}(\sigma_{\text{subscription.user\_id} = :\text{user\_id} \wedge \text{subscription\_activity.activity\_id} = \text{activity.id} \wedge \text{subscription\_activity.subscription\_id} = \text{subscription.id} \wedge \text{event.id} = \text{subscription.event\_id}}(\text{activity} \times \text{subscription} \times \text{subscription\_activity} \times \text{event}))$
Verificar histórico de eventos participados	
SELECT event.id, event.title, event.start_datetime, event.end_datetime, subscription.cost, subscription.date, subscription_type.description, (SELECT count(subscription_activity.activity_id) FROM subscription_activity WHERE subscription_activity.subscription_id = subscription.id) as atividades_inscritas, FROM event INNER JOIN subscription on (subscription.event_id = event.id AND subscription.user_id = :user_id) LEFT JOIN subscription_type on (subscription_type.id = subscription.subscription_type_id)	$\pi_{\text{event.id, event.title, event.start\_datetime, event.end\_datetime, subscription.cost, subscription.date, subscription\_type.description,}}\rho(\text{atividades\_inscritas}, (\text{COUNT}(\sigma_{\text{subscription\_activity.subscription\_id} = \text{subscription.id}}(\text{subscription\_activity}))(\sigma_{\text{subscription.user\_id} = :\text{user\_id} \wedge \text{subscription.event\_id} = \text{event.id} \wedge \text{subscription\_type.id} = \text{subscription.subscription\_type\_id}}(\text{event} \times \text{subscription} \times \text{subscription\_type}))$

Implementação da base de dados

```
CREATE SCHEMA IF NOT EXISTS Eventum DEFAULT CHARACTER SET utf8;
USE Eventum;

CREATE TABLE IF NOT EXISTS Eventum.user (
  id INT NOT NULL AUTO_INCREMENT,
  user_name VARCHAR(60) NOT NULL,
  password VARCHAR(45) NOT NULL,
  show_name VARCHAR(60) NULL,
  email VARCHAR(45) NULL,
  phone VARCHAR(20) NULL,
  street VARCHAR(255) NULL,
  house_number INT NULL,
  complement VARCHAR(255) NULL,
  district VARCHAR(100) NULL,
  CEP VARCHAR(20) NULL,
  CPF VARCHAR(12) NULL,
  active TINYINT NULL DEFAULT 1,
  type_organizer TINYINT NULL DEFAULT 0,
  type_participant TINYINT NULL DEFAULT 0,
  link VARCHAR(255) NULL,
  profile_picture VARCHAR(255) NULL,
  PRIMARY KEY (id),
  UNIQUE INDEX id_UNIQUE (id ASC));

CREATE TABLE IF NOT EXISTS Eventum.event (
  id INT NOT NULL AUTO_INCREMENT,
  title VARCHAR(100) NOT NULL,
  start_datetime TIMESTAMP NOT NULL,
  end_datetime TIMESTAMP NOT NULL,
  activity_limit INT NULL,
  PRIMARY KEY (id),
  UNIQUE INDEX id_UNIQUE (id ASC));

CREATE TABLE IF NOT EXISTS Eventum.organize_event (
  user_id INT NOT NULL,
  event_id INT NOT NULL,
  PRIMARY KEY (user_id, event_id),
  INDEX fk_user_has_event_event1_idx (event_id ASC),
  INDEX fk_user_has_event_user_idx (user_id ASC),
  CONSTRAINT fk_user_has_event_user
    FOREIGN KEY (user_id)
    REFERENCES Eventum.user (id)
    ON DELETE NO ACTION
```

```
    ON UPDATE NO ACTION,
  CONSTRAINT fk_user_has_event_event1
    FOREIGN KEY (event_id)
    REFERENCES Eventum.event (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

CREATE TABLE IF NOT EXISTS Eventum.activity (
  id INT NOT NULL AUTO_INCREMENT,
  event_id INT NOT NULL,
  title VARCHAR(100) NOT NULL,
  location VARCHAR(255) NOT NULL,
  start_datetime TIMESTAMP NOT NULL,
  end_datetime TIMESTAMP NOT NULL,
  cost FLOAT NOT NULL DEFAULT 0,
  description VARCHAR(255) NULL,
  subscription_limit INT NULL,
  PRIMARY KEY (id),
  UNIQUE INDEX id_UNIQUE (id ASC),
  INDEX fk_activity_event1_idx (event_id ASC),
  CONSTRAINT fk_activity_event1
    FOREIGN KEY (event_id)
    REFERENCES Eventum.event (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

CREATE TABLE IF NOT EXISTS Eventum.subscription_type (
  id INT NOT NULL AUTO_INCREMENT,
  event_id INT NOT NULL,
  description VARCHAR(255) NOT NULL,
  cost FLOAT NULL DEFAULT 0,
  PRIMARY KEY (id),
  UNIQUE INDEX id_UNIQUE (id ASC),
  INDEX fk_subscription_type_event1_idx (event_id ASC),
  CONSTRAINT fk_subscription_type_event1
    FOREIGN KEY (event_id)
    REFERENCES Eventum.event (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

CREATE TABLE IF NOT EXISTS Eventum.subscription (
  id INT NOT NULL AUTO_INCREMENT,
  user_id INT NOT NULL,
  subscription_type_id INT NOT NULL,
  event_id INT NOT NULL,
  date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
```

```
  cost FLOAT NULL,
  PRIMARY KEY (id),
  INDEX fk_user_has_event_event2_idx (event_id ASC),
  INDEX fk_user_has_event_user1_idx (user_id ASC),
  UNIQUE INDEX id_UNIQUE (id ASC),
  INDEX fk_subscription_subscription_type1_idx (subscription_type_id
ASC),
  CONSTRAINT fk_user_has_event_user1
    FOREIGN KEY (user_id)
    REFERENCES Eventum.user (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT fk_user_has_event_event2
    FOREIGN KEY (event_id)
    REFERENCES Eventum.event (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT fk_subscription_subscription_type1
    FOREIGN KEY (subscription_type_id)
    REFERENCES Eventum.subscription_type (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);

CREATE TABLE IF NOT EXISTS Eventum.subscription_activity (
  activity_id INT NOT NULL,
  subscription_id INT NOT NULL,
  cost FLOAT NOT NULL,
  INDEX fk_activity_has_subscription_subscription1_idx
(subscription_id ASC),
  INDEX fk_activity_has_subscription_activity1_idx (activity_id
ASC),
  PRIMARY KEY (subscription_id, activity_id),
  CONSTRAINT fk_activity_has_subscription_activity1
    FOREIGN KEY (activity_id)
    REFERENCES Eventum.activity (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT fk_activity_has_subscription_subscription1
    FOREIGN KEY (subscription_id)
    REFERENCES Eventum.subscription (id)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);
```