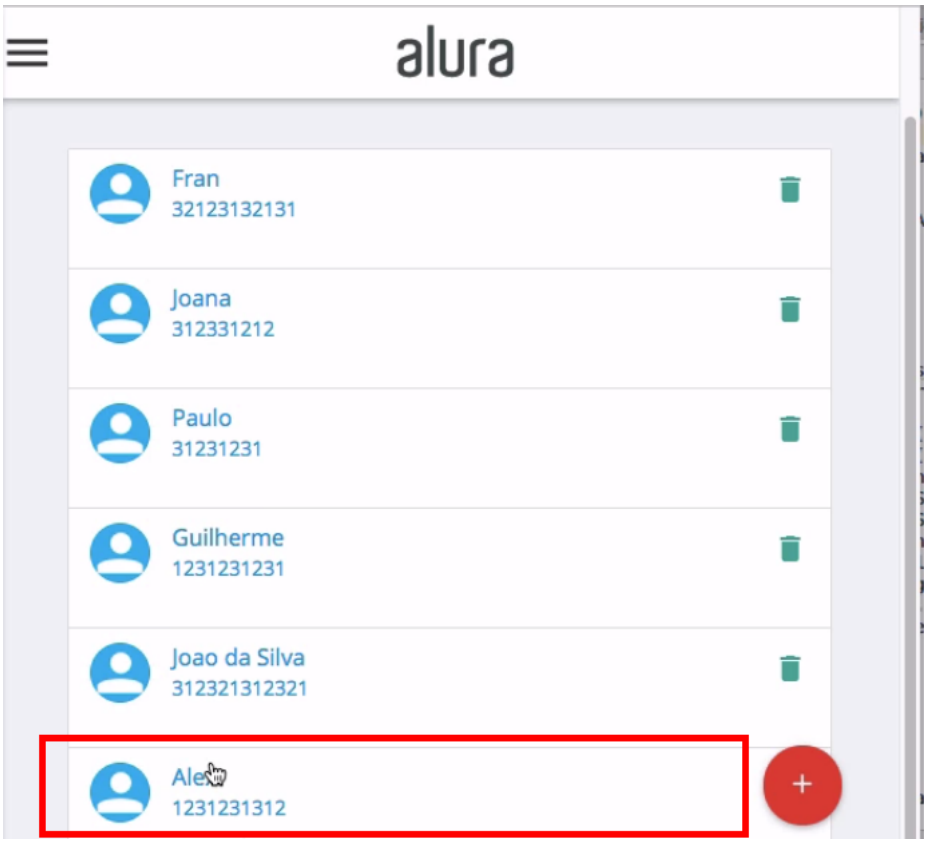
07

Removendo o aluno quando recupera a conexão com o servidor

Transcrição

Conseguimos colocar o *Soft Delete* em nossa aplicação, mas ainda veremos o que acontece quando recuperamos a conexão e tentamos sincronizar os dados.

Executando a aplicação, vamos recuperar a conexão. Antes de fazermos uma *swipe*, verificaremos se o último aluno que tentamos remover ainda está no servidor. Fazendo o *swipe* e atualizando o servidor, perceberemos que nada mudou e o aluno continua no servidor.



Por que será que o aluno "Alex" não foi removido? Apesar de parecer complicado, é mais simples do que parece. É isso que vamos tentar entender.

Qual era o mecanismo que utilizávamos para pegar todas as alterações feitas na aplicação e depois mandar ao servidor? Simplesmente, pegávamos as informações dos alunos que não eram sincronizados. Precisamos indicar que no momento que desativamos um aluno, ele passa a ser "não sincronizado".

Para fazer isso, acessaremos a classe `Aluno`, e dentro do método `desativa()`, chamaremos o método `desincroniza()`.

```
// ...

public void desativa(){
    this.desativado = 1;
    desincroniza();
}
```

```
// ...
```

Executando a aplicação novamente e fazendo um *swipe,* o aluno que tentamos remover permanece no servidor. Isso ocorre porque quando desativamos o aluno ainda não tínhamos implementado a dessincronização.

Alteramos o aluno no servidor para ele retornar para a listagem na aplicação com o *swipe*. Agora que ele voltou para a listagem, podemos colocar no modo avião novamente e removê-lo. Ao recuperarmos a conexão e atualizarmos o servidor, perceberemos que ele foi removido também.

Mas ainda tem um ponto que precisamos prestar atenção, na classe `AlunoSincronizador` , temos o método `deleta()` . Nele, criaremos uma *call* que não tem nenhum tipo de ação caso ela funcione corretamente.

Se tudo der certo, por exemplo, se estivermos online, podemos dizer ao nosso `AlunoDAO` para remover fisicamente, afinal já tivemos uma reposta.

Dentro do `onResponse()` , vamos instanciar o `AlunoDAO` passando o `context` no construtor. Com o objeto em mãos, chamaremos o `dao.deleta(aluno)` e não poderemos esquecer de chamar o `dao.close()` .

```
// ...

public void deleta(final Aluno aluno){
    Call<Void> call = RetrofitInicializador().getAlunoService().deleta(aluno.getId());

    call.enqueue(new Callback<Void>(){
        @Override
        public void onResponse(Call<Void> call, Response<Void> response){
            AlunoDAO dao = new AlunoDAO(context);
            dao.deleta(aluno);
            dao.close();
        }

        @Override
        public void onFailure(Call<Void> call, Throwable t){

        }
    });
}

// ...
```

Estamos fazendo o passo no qual tudo funciona corretamente, sem falhas de conexão. Se já conseguimos nos comunicar com o servidor, sem a necessidade de esperar uma reposta do servidor depois que tentarmos fazer uma sincronização.

Agora, nossa aplicação funciona bem tanto nos casos em que temos o cenário perfeito, quanto nos cenários no quais ocorrem problemas de conexão.