Projeto Trainee – App de Tarefas

Augusto Fernandes Ildefonso

Eduardo Neves

ICMC Júnior

Instituto de Ciências Matemáticas e de Computação

Sumário

Geral

Descrição do projeto

O projeto consiste em um aplicativo mobile, tanto iOS quanto Android, de gerenciamento de tarefas. Nesse app, o usuário poderá adicionar tarefas com diversas propriedades, como nome, data de entrega, horário, descrição e status. Além disso, ele poderá editá-las e excluí-las, ou então, adicionar uma tarefa rápida que terá apenas nome.

O aplicativo divide as tarefas de acordo com o status dela, para facilitar tanto a visualização, quanto a organização do usuário. Desse modo, existem as seguintes divisões: atrasadas, não iniciadas, em andamento e concluídas. Esses status são atribuídos pelo usuário, exceto pelo status de atrasado, que é calculado pelo app de acordo com a data de entrega.

Além dessas funcionalidades, o usuário também terá uma conta. Com ela, ele terá acesso à todas as suas tarefas, em qualquer aparelho que conecte. O usuário também terá liberdade de alterar seus dados pessoais, sair da conta e deletá-la.

Tecnologias utilizadas

O app é dividido entre front-end, back-end e banco de dados. Cada uma dessas partes utilizou uma tecnologia diferente para a sua implementação. Ademais, a sua prototipação foi feita com o Figma, que permite um desenvolvimento de protótipos tanto de baixo nível quanto de alto nível

O front-end foi implementado com *React-Native*, um framework baseado em *Javascript*, utilizado para desenvolver aplicativos nativos para iOS, Android, entre outros. Por essa característica de servir para várias plataformas, que ele se tornou um bom candidato para desenvolver o projeto.

O back-end foi implementado com *NodeJS*, que é uma ferramenta que permite simular um servidor, interpretando código *Javascript*. As bibliotecas utilizadas foram: *express*, *esm*, *cors*, *bcrypt*, *jsonwebtoken*, *mongoose*. Além disso, o servidor foi implementado como uma *Rest API*, seguindo os princípios básicos desse modelo.

O banco de dados foi implementado com *MongoDB*, um software de banco de dados não relacional, *NoSQL*, orientado a documentos, ao invés de tabelas e linhas. Ele foi escolhido por causa da sua fácil implementação com o *NodeJS* através da biblioteca *mongoose*.

Por último, foi utilizado o *Expo*, uma plataforma para fazer apps nativos para diversos sistemas com *React* e *Javascript*, para simulação do aplicativo. Ele foi usado através da sua biblioteca para o *NodeJS*.

Front-End

Organização do projeto

O front-end é dividido em algumas pastas principais para facilitar a organização. Na pasta do próprio projeto, está o *App.js*, arquivo que contém a função responsável por definir o aplicativo.

Agora, na pasta *src* estão os códigos dos componentes, das telas e da navegação. Na pasta *components* estão os códigos de todos os componentes que foram usados para montar as telas. Eles seguem o seguinte padrão para o nome: tipo de componente + dado que armazena/função que faz + tela a que pertencente (esse último é opcional); um exemplo é *inputCPFAlterar*.

Ainda na pasta *src*, temos a pasta *navigators*, que contém o arquivo *RootNavigator*. Esse arquivo é o responsável por criar a pilha de telas que formam o aplicativo e assim permitir a navegação entre elas. Agora, a última pasta dentro de *src* é *screens* que possui quatro subpastas, cada uma referente a uma tela.

Principais Arquivos

Dentre todas essas pastas, alguns arquivos se destacam, entre eles estão: *App.js*, *RootNavigator.js*, *TelaLogin.js*, *TelaCadastro.js*, *TelaAlterar.js* e *TelaTarefas.js.* O primeiro deles, *App.js*, é o arquivo que contém a função que cria o aplicativo, chamada de *App*. Além disso, nele é configurado as fontes personalizadas que são usadas no projeto.

Já no arquivo *RootNavigator*, como foi dito antes, são configuradas as telas, criando a pilha através do componente *Stack*. Com essas pilhas criadas, é possível alternar entre as telas usando a função *navigation.navigate("Nome da Tela")*.

Nos quatro últimos arquivos, são configuradas as telas do projeto. Para criá-las, utilizou-se componentes, tanto os padrões do *React-Native* quanto os personalizados, que se encontram na pasta *components*.

Descrição de telas

Agora iremos explorar mais a fundo cada uma das quatro telas. Começando pela tela de login, do arquivo *TelaLogin.js*, ela é a tela inicial do aplicativo, então ele sempre irá abrir nela. Caso o usuário já tenha efetuado o login anteriormente, ele será guardado na memória local do aparelho e não será necessário efetuá-lo novamente, ou seja, irá pular para a tela de tarefas. Porém se for a primeira vez fazendo login ele irá abrir na tela de login e permanecer nela até ser feito o login.

Da tela de login, ao clicar no botão "Criar Conta", seremos direcionados a tela de cadastro, do arquivo *TelaCadastro.js*. Essa tela possui diversos inputs de dados pessoais que o usuário deverá preencher para que possa criar a sua conta. Quando terminar de preencher, ele deve clicar para criar conta. Assim, ele será encaminhado a tela de login e precisará fazer login, mas das próximas vezes o seu login já estará salvo.

Ao fazer login, vamos para a tela de tarefas, do arquivo *TelaTarefas.js*, que é a tela principal do app. Nela pode-se ver as tarefas de acordo com o seu status, criar tarefas através do botão de "mais" e criar tarefas rápidas (que só possuem nome) através do input no canto inferior. Além disso, nessa tela há o menu lateral, que ao ser fornece 3 opções ao usuário: sair da conta, deletar conta e alterar dados.

Por último, a tela de alterar dados, do arquivo *TelaAlterar.js*, pode ser acessada ao clicar para alterar dados no menu lateral. Ela possui diversos inputs que o usuário deve preencher, exceto o cpf e o email que não permitem edição, e que ao clicar para alterar dados, os valores que ele inseriu serão modificados. O cpf e o menu não podem ser modificados pois são com eles que o usuário é identificado na base de dados.

Principais funções de services

As funções podem ser divididas em duas categorias: de usuário e de tarefas. As funções de usuários são aquelas que estão relacionadas a própria conta e seus dados. Já as funções de tarefas são aquelas relacionadas diretamente as tarefas.

Começando pelas funções de usuários, temos quatro funções: *create, login*, *update* e *delete*. A função *create* está associada ao botão de criar conta e ela envia ao servidor os dados necessários para criar a conta do usuário.

A função *login* é a função responsável por enviar ao servidor os dados (login e senha) necessários para fazer o login. Além disso, ela também é responsável por receber e armazenar o token de acesso do usuário, para que não seja necessário sempre fazer login ao enviar uma requisição ou então quando entrar no app.

A função *update* é a responsável por alterar os dados do usuário. Logo, ela está associada ao botão de alterar dados da tela *TelaAlterar.js*. Ela envia todos os dados do usuário para o servidor e ela altera todos menos o email e o cpf.

Por último, a função *delete* é a responsável por deletar a conta do usuário. Ela envia para o servidor somente o login, seja email ou cpf, para que ele ache a conta referente à ele e a delete.

Agora quanto as tarefas, também há quatro funções: *create*, *update*, *delete*, *return*. A função create é a responsável por criar as tarefas na base de dados, seja ela tarefa rápida ou não.

A função update atualiza os dados de uma tarefa. Ela é usada quando o usuário quer alterar o status de uma tarefa (ou então quando o servidor vai marcar a tarefa como atrasada), ou, então, quando que alterar alguma outra informação da tarefa.

A função delete é a responsável por deletar uma tarefa. É passado o login do usuário e o nome da tarefa e ela apaga a tarefa. Ela está associada ao ícone de lixeira do pop-up de editar tarefa.

Por último a função return é a responsável por retornar todas as tarefas encontradas de um usuário. Assim, o front-end processa elas e mostra de acordo com o status de cada uma.

Back-End

Organização do projeto

O projeto está contido praticamente dentro da pasta *src*. O único arquivo relevante fora dele é o *package.json* que é um *json* responsável por definir e descrever o projeto. Agora dentro da pasta *src*, temos quatro subpastas.

A primeira é chamada de *controllers*, nela temos arquivos responsáveis por lidar com as *requisições HTTP*, definindo funções para cada uma das requisições que o front-end faz (são as funções definidas na seção "Principais funções de service"). Por tratar dessas requisições, ela é dividida em dois arquivos: um para as requisições referentes ao usuário e um referente as requisições das tarefas.

A próxima é chamada *db* e ela contém as informações relacionadas ao banco de dados. Ela possui o arquivo *connection.js* que é responsável por conectar o back-end com o servidor e possui também uma pasta chamada *models*. Essa pasta guarda dois arquivos que são modelos dos documentos que serão guardados no servidor, um modelo de usuário e um modelo de tarefa.

Em seguida há a pasta *middleware* que guarda os códigos responsáveis pela autenticação. Eles estão no arquivo *auth.js* e eles garantem que o usuário estará autenticado antes de realizar as funções das tarefas e também algumas funções dos usuários, como alterar dados e deletar conta.

Por fim, a pasta *routers* guarda os routers do usuário e das tarefas. Neles, são definidos os caminhos referentes a cada função, tanto de tarefa quanto de usuário.

Principais arquivos

O principal arquivo é o *index.js* pois ele é o arquivo que cria o nosso servidor e importa as funções necessárias. Entre elas estão o body-parser e o cors. Além disso, ele define o uso das rotas de usuário e de tarefas.

Outro arquivo importante é o *auth.js*, pois ele é o responsável por configurar a autenticação automática do usuário. Isso ocorre através do token do usuário, que é criado para cada conta nova.

Os arquivos de *router* também são importantes pois eles dividem e organizam as funções de acordo com o caminho da requisição. Os arquivos de *controller* trabalham junto do *router* e são os responsáveis por definir as funções que ocorreram em cada tipo de requisição.

Lista de endpoints

- '/users/create' – para criar um usuário
- '/users/login' – para fazer login
- '/users/update' – para alterar dados da conta
- '/users/delete/:login' – para deletar a conta referente ao login enviado
- '/tasks/create' – para criar uma tarefa
- '/tasks/update' – para alterar dados de uma tarefa
- '/tasks/delete/:login' – para deletar uma tarefa do usuário fornecido por login e cujo nome da tarefa está no body
- '/tasks/:login' – para retornar todos as tarefas do usuário cujo login foi fornecido

Autenticação e Autorização

A autenticação é feita com a biblioteca *jsonwebtoken* que cria tokens para os usuários quando eles fazem login e esses tokens ficam salvos no dispositivo. Quando o usuário quer acessar uma área que requer login, o token é enviado no header da requisição e o arquivo *auth.js* contém a função responsável por verificar se o token recebido é válido para aquele usuário e então ele libera o acesso. Para verificar que o token é válido, usa-se a função *jwt.verify*.

As rotas que requerem autenticação são todas aquelas que dependem do usuário e mudam de acordo com cada conta. Ou seja, todas as rotas de tarefas precisam de autenticação porque todas dependem do usuário e praticamente todas as rotas do user precisam de autenticação pois dependem de estar logado para conseguir realizar uma ação (como sair da conta, deletar a conta e alterar dados da conta). As únicas rotas que não dependem de autenticação são o login (pois nele, no primeiro login, que

é recebido o token) e a de criação de conta (pois o token só é criado no login, que ocorre após a criação de conta).

ORM

No projeto, foi usado como ORM o *mongoose* para poder manipular facilmente os documentos da base de dados. Já a base de dados escolhida foi o *MongoDB*, uma base de dados não relacional, *NoSQL*.

O *mongoose* foi usado nos arquivos de definição de modelos, ou seja, no *user.js* e no *task.js*, que definem, respectivamente, os documentos dos usuários e os documentos das tarefas. Usou-se a função *Schema* para criar um esquema para o modelo e a função *model* foi a função que de fato criou o modelo a partir do *Schema*. Assim, sempre que quiser interagir com o banco de dados pode-se usar o modelo *Task* ou *User*, garantindo um fácil acesso.

Criptografia e proteção de dados

Para garantir a segurança do projeto, foi aplicada criptografia na senha da conta do usuário. Ao criar uma conta a senha enviada pelo usuário passa pelo método *pre* do modelo, antes de passar pelo método *save,* que criptografa a senha usando a biblioteca *bcrypt* e a função *hash* da biblioteca. Essa senha criptografada que é salva no banco de dados.

Quando é feita alguma tentativa de login ou então alguma ação que depende da senha, o que acontece é que a senha enviada é comparada com a do banco de dados através da função *compare* da biblioteca *bcrypt*. Essa função permite comparar a senha criptografada do banco de dados com a recebida que não está criptografada, sem que haja falhas ou vazamentos da senha.

Desse modo temos um sistema que garante segurança tanto para armazenar os dados quanto para consultá-los e analisá-los. Isso graças a biblioteca bcrypt que facilita construir aplicações seguras.