Knex

Até agora não foi discutido nenhum mecanismo de persistência de dados para as APIs desenvolvidas. Utilizar a memória do processo é útil mas possui grandes limitações, sendo a mais óbvia delas a não-persistência entre reinicializações da sua aplicação, ou até mesmo o não compartilhamento entre os vários processos de um cluster.

Quando se analisa as possíveis escolhas para persistência de dados em um projeto, o modelo mais simples e difundido é o uso de uma base de dados relacional. O fato de possuir um esquema robusto, ser flexível, ter muito conhecimento difundido na Internet e escalar apropriadamente para a grande maioria dos cenários a qual é subjulgada faz desse tipo de base de dados a escolha padrão. É quando o cenário é específico demais, seja pelo volume de informações, modelo muito variado, características de acesso recursivas dentre outras peculiaridades que outros modelos como o baseado em documentos, chave-valor ou grafos surgem para atender a demanda.

Uma vez decidido o uso de uma base relacional, o próximo passo é escolher o sistema gerenciador de base de dados (MySQL, PostgreSQL, SQL Server etc.). Os fatores que permeiam essa decisão estão fora do escopo dessa disciplina, e não influenciam significativamente no passo seguinte. Para o decorrer da disciplina o SGBD PostgreSQL será utilizado, por conveniência.

Criação das tabelas no PostgreSQL

Antes de prosseguir, será necessário criar as tabelas que refletem o modelo da aplicação de gestão de tarefas que vem sendo desenvolvida até então. Além das tabelas de usuários e tarefas, serão criadas posteriormente tabelas de etiquetas e checklists, permitindo a análise de relacionamentos do tipo muitos para muitos e um para muitos.

O restante do material assume que você possui um servidor PostgreSQL versão 9 ou superior, e credenciais de acesso a ele. Recomenda-se o uso do cliente `pgAdmin`, mas não é uma exigência.

Crie primeiramente uma tabela de usuários, usando o seguinte comando SQL:

```
create sequence usuarios_id_seq;

create table usuarios (
  id int not null,
  login varchar(100) not null,
  senha bytea not null,

  constraint pk_usuarios primary key (id),
  constraint un_usuarios_login unique (login)
);
```

Crie agora a tabela de tarefas:

```
create sequence tarefas_id_seq;

create table tarefas (
  id int not null,
  usuario_id int not null,
  descricao varchar(100) not null,
  previsao timestamp not null,
  conclusao timestamp,
```

```
  constraint pk_tarefas primary key (id),
  constraint fk_tarefas_usuario foreign key (usuario_id) references usuarios (id)
);
```

Nota sobre armazenamento de hashes/GUIDs

Quando se discute sobre armazenamento de senhas, é senso comum *não armazená-las* como texto plano, pois isso é uma falha gravíssima de segurança. O que se discute com menos frequência é a técnica de *espalhamento/hash* usada para proteger o valor antes de armazená-lo, e que tipo de dado utilizar na base.

Com relação às possíveis técnicas, uma combinação bastante aceita é a seguinte: use uma função espalhadora forte, como a SHA-256, *repetidamente* sobre o valor de entrada *concatenado* com uma palavra segredo, chamada de *salt*. Por exemplo, se a senha do usuário é `123456`, você pode obter um hash seguro com esse código Node:

```
const crypto = require('crypto');

console.log(crypto.createHash('sha256')
    .update('123456')
    .digest());

console.log(crypto.createHash('sha256')
    .update('123456')
    .digest('hex'));

console.log(crypto.createHash('sha256')
    .update('123456')
    .digest('base64'));

const senha = '123456';
const salt = 'segredo';
let resultado = `${salt}${senha}`;
for (let i = 0; i < 10; i++) {
    resultado = crypto.createHash('sha256')
        .update(resultado)
        .digest();
}
console.log(resultado);
```

Isso produz um vetor de 32 bytes que pode ser armazenado no banco, mas como armazená-lo? Dentre os possíveis formatos, três se destacam:

- Uso de uma coluna de texto de 44 posições e armazenar o valor codificado como base64.
- Uso de uma coluna de texto de 64 posições e armazenar o valor codificado como hexadecimal.
- Uso de uma coluna de vetor de bytes de 32 posições e armazenar o valor sem codificação.

Normalmente se usa uma das primeiras duas opções, pois manipular cadeias de caracteres costuma ser mais conveniente e o ganho de performance/armazenamento não é tão expressivo, mas nesse material utilizaremos a terceira para ter uma visão diferente sobre o tema.

Driver, Query Builder ou ORM?

O último passo é decidir como interfacear com o banco de dados no código da sua aplicação. Neste ponto existem três categorias principais, cada uma com seus prós e contras. Vale notar que nenhuma opção é indiscutivelmente melhor que a outra, e é possível encontrar boas discussões na Internet sobre o assunto [1].

[1] https://blog.logrocket.com/why-you-should-avoid-orms-with-examples-in-node-js-e0baab73fa5/

Driver baixo nível

Para scripts simples, projetos muito pequenos ou que exijam máxima atenção com performance, a melhor opção pode ser usar uma biblioteca que simplesmente encapsula a conexão com o banco de dados, não ajudando em muito além disso.

Para exercitar esse modelo, implemente um script que cadastra usuários na base com uma senha pré-definida. Comece instalando no projeto o pacote `pg`:

```
npm install pg
```

E implemente um script com o seguinte código:

```
const crypto = require('crypto');
const { Client } = require('pg');
const client = new Client();

async function main() {

    await client.connect();

    const login = 'samuel';
    const senha = '123456';

    const salt = 'segredo';
    let senhaCriptografada = `${salt}${senha}`;
    for (let i = 0; i < 10; i++) {
        senhaCriptografada = crypto.createHash('sha256')
            .update(senhaCriptografada)
            .digest();
    }

    const res = await client.query(
        'insert into usuarios (id, login, senha) ' +
        'values (nextval(\'usuarios_id_seq\'), $1, $2)',
        [login, senhaCriptografada]);

    console.log(`Linhas alteradas: ${res.rowCount}`);
    await client.end();

}

main();
```

Ao tentar executar o script, a aplicação irá estourar com um erro de autenticação. Isso é natural, afinal não foi informada a ela as credenciais de acesso ao PostgreSQL. Existem duas maneiras de resolver isso. Uma delas é através do uso de variáveis de ambiente:

```
PGUSER=postgres PGDATABASE=db PGPASSWORD=postgres node script.js
```

Lembre-se de ajustar os valores conforme apropriado para o seu cenário.

A outra opção é passar as credenciais direto no script:

```
const client = new Client({
    database: 'db',
    user: 'postgres',
    password: 'postgres'
});
```

Cada projeto vai definir a estratégia que melhor lhe couber, mas recentemente tem-se dado preferência para variáveis de ambiente, pois são mais amigáveis com múltiplos ambientes e containerização.

Dica: use o comando SQL `select encode(senha, 'hex') from usuarios;` para obter uma versão legível da senha. `encode(senha, 'base64')` também é muito útil.

ORM

Com a popularização da programação orientada a objetos, naturalmente surgiram ideias para facilitar o uso de bases de dados relacionais neste meio. Uma das vertentes mais agressivas propõe o maior nível de abstração possível, e é chamada de ORM (mapeamento objeto-relacional). A ideia do ORM é configurar previamente um conjunto de classes (ou equivalente) e atributos de modo a representarem o modelo de tabelas/colunas do banco de dados em um grafo de objetos. Com base nessa configuração, a ferramenta de ORM é capaz de fornecer APIs de consulta e manipulação dos dados de maneira simplificada.

A primeira vista essa parece a melhor opção, mas como o mapeamento entre orientação a objetos e modelo de dados relacional é complexo e incompleto, o resultado quase sempre é o sacrifício de performance para o ganho de simplicidade e produtividade no desenvolvimento. Portanto o uso dessa técnica deve ser feito com cuidado, e ela nem de longe exime o desenvolvedor de possuir um conhecimento aprofundado em modelos relacionais.

Para entender o funcionamento de um ORM, com a finalidade de comparar com o uso do Query Builder Knex, será mostrado agora como desenvolver uma API capaz de autenticar usuários, listar e trazer detalhes de tarefas usando o ORM `Sequelize`.

Comece criando um novo projeto Express com autenticação e geração de token JWT:

```
npm init
npm i express express-jwt jsonwebtoken moment
```

```
const express = require('express');
const jwt = require('express-jwt');
const { SEGREDO_JWT } = require('./seguranca');
const loginRouter = require('./login/login-router');
const tarefasRouter = require('./tarefas/tarefas-router');

const app = express();

app.use(express.json());
```

```
app.use(jwt({
    secret: SEGREDO_JWT
}).unless({ path: '/login' }));

app.use('/login', loginRouter);
app.use('/tarefas', tarefasRouter);

app.listen(3000);
```

Crie o módulo `seguranca`:

```
module.exports.SEGREDO_JWT = 'ormsegredo';
```

Crie o roteador de login:

```
const express = require('express');
const jsonwebtoken = require('jsonwebtoken');
const moment = require('moment');
const loginService = require('./login-service');
const { SEGREDO_JWT } = require('../seguranca');


const router = express.Router();

router.post('/', (req, res) => {
    const { usuario, senha } = req.body;
    loginService.credenciaisValidas(usuario, senha)
        .then(validas => {
            if (validas) {
                const token = jsonwebtoken.sign({
                    sub: usuario,
                    exp: moment().add(30, 'minutes').unix()
                }, SEGREDO_JWT);
                res.send({ token });
            } else {
                res.status(401).send();
            }
        })
        .catch(err => {
            console.error(err);
            res.status(500).send();
        });
});

module.exports = router;
```

Agora o esqueleto do serviço de login:

```
module.exports.credenciaisValidas = (usuario, senha) => {
    return Promise.resolve(true); // será substituído pela integração com ORM
};
```

Crie também o router de tarefas:

```
const express = require('express');
const tarefasService = require('./tarefas-service');


const router = express.Router();

router.get('/', async (req, res) => {
    const usuario = req.user.sub;
    try {
        const tarefas = await tarefasService.listar(usuario);
        res.send(tarefas.map(tarefa => ({
            id: tarefa.id,
            descricao: tarefa.descricao,
            previsao: tarefa.previsao,
            conclusao: tarefa.conclusao
        })));
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

router.get('/:id', async (req, res) => {
    const usuario = req.user.sub;
    const id = req.params.id;
    try {
        const tarefa = await tarefasService.buscarPorId(id);
        if (!tarefa) {
            res.status(404).send();
        } else if (tarefa.usuario.login !== usuario) {
            res.status(403).send();
        } else {
            res.send({
                id: tarefa.id,
                descricao: tarefa.descricao,
                previsao: tarefa.previsao,
                conclusao: tarefa.conclusao
            });
        }
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});

module.exports = router;
```

E o esqueleto do service de tarefas:

```
const moment = require('moment');
```

```
module.exports.listar = usuario => {
    return Promise.resolve([{
        id: 1,
        descricao: 'Tarefa 1',
        previsao: moment(),
        conclusao: null,
        usuario: {
            login: 'samuel'
        }
    }]);
};


module.exports.buscarPorId = id => {
    return Promise.resolve({
        id,
        descricao: 'Tarefa 1',
        previsao: moment(),
        conclusao: null,
        usuario: {
            login: 'samuel'
        }
    });
};
```

Depois de garantir que essa base está funcionando corretamente, o próximo passo é configurar o `Sequelize` e ajustar os módulos de serviço para buscar os dados no banco.

Primeiro instale a biblioteca e o driver PostgreSQL usado por ela:

```
npm i sequelize pg pg-hstore
```

Agora crie um módulo chamado `orm` na raíz do projeto, com o seguinte conteúdo:

```
const Sequelize = require('sequelize');


module.exports.sequelize = new
Sequelize('postgres://postgres:postgres@localhost:5432/tarefas1', {
    define: {
        timestamps: false,
        underscored: true
    }
});
```

Lembre-se de ajustar a string de conexão conforme o seu ambiente.

Agora crie o módulo `usuarios/usuarios-modelo`:

```
const Sequelize = require('sequelize');
const { sequelize } = require('../orm');


module.exports.Usuario = sequelize.define('usuario', {
```

```
    login: {
        type: Sequelize.STRING,
        allowNull: false
    },
    senha: {
        type: Sequelize.BLOB,
        allowNull: false
    }
});
```

E o módulo `tarefas/tarefas-modelo`:

```
const Sequelize = require('sequelize');
const { sequelize } = require('../orm');
const { Usuario } = require('../usuarios/usuarios-modelo');


const Tarefa = sequelize.define('tarefa', {
    descricao: {
        type: Sequelize.STRING,
        allowNull: false
    },
    previsao: {
        type: Sequelize.DATE,
        allowNull: false
    },
    conclusao: {
        type: Sequelize.DATE
    }
});
Tarefa.belongsTo(Usuario);

module.exports.Tarefa = Tarefa;
```

Por fim adapte os serviços de login e tarefas:

```
const crypto = require('crypto');
const { Usuario } = require('../usuarios/usuarios-modelo');


const criptografar = senha => {
    const salt = 'segredo';
    let senhaCriptografada = `${salt}${senha}`;
    for (let i = 0; i < 10; i++) {
        senhaCriptografada = crypto.createHash('sha256')
            .update(senhaCriptografada)
            .digest();
    }
    return senhaCriptografada;
};


module.exports.credenciaisValidas = async (login, senha) => {
```

```
    const usuario = await Usuario.findOne({
        where: { login, senha: criptografar(senha) }
    });
    return !!usuario;
};
```

```
const { Tarefa } = require('./tarefas-modelo');
const { Usuario } = require('../usuarios/usuarios-modelo');


module.exports.listar = usuario => {
    return Tarefa.findAll({
        include: [{
            model: Usuario,
            where: { login: usuario }
        }]
    });
};


module.exports.buscarPorId = id => {
    return Tarefa.findByPk(id, {
        include: [{
            model: Usuario
        }]
    });
};
```

Query Builder

A terceira opção que será apresentada para comunicação com o banco de dados é um intermediário entre as outras duas. Ao invés de escrever os comandos SQL manualmente, como na primeira opção, ou configurar uma camada de abstração orientada a objetos, como na segunda, existem bibliotecas que ajudam a montar dinamicamente os comandos SQL sem remover a essência relacional do processo. Esse é o fundamento por tras dos Query Builders, como o Knex.

Para demonstrar o uso dessa biblioteca, serão implementados endpoints para cadastro de tarefa (com suporte para definição de conjunto etiquetas) e para listar as etiquetas de uma determinada tarefa.

O primeiro passo é criar as tabelas que darão suporte para as etiquetas:

```
create table etiquetas (
    id int not null,
    descricao varchar(100) not null,

    constraint pk_etiquetas primary key (id),
    constraint un_etiquetas_descricao unique (descricao)
);
insert into etiquetas (id, descricao) values (1, 'Casa'), (2, 'Trabalho');

create table tarefa_etiqueta (
    tarefa_id int not null,
    etiqueta_id int not null,
```

```
    constraint pk_tarefa_etiqueta primary key (tarefa_id, etiqueta_id),
    constraint fk_tarefa_etiqueta_tarefa
      foreign key (tarefa_id)
      references tarefas (id),
    constraint fk_tarefa_etiqueta_etiqueta
      foreign key (etiqueta_id)
      references etiquetas (id)
);
```

Instale o Knex no projeto e um driver Postgres (no caso já estará instalado, mas também é uma dependência do Knex e teria que ser instalado manualmente caso o projeto não estivesse pré-configurado com o Sequelize):

```
npm i knex pg
```

Agora crie um módulo chamado `querybuilder` na raíz, com o seguinte código:

```
module.exports.knex = require('knex')({
    client: 'pg',
    connection: 'postgres://postgres:postgres@localhost:5432/tarefas1',
    debug: true
});
```

Novamente, atente-se em ajustar a string de conexão conforme o seu ambiente.

Adicione agora um método no módulo `tarefas-service` capaz de cadastrar uma tarefa:

```
const { knex } = require('../querybuilder');


module.exports.cadastrar = async (tarefa, usuario) => {
    return knex('tarefas')
        .insert({
            id: knex.raw('nextval(\'tarefas_id_seq\')'),
            descricao: tarefa.descricao,
            previsao: tarefa.previsao,
            usuario_id: knex('usuarios').select('id').where('login', usuario)
        })
        .returning('id')
        .then(x => x[0]); // .first() não pode ser usado em inserts
};
```

E também um endpoint no `tarefas-router`:

```
router.post('/', async (req, res) => {
    const usuario = req.user.sub;
    const tarefa = req.body;
    try {
        const id = await tarefasService.cadastrar(tarefa, usuario);
        res.send({ id });
    } catch (err) {
        console.error(err);
```

```
        res.status(500).send();
    }
});
```

O próximo passo é receber e tratar um conjunto de etiquetas. Note que apenas os identificadores das etiquetas são necessários. Faça o seguinte ajuste no módulo `tarefas-service`:

```
module.exports.cadastrar = async (tarefa, usuario) => {

    const id = await knex('tarefas')
        .insert({
            id: knex.raw('nextval(\'tarefas_id_seq\')'),
            descricao: tarefa.descricao,
            previsao: tarefa.previsao,
            usuario_id: knex('usuarios').select('id').where('login', usuario)
        })
        .returning('id')
        .then(x => x[0]); // .first() não pode ser usado em inserts

    if (tarefa.etiquetas) {
        tarefa.etiquetas.forEach(etiqueta => {
            knex('tarefa_etiqueta')
                .insert({
                    tarefa_id: id, etiqueta_id: etiqueta
                }).then();
        });
    }

    return id;
};
```

A primeira vista parece uma implementação adequada, mas ela possui uma série de problemas. O primeiro deles é que a requisição está retornando *antes* da execução dos inserts na tabela de etiquetas. Isso pode acarretar em condição de corrida com o front-end. Para resolver este ponto existem pelo menos duas opções.

A primeira delas é agregar cada promessa de inserção de etiqueta e usar a promessa combinada como retorno da função original. Veja:

```
if (tarefa.etiquetas) {
    await Promise.all(
        tarefa.etiquetas.map(etiqueta =>
            knex('tarefa_etiqueta').insert({
                tarefa_id: id, etiqueta_id: etiqueta
            })));
}
```

A segunda é usar o utilitário `batchInsert` do próprio knex:

```
await knex.batchInsert('tarefa_etiqueta', tarefa.etiquetas.map(x => ({
    tarefa_id: id,
    etiqueta_id: x
})));
```

O segundo problema com essa implementação é a ausência de controle transacional. O que acontece se você cadastrar uma tarefa com um identificador de etiqueta inexistente ou duplicado? A inclusão desse registro vai falhar, *mas* a tarefa já foi cadastrada e permanecerá no banco, o que dificilmente é o que você, ou seu usuário, deseja.

O knex oferece controle transacional através do método `knex.transaction`. Esse método recebe uma função como parâmetro, que é chamada com uma instância de transação. Essa instância pode então ser usada para construir queries ou ser passada no método `transacting` de queries construídas através do objeto `knex`. Veja:

```
module.exports.cadastrar = async (tarefa, usuario) => {
    return knex.transaction(async trx => {
        const id = await knex('tarefas')
            .transacting(trx)
            .insert({
                id: knex.raw('nextval(\'tarefas_id_seq\')'),
                descricao: tarefa.descricao,
                previsao: tarefa.previsao,
                usuario_id: knex('usuarios').select('id').where('login', usuario)
            })
            .returning('id')
            .then(x => x[0]); // .first() não pode ser usado em inserts

        if (tarefa.etiquetas) {
            await knex
                .batchInsert('tarefa_etiqueta', tarefa.etiquetas.map(x => ({
                    tarefa_id: id,
                    etiqueta_id: x
                })))
                .transacting(trx);
        }

        return id;
    });
};
```

Ao invés de chamar a função `transacting`, o próprio objeto `trx` pode ser usado no lugar do `knex`:

```
const id = await trx('tarefas')
    .insert({
        id: knex.raw('nextval(\'tarefas_id_seq\')'),
        descricao: tarefa.descricao,
        previsao: tarefa.previsao,
        usuario_id: knex('usuarios').select('id').where('login', usuario)
    })
    .returning('id')
    .then(x => x[0]); // .first() não pode ser usado em inserts
```

O commit ocorre quando a Promise passada para a função `knex.transaction` resolve com sucesso, enquanto o rollback ocorre no caso de erro nessa promessa. É muito importante garantir a propagação correta da transação, pois qualquer chamada no objeto `knex` sem passar a transação será executada em um contexto transacional separado e poderá resultar em corrupção no estado da base.

Para concluir essa seção, crie um método que busca uma lista de etiquetas dado um identificador de tarefa e um usuário, no módulo `tarefas-service`:

```
module.exports.buscarEtiquetas = (idTarefa, usuario) => {
    return knex('etiquetas')
        .join('tarefa_etiqueta', 'etiquetas.id', 'tarefa_etiqueta.etiqueta_id')
        .join('tarefas', 'tarefas.id', 'tarefa_etiqueta.tarefa_id')
        .join('usuarios', 'usuarios.id', 'tarefas.usuario_id')
        .select('etiquetas.id', 'etiquetas.descricao')
        .where({
            'tarefa_etiqueta.tarefa_id': idTarefa,
            'usuarios.login': usuario
        });
};
```

E exponha essa funcionalidade em um novo endpoint no `tarefas-router`:

```
router.get('/:id/etiquetas', async (req, res) => {
    const usuario = req.user.sub;
    const idTarefa = req.params.id;
    try {
        const etiquetas = await tarefasService.buscarEtiquetas(idTarefa, usuario);
        res.send(etiquetas);
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});
```

Evolução do modelo

Suponha que sua aplicação está em produção, sendo utilizada por vários usuários. Em determinado momento você decide evoluí-la, adicionando uma funcionalidade de checklists dentro de cada tarefa. Isso vai significar, em algum momento, a adição de uma nova tabela no banco de dados. Como gerenciar esse tipo de evolução? "Lembrar" de executar scripts SQL é suscetível a erros entre ambientes, e a última coisa que você deseja é diferenças entre os ambientes de desenvolvimento/testes e produção.

Para resolver este ponto existe o conceito de "migrações de modelo de dados". Basicamente cada "migração" é um script (que pode ser SQL, ou código escrito em alguma biblioteca de interface com o banco como o próprio Knex) que atualiza o banco de dados para que ele fique compatível com a versão mais recente do código fonte. Essa migração pode ser disparada via CLI ou por dentro do código da própria aplicação. O importante é que ela faça parte do processo de integração/entrega contínua sem exigir nenhum tipo de intervenção humana.

Nesta seção abordaremos a solução de migração oferecida pela ferramenta `db-migrate`. Note que o próprio Knex possui uma solução nativa de mesma categoria (as duas são muito similares, a opção por mostrar o `db-migrate` foi tomada para apresentar uma ferramenta ligeiramente mais genérica que a embutida no Knex).

Comece removendo toda a estrutura de tabelas e sequências existente no banco de dados (visto que agora elas serão criadas através de migrações):

```
drop table tarefa_etiqueta;
drop table etiquetas;
drop table tarefas;
drop table usuarios;
drop sequence usuarios_id_seq;
drop sequence tarefas_id_seq;
```

Instale agora o CLI `db-migrate` de modo global, juntamente com o driver para Postgres:

```
npm i -g db-migrate db-migrate-pg
```

Crie um arquivo chamado `database.json`, usado pelo `db-migrate` para obter dados de conexão com o banco de dados:

```
{
  "dev": "postgres://postgres:postgres@localhost:5432/tarefas1"
}
```

Crie o primeiro script de migração com o seguinte comando:

```
db-migrate create usuarios
```

Ajuste o arquivo criado na pasta `migrations` do projeto com esse conteúdo nos métodos `up` e `down`:

```
exports.up = function(db) {
  return db.createTable('usuarios', {
    id: { type: 'int', primaryKey: true },
    login: { type: 'string', length: 100, notNull: true, unique: true },
    senha: { type: 'blob', notNull: true }
  });
};

exports.down = function(db) {
  return db.dropTable('usuarios');
};
```

Para executar a migração, execute o comando `db-migrate up`. Para revertê-la, use o comando `db-migrate down`.

Ainda relacionado aos usuários existe a sequência para o ID. Infelizmente o `db-migrate` não oferece suporte para essa estrutura (nem todos os SGBDs possuem suporte para sequências), e nesses casos a saída é usar o `runSql`:

```
exports.up = function(db) {
  return db
    .createTable('usuarios', {
      id: { type: 'int', primaryKey: true },
      login: { type: 'string', length: 100, notNull: true, unique: true },
      senha: { type: 'blob', notNull: true }
    })
    .then(() => db.runSql('create sequence usuarios_id_seq;'));
};
```

```
exports.down = function(db) {
  return db
    .runSql('drop sequence usuarios_id_seq;')
    .then(() => db.dropTable('usuarios'));
};
```

Uma outra opção, mais flexível, é usar scripts SQL ao invés de código JavaScript para elaborar as migrações. Isso é possível passando o switch `--sql-file` para o comando `db-migrate create`:

```
db-migrate create tarefas --sql-file
```

Note que foi criado uma migração `up` e `down` em uma pasta chamada `sqls`. Implemente-as, respectivamente, da seguinte maneira:

```
create sequence tarefas_id_seq;

create table tarefas (
  id int not null,
  usuario_id int not null,
  descricao varchar(100) not null,
  previsao timestamp not null,
  conclusao timestamp,

  constraint pk_tarefas primary key (id),
  constraint fk_tarefas_usuario foreign key (usuario_id) references usuarios (id)
);

create table etiquetas (
    id int not null,
    descricao varchar(100) not null,

    constraint pk_etiquetas primary key (id),
    constraint un_etiquetas_descricao unique (descricao)
);
insert into etiquetas (id, descricao) values (1, 'Casa'), (2, 'Trabalho');

create table tarefa_etiqueta (
    tarefa_id int not null,
    etiqueta_id int not null,

    constraint pk_tarefa_etiqueta primary key (tarefa_id, etiqueta_id),
    constraint fk_tarefa_etiqueta_tarefa
      foreign key (tarefa_id)
      references tarefas (id),
    constraint fk_tarefa_etiqueta_etiqueta
      foreign key (etiqueta_id)
      references etiquetas (id)
);
```

```
drop table tarefa_etiqueta;
drop table etiquetas;
```

```
drop table tarefas;
drop sequence tarefas_id_seq;
```

As duas maneiras de elaborar os scripts de migração são úteis, cabendo ao projeto definir um padrão, e usando a alternativa quando ela for mais apropriada. Uma consideração importante é que alguns projetos decidem *não* oferecer suporte para rollback (o método `down` das migrações). Garantir a qualidade desses módulos é custoso e eles praticamente nunca são usados, portanto o argumento principal é não implementá-los e atuar nos rollbacks manualmente quando algum imprevisto ocorrer nos ambientes produtivos.

Implementando suporte para checklists

Uma vez com a estrutura de migrações configurada e funcional, crie uma nova migração para adicionar a tabela de checklists:

```
db-migrate create checklists --sql-file
```

```
create table checklists (
    id uuid not null,
    tarefa_id int not null,
    descricao varchar(100) not null,

    constraint pk_checklists primary key (id),
    constraint fk_checklists_tarefa foreign key (tarefa_id) references tarefas (id)
);

create table items_checklist (
    id uuid not null,
    checklist_id uuid not null,
    descricao varchar(100) not null,
    completado boolean default false,

    constraint pk_items_checklist primary key (id),
    constraint fk_items_checklist_checklist
      foreign key (checklist_id)
      references checklists (id)
);
```

```
drop table items_checklist;
drop table checklists;
```

Note que dessa vez foram utilizadas colunas do tipo `uuid` ao invés de inteiros com auto-incremento. Essa é uma alternativa interessante para identificação única global de registros, amigável com arquiteturas compostas por várias fontes de dados distintas (o UUID pode ser gerado em qualquer lugar e distribuído para todas as fontes de dados envolvidas).

A próxima decisão a ser tomada é como desenhar os endpoints que manipulam os checklists. Existem basicamente dois extremos:

1) Implementar toda a gestão de checklists nos endpoints de cadastro e alteração de tarefa; 2) Implementar endpoints específicos para criar/alterar/remover um checklist e criar/alterar/remover itens de checklist.

A usabilidade desejada para a aplicação irá ditar o melhor caminho a ser seguido. Neste material implementaremos a primeira, e depois discutiremos como seria a implementação da segunda.

Primeiramente ajuste o método `cadastrar` no módulo `tarefas-service` para suportar a criação de checklists:

```
module.exports.cadastrar = async (tarefa, usuario) => {
    return await knex.transaction(async trx => {
        const id = await inserirTarefa(trx, tarefa, usuario);
        await inserirEtiquetas(trx, id, tarefa);
        await inserirChecklists(trx, id, tarefa);
        return id;
    });
};


function inserirTarefa(trx, tarefa, usuario) {
    return trx('tarefas')
        .insert({
            id: knex.raw('nextval(\'tarefas_id_seq\')'),
            descricao: tarefa.descricao,
            previsao: tarefa.previsao,
            usuario_id: knex('usuarios').select('id').where('login', usuario)
        })
        .returning('id')
        .then(x => x[0]); // .first() não pode ser usado em inserts
}


function inserirEtiquetas(trx, idTarefa, tarefa) {
    if (tarefa.etiquetas) {
        return trx.batchInsert('tarefa_etiqueta', tarefa.etiquetas.map(x => ({
            tarefa_id: idTarefa,
            etiqueta_id: x
        })));
    } else {
        return Promise.resolve();
    }
}


function inserirChecklists(trx, idTarefa, tarefa) {
    if (tarefa.checklists) {
        return Promise.all(tarefa.checklists.map(async checklist => {
            const idChecklist = uuidv4();
            await trx('checklists').insert({
                id: idChecklist,
                tarefa_id: idTarefa,
                descricao: checklist.descricao
            });
            await inserirItemsChecklist(trx, idChecklist, checklist);
        }));
    } else {
```

```
        return Promise.resolve();
    }
}


function inserirItemsChecklist(trx, idChecklist, checklist) {
    if (checklist.items) {
        return trx.batchInsert('items_checklist', checklist.items.map(x => ({
            id: uuidv4(),
            checklist_id: idChecklist,
            descricao: x.descricao,
            completado: x.completado
        })));
    } else {
        return Promise.resolve();
    }
}
```

Note que o pacote `uuid/v4` não é nativo do Node. Para instalá-lo use o seguinte comando:

```
npm install uuid
```

Repare também que o método todo foi refatorado, para fins de melhorar a legibilidade.

Crie agora um método no service para retornar a lista de checklists, juntamente com seus itens, dado um identificador de tarefa:

```
module.exports.buscarChecklists = (idTarefa, usuario) => {
    return knex('checklists')
        .leftJoin('items_checklist', 'items_checklist.checklist_id', 'checklists.id')
        .join('tarefas', 'tarefas.id', 'checklists.tarefa_id')
        .join('usuarios', 'usuarios.id', 'tarefas.usuario_id')
        .select(
            knex.ref('checklists.id').as('checkId'),
            knex.ref('checklists.descricao').as('checkDesc'),
            knex.ref('items_checklist.id').as('itemId'),
            knex.ref('items_checklist.descricao').as('itemDesc'),
            knex.ref('items_checklist.completado').as('itemCompl')
        )
        .where({
            'checklists.tarefa_id': idTarefa,
            'usuarios.login': usuario
        })
        .then(async res => {
            const checklists = {};
            res.forEach(linha => {
                if (!checklists[linha.checkId]) {
                    checklists[linha.checkId] = {
                        id: linha.checkId,
                        descricao: linha.checkDesc,
                        items: []
                    };
                }
```

```
                if (linha.itemId) {
                    checklists[linha.checkId].items.push({
                        id: linha.itemId,
                        descricao: linha.itemDesc,
                        completado: linha.itemCompl
                    });
                }
            });
            return Object.values(checklists);
        });
};
```

E exponha esse endpoint no router de tarefas:

```
router.get('/:id/checklists', async (req, res) => {
    const usuario = req.user.sub;
    const idTarefa = req.params.id;
    try {
        const checklists = await tarefasService.buscarChecklists(idTarefa, usuario);
        res.send(checklists);
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});
```

Implemente agora um método para *alterar* os checklists de uma tarefa. Note que o processo é bem complexo, pois envolve uma *mesclagem* (merge) entre duas listas. Para simplificar, o seguinte método utilitário foi adicionado em um módulo `colecoes` na raíz do projeto:

```
/**
 * Compara os itens entre origem e destino, executando as seguintes ações:
 *
 * - "criar" é chamado sempre que um item é encontrado em origem e não em destino.
 * - "atualizar" é chamado sempre que um item é encontrado dos dois lados.
 * - "remover" é chamado sempre que um item está no destino mas não está na origem.
 *
 * A igualdade entre itens é decidida através de uma chamada ao método "saoIguais".
 *
 * Os vetores originais não são modificados.
 */
module.exports.mesclar = async (origem, destino, saoIguais, criar, atualizar, remover)
=> {
    const restantes = [].concat(destino);
    for (const x of origem) {
        let indiceDoExistente = -1;
        for (let i = 0; i < restantes.length; i++) {
            if (saoIguais(x, restantes[i])) {
                indiceDoExistente = i;
                break;
            }
        }
        if (indiceDoExistente >= 0) {
```

```
            await atualizar(x, restantes[indiceDoExistente]);
            restantes.splice(indiceDoExistente, 1);
        } else {
            await criar(x);
        }
    }
    for (const x of restantes) {
        await remover(x);
    }
}
```

Agora, usando este utilitário, implemente o método no `tarefas-service`:

```
module.exports.alterarChecklists = (idTarefa, checklists, usuario) => {
    return knex.transaction(async trx => {
        const origem = checklists;
        const destino = await buscarChecklists(idTarefa, usuario);
        await mesclar(origem, destino,
            (x, y) => x.id === y.id,

            // criar
            x => inserirChecklist(trx, idTarefa, x),

            // atualizar
            (x, y) => atualizarChecklist(trx, x, y),

            // remover
            x => removerChecklist(trx, x));
    });
};


function inserirChecklist(trx, idTarefa, checklist) {
    const idChecklist = checklist.id || uuidv4();
    return trx('checklists')
        .insert({
            id: idChecklist,
            tarefa_id: idTarefa,
            descricao: checklist.descricao
        })
        .then(() => inserirItemsChecklist(trx, idChecklist, checklist));
}


function atualizarChecklist(trx, novo, existente) {
    return trx('checklists')
        .where({ id: existente.id })
        .update({ descricao: novo.descricao })
        .then(() => atualizarItemsChecklist(trx, novo, existente));
}
```

```
function atualizarItemsChecklist(trx, novo, existente) {
    const origem = novo.items || [];
    const destino = existente.items;
    return mesclar(origem, destino,
        (x, y) => x.id === y.id,

        // criar
        x => trx('items_checklist').insert({
            id: x.id || uuidv4(),
            checklist_id: existente.id,
            descricao: x.descricao,
            completado: x.completado
        }),

        // atualizar
        (x, y) => trx('items_checklist').where({ id: y.id }).update({
            descricao: x.descricao,
            completado: x.completado
        }),

        // remover
        x => trx('items_checklist').where({ id: x.id }).del());
}


function removerChecklist(trx, checklist) {
    return trx('items_checklist')
        .where({ checklist_id: checklist.id })
        .del()
        .then(() => trx('checklists')
            .where({ id: checklist.id })
            .del());
}
```

Note que foi necessário extrair o `buscarChecklists` para uma constante.

E exponha-o no `tarefas-router`:

```
router.put('/:id/checklists', async (req, res) => {
    const usuario = req.user.sub;
    const idTarefa = req.params.id;
    try {
        const checklists = req.body;
        await tarefasService.alterarChecklists(idTarefa, checklists, usuario);
        res.send();
    } catch (err) {
        console.error(err);
        res.status(500).send();
    }
});
```

Cuidado deve ser tomado, pois a implementação atual *é insegura*, pois permite que um usuário altere checklists de uma tarefa que não é dele. Corrija isso no `tarefas-service`:

```
module.exports.alterarChecklists = (idTarefa, checklists, usuario) => {
    return knex.transaction(async trx => {
        await validarPermissaoDeAcesso(idTarefa, usuario);
        // [...]
    });
};

async function validarPermissaoDeAcesso(idTarefa, usuario) {
    const res = await knex('tarefas')
        .join('usuarios', 'usuarios.id', 'tarefas.usuario_id')
        .where({
            'usuarios.login': usuario,
            'tarefas.id': idTarefa
        })
        .count()
        .first();
    if (res.count <= 0) {
        throw Error('Acesso negado!');
    }
}
```

Uma nota sobre pooling de conexões

Independentemente da estratégia adotada para se comunicar com o banco de dados, é interessante otimizar o seu pool de conexões, a fim de garantir uma boa performance e aproveitamento de recursos do servidor. Por padrão o Knex configura um mínimo de 2 conexões e um máximo de 10, mas isso pode ser facilmente modificado.

Pools muito pequenos podem causar gargalos, e pools muito grandes podem usar mais do banco de dados do que lhe é necessário. Encontrar o balanço correto é fruto de monitoramento e análise constante do ambiente em execução.