УДК 004

Создание приложения с использованием Electron и Angular

Вавилов Егор Дмитриевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье описан процесс создания приложения, которое отправляет запрос на добавление и удаление item. В процессе разработки используются Electron и Angular. Практическим результатом является рабочее приложение.
Ключевые слова: Electron, Приложение, Разработка, Angular

Building an app using Electron and Angular

Vavilov Yegor Dmitrievich
Sholom-Aleichem Priamursky State University
student

Abstract
This article describes the process of creating an application that submits a request to add and remove item. The development process uses Electron and Angular. The bottom line is a working application.
Keywords: Electron, App, Development, Angular

Electron – это разработка собственного фреймворка командой GitHub. Он позволяет разрабатывать нативные приложения для ОС с использованием web-технологий. Для использования Electron необходима установка Node.js, для работы с back-end.

Цель исследования – разработать приложение позволяющее создавать и удалять item.

Исследованиями в данной теме занимались следующие авторы. Е.Д.Зайцев, Д.М.Зайцев раскрыли вопросы эффективности автоматизации тестирования мобильных приложений и web [1]. А.К.Борисов разработал web-приложение позволяющее командированному сотруднику получить унифицированный доступ к приложениям IT-инфраструктурам [2]. Т.И.Тимофеев, В.В.Козлов произвели сравнительный обзор фреймворков для настольных приложений по ряду критериев [3]. Д.А.Викулина, С.Н.Макаров рассмотрели в своей работе передовые средства и технологии для разработки настольных приложений, провели их анализ и сравнительную характеристику [4].

Для начала установим Electron-Forge CLI с помощью команды «$ npm i -g electron-forge».

Следующим делом создаем шаблон Angular (рис.1)

```
$ electron-forge init electron-angular-sqlite3 --template=angular2
$ cd electron-angular-sqlite3
```

Рисунок 1 – Команда установки

Создадим еще некоторые каталоги внутри приложения командой «$ mkdir ./src/assets/data ./src/assets/model».

В качестве первого шага добавим файл модели, который будет соответствовать схеме базы данных. Для этого примера создадим класс с именем «Item». Каждый элемент будет содержать идентификатор и свойство имени. Сохраним файл в проекте по адресу «src/assets/model/item.schema.ts» (рис.2).

```
1  import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';
2
3  @Entity()
4  export class Item {
5      @PrimaryGeneratedColumn()
6      id: number;
7
8      @Column()
9      name: string;
10 }
```

Рисунок 2 – файл схемы

«TypeORM» использует декораторы машинописного текста. Будем использовать «Entity», чтобы объявить класс «Item» таблицей. «@PrimaryGeneratedColumn()» заявляет «id» как уникальный идентификатор и говорит базе данных автоматически генерировать его.

Следующим действием будет создание службы приложений, которая будет обрабатывать обмен данными от внешнего к внутреннему. Electron предоставляет «IpcRenderer» класс именно для этого. «IpcRenderer» - это класс межпроцессного взаимодействия Electron, который используется в процессе рендеринга. По сути, если необходимо использовать «IpcRenderer» для отправки сообщений основному процессу Electron. Эти сообщения будут передавать информацию основному процессу, чтобы он мог обрабатывать взаимодействия с базой данных. Electron полагается на метод «window.require()», который доступен только внутри процесса рендеринга Electron. Чтобы обойти это, можно использовать пакет «ThornstonHans ngx-electronics», который объединяет все API-интерфейсы Electron, представленные процессу рендеринга, в единую службу.

Прежде чем использовать «ngx-electron», нужно установить его командой «$ npm install ngx-electron --save». Теперь создадим сервис для обработки IpcRenderer (Рис.3).

```
import { Injectable } from '@angular/core';

import { Item } from './assets/model/item.schema';

import { ElectronService } from 'ngx-electron';
import { Observable } from 'rxjs/observable';
import { of } from 'rxjs/observable/of';
import { catchError } from 'rxjs/operators';

@Injectable()
export class AppService {
  constructor(private _electronService: ElectronService) {}

  getItems(): Observable<Item[]> {
    return of(this._electronService.ipcRenderer.sendSync('get-items')).pipe(
      catchError((error: any) => Observable.throw(error.json))
    );
  }

  addItem(item: Item): Observable<Item[]> {
    return of(
      this._electronService.ipcRenderer.sendSync('add-item', item)
    ).pipe(catchError((error: any) => Observable.throw(error.json)));
  }

  deleteItem(item: Item): Observable<Item[]> {
    return of(
      this._electronService.ipcRenderer.sendSync('delete-item', item)
    ).pipe(catchError((error: any) => Observable.throw(error.json)));
  }
}
```

Рисунок 3 - app.service.ts

Когда сервис завершен, сделаем «app.component.ts», зарегистрируем его для DI. Там добавим простой html-шаблон и функции для обработки событий кнопок (рис.4).

```
@Component({
  selector: 'App',
  template: `<div style="text-align:center">
    <h1>
        Welcome to {{ title }}!
    </h1>
    <button (click)="addItem()" mat-raised-button>Add Item</button>
    <button (click)="deleteItem()" mat-raised-button>Delete Item</button>
    <h2>Here is the contents of the database: </h2>
    <div>
        <ul style="list-style: none">
            <li *ngFor="let item of itemList">
                {{ item.name }}
            </li>
        </ul>
    </div>
</div>`,
})
export class AppComponent implements OnInit {
  public readonly title = 'my app';
  itemList: Item[];

  constructor(private appservice: AppService) {}

  ngOnInit(): void {
    console.log('component initialized');
    this.appservice.getItems().subscribe((items) => (this.itemList = items));
  }

```

Рисунок 4 - app.component.ts

Теперь откроем «src/index.ts» файл и добавим код для подключения к базе данных. Нужно сделать две вещи: подключиться к базе данных и добавить функции для обработки запросов от процесса рендеринга (рис.5-6).

```
16  const createWindow = async () => {
17    const connection = await createConnection({
18      type: 'sqlite',
19      synchronize: true,
20      logging: true,
21      logger: 'simple-console',
22      database: './src/assets/data/database.sqlite',
23      entities: [ Item ],
24    });
25
26    const itemRepo = connection.getRepository(Item);
27
28
29    mainWindow = new BrowserWindow({
30      width: 800,
31      height: 600,
32    });
33
35    mainWindow.loadURL(`file://${__dirname}/index.html`);
36
38    if (isDevMode) {
39      mainWindow.webContents.openDevTools();
40    }
41
43    mainWindow.on('closed', () => {
47      mainWindow = null;
48    });
49
50    ipcMain.on('get-items', async (event: any, ...args: any[]) => {
51      try {
52        event.returnValue = await itemRepo.find();
53      } catch (err) {
```

Рисунок 5 - index.ts

```
52        event.returnValue = await itemRepo.find();
53      } catch (err) {
54        throw err;
55      }
56    });
57
58    ipcMain.on('add-item', async (event: any, _item: Item) => {
59      try {
60        const item = await itemRepo.create(_item);
61        await itemRepo.save(item);
62        event.returnValue = await itemRepo.find();
63      } catch (err) {
64        throw err;
65      }
66    });
67
68    ipcMain.on('delete-item', async (event: any, _item: Item) => {
69      try {
70        const item = await itemRepo.create(_item);
71        await itemRepo.remove(item);
72        event.returnValue = await itemRepo.find();
73      } catch (err) {
74        throw err;
75      }
76    });
77  };
78
82  app.on('ready', createWindow);
```

Рисунок 5 - index.ts

Сначала нужно импортировать «createConnection» класс из библиотеки «TypeORM». Также необходимо импортировать схему элемента.

«createConnectionкласс» создает соединение с базой данных. Передаем ему конструктор с такими параметрами, как тип, база данных и сущности. Тип - это строка, описывающая, какой тип базы данных используется. База данных - это строка, указывающая на расположение базы данных. Entities - это то место, где сообщается «TypeORM», какие схемы ожидать.

Последний шаг - создать функции для обработки сообщений, отправляемых из «IpcRenderer». Каждая функция будет использовать «itemRepo» созданный объект для доступа к базе данных. После успешного завершения каждой транзакции функции передают новое состояние базы данных обратно в средство визуализации.

Теперь осталось запустить проект командой «$ npm run start».

Проверяем работоспособность на добавление и удаление Item (рис.6-7).

Рисунок 6 – Проверка приложения (добавление)

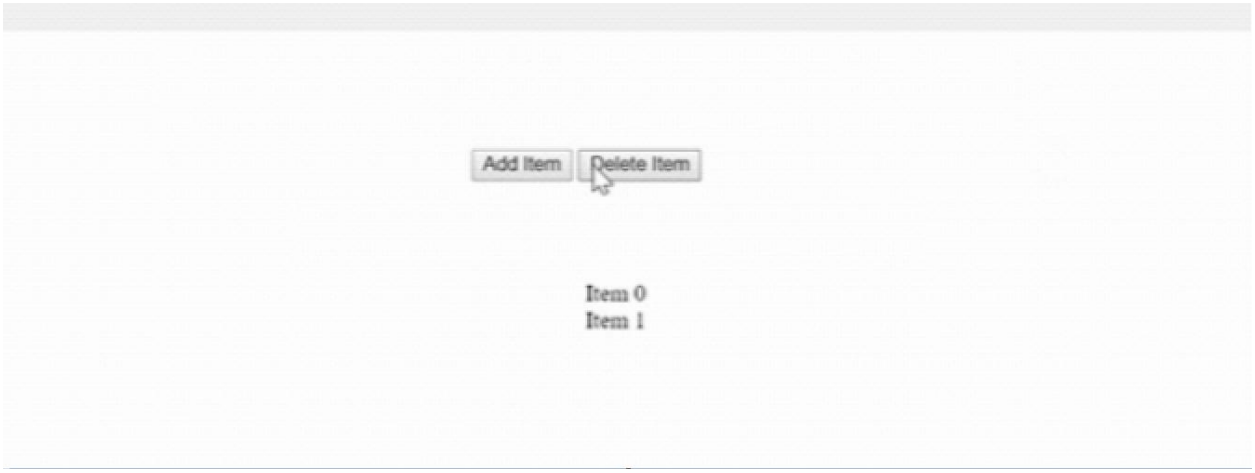

Рисунок 6 – Проверка приложения (удаление)

В данной статье был рассмотрен процесс разработки приложения, которое отправляет запрос на добавление и удаление в базу и выводит на экран. При разработки использовался Angular и Electron.

Библиографический список

1. Зайцев Е.Д., Зайцев Д.М. К вопросу об эффективности автоматизации тестирования web-, desktop- и мобильных приложений // Проблемы инфокоммуникаций 2018. №2(8). С. 56-63.
2. Борисов А.К. Sun secure global desktop все ваши приложения в окне браузера // Системный администратор. 2009. №9(82). С. 48-52.
3. Тимофеев Т.И., Козлов В.В. Обзор современных средств создания интерфейса пользователя для desktop приложений // Традиции и инновации в строительстве и архитектуре. строительные технологии. 2017. №1. С.585-588.
4. Викулина Д.А., Макаров С.Н. Современные технологии создания desktop-приложений // Наука и современность 2012. №18. С. 180-186.