Developer Manual
sec-10-k-scraper

Creators: Ed Carl, Chloe Dorward, Seth Markarian, Sena Yevenyo

An up-to-date version of this document can always be found on Google Docs.

Table of Contents

Purpose of Application

This app is a local desktop application for Windows, MacOS, and Linux that obtains companies' 10-K, 10-Q, and 10-F SEC filings, downloads the filings as HTML files, extracts the text of key sections of those documents, analyzes the text for named entities, and gathers that information in a Microsoft Excel spreadsheet. Users can either search for filings based on company names/ CIK numbers, filing type, and date range, or upload a file including similar parameters for batch uploading.

Using a desktop application rather than a web service minimizes the technical and financial requirements for FracTracker to operate it. The application is entirely self-contained on users' computers and there will be no need to spend time on configuring it or money on hosting a server. A user only needs to have an internet connection in order to run the application.

Repository Structure & Tooling

The repository is structured heavily after that of https://github.com/cawa-93/vite-electron-builder, the template that our tooling is based on. See our repository's README for full details.

Getting Started

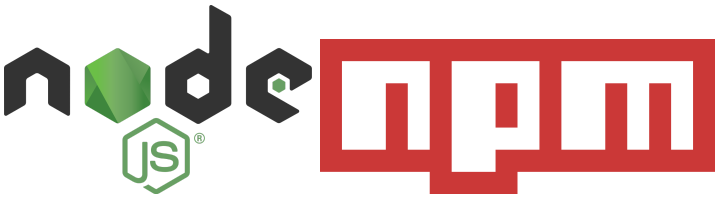
For the New Developer

Installing Required Technologies

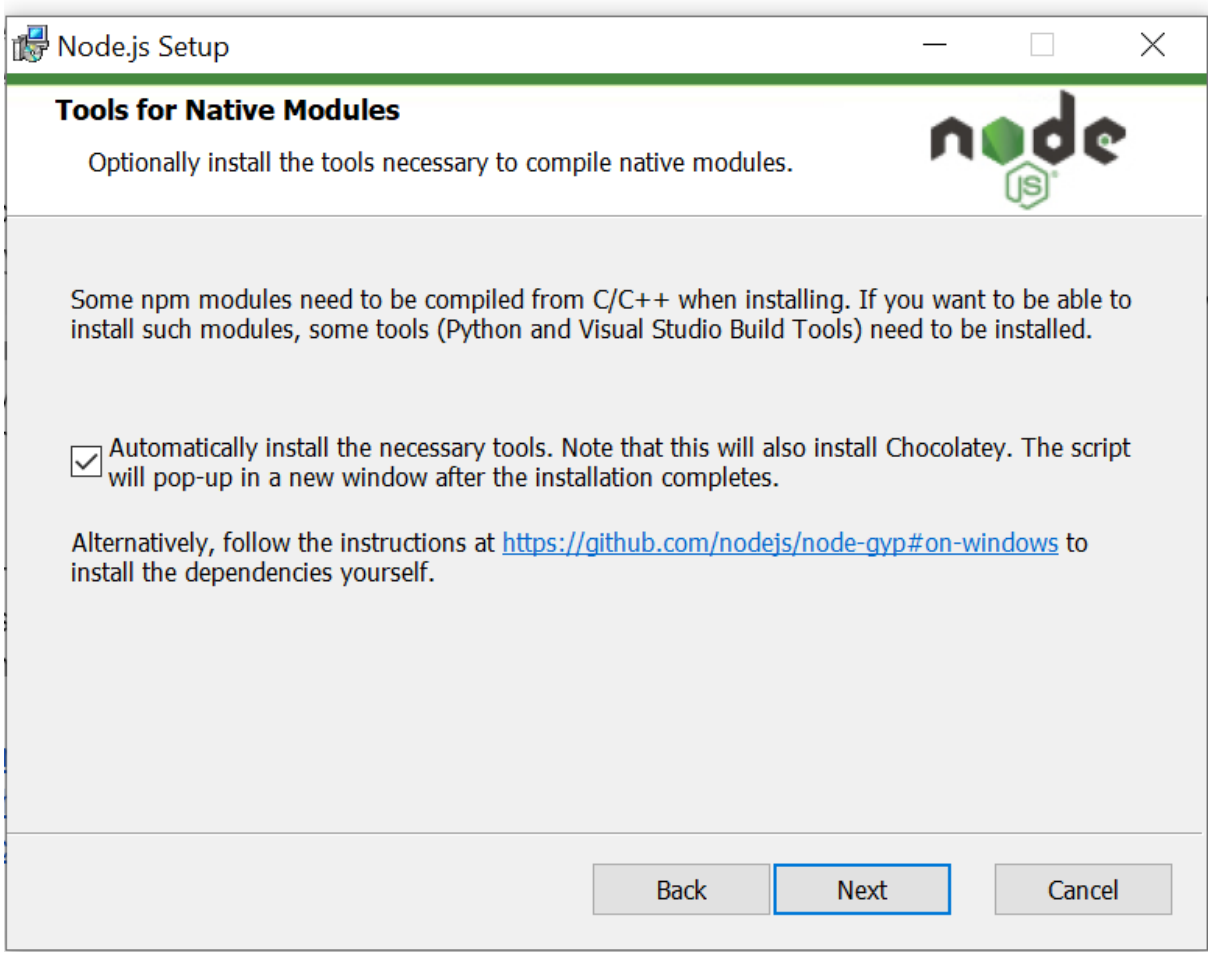
I. Integrated Development Environment (IDE)
We recommend installing Visual Studio, and denoted packages, then using Visual Studio Code. Ultimately, your choice in IDE is up to you.

A. Visual Studio (Community)
Navigate to visualstudio.microsoft.com, and download the appropriate version for your operating system. If you are using Linux, you can only download Visual Studio Code.
Run the installer (VisualStudioSetup.xxx), which can be found in your computer's Downloads folder.
Select the following workloads to install: Desktop development with C++ (Desktop & Mobile), Node.js development (Web & Cloud), and Python Development (Web & Cloud).
Click "Install While Downloading".
Once everything is installed, restart your computer.

B. Visual Studio Code
Navigate to code.visualstudio.com, and download the latest stable build for your operating system.
Run the installer (VSCodeUserSetup-xxxxxx.xxx).
Navigate through the prompts.



II. Node.js/ NPM
Navigate to nodejs.org/en/download/, and download the LTS/ recommended version for your operating system.
Run the installer (node-vxxx.xxx), which can be found in your computer's Downloads folder.
Navigate through the prompts. When you reach the appropriate screen, check the box to install necessary build tools.



Continue through the prompts, and install.
To ensure installation completed properly, run your terminal/ command prompt of choice, and run the command:

```
npm --version
```

III. Python

Navigate to https://www.python.org/downloads/, and click “Download”. This should automatically download the necessary installer for your operating system.
Note: this application was developed using Python 3.9. You may need to download a 3.9 version specifically once a newer version is released.
Run the installer (python-xxxx-xxxx.xxx), which can be found in your computer’s Downloads folder.
Navigate through the appropriate prompts.
ENSURE YOU CHECK THE BOX TO ADD ENVIRONMENT VARIABLES.

```
python --version
```

IV. Poetry
Navigate to python-poetry.org/docs/ and scroll down to the installation command for your system - copy it. Open your computer's terminal/ bash/ powershell. Paste the command into your terminal, and run it.
`poetry --version`

V. Git
Navigate to https://git-scm.com/downloads and download the appropriate installer for your system. The recommended version will appear in the little monitor graphic to the right. Next, run the installer (Git-xxx-xx.xxx), which can be found in your computer's Downloads folder. Navigate through the appropriate prompts. Default settings should be adequate.
`git --version`

Learning Relevant Languages & Technologies

The majority of the code that you may work with is written in Python for the backend, and Typescript React for the frontend. Knowing the basics of some other languages/ technologies may be helpful.

I. HTML
II. CSS
III. Typescript
IV. React
V. Bootstrap (via React-Bootstrap)
VI. Python
VII. ZeroRPC
VIII. Gevent (powers ZeroRPC behind the scenes)
IX. Electron

The App

Structure of Application

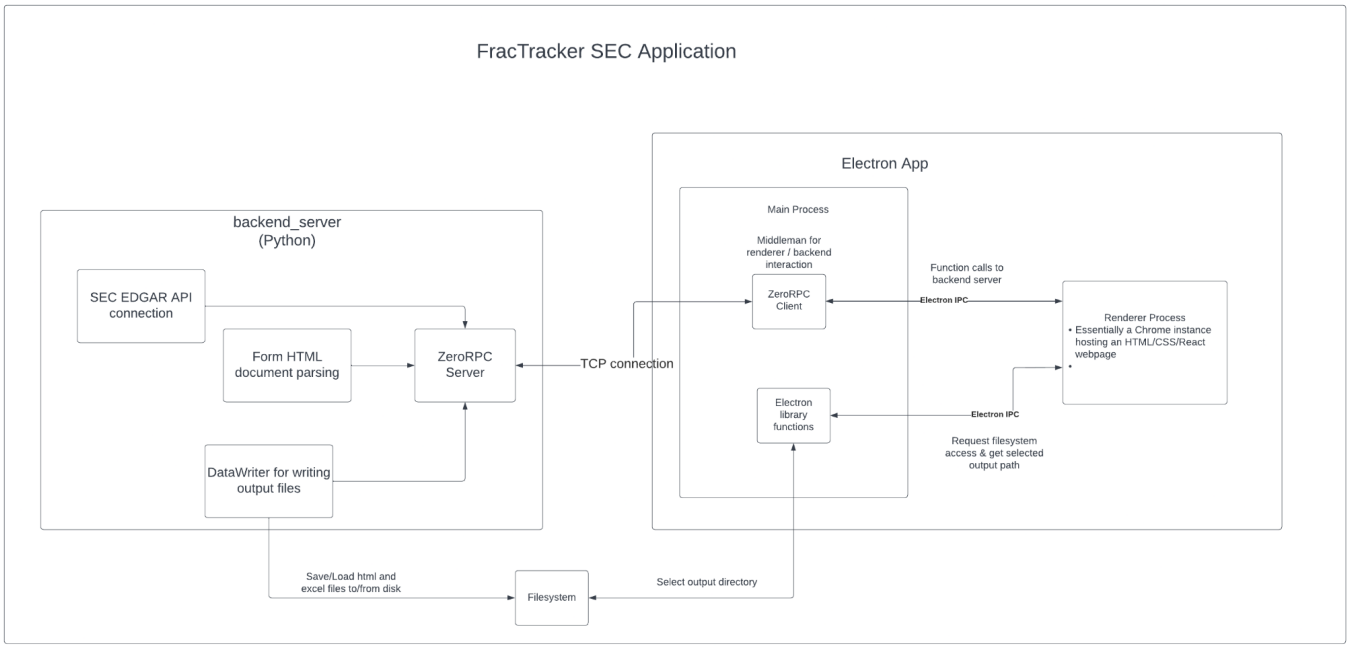


Figure: Structure of Application

Summary

The application is developed as a single desktop UI application developed using the ElectronJS Framework. The application is divided into two sections:

- The Front End (Electron)
- The Back End (Python)

These two sections communicate with each other over ZeroRPC, a cross-language library for remote procedure calls (that is, function calls to other processes or devices) over TCP.

The Front End

The app's front end is written as a single-page web app in ElectronJS, a NodeJS framework that allows developers to turn HTML/CSS/JS web pages into self-contained, cross-platform desktop apps. The app is written in TypeScript for type safety, using ReactJS to create a responsive UI. Styling was done with the React-Bootstrap component library to create a consistent aesthetic across the app.

For security purposes, we adhere to Electron's model of context isolation. Operations that require the NodeJS standard library, like launching the backend process, selecting an output folder, or using external libraries, like making ZeroRPC calls, make use of Electron's context

bridge between the main and renderer processes. To understand why this is the case, and why some functions are therefore located under ui/packages/main and ui/packages/preload instead of ui/packages/renderer, check out the documentation for Electron's process model.

Displaying Information

The job of the frontend is, of course, to display relevant information to the user. The frontend performs no business logic of its own; it gets all of its information from remote procedure calls to the backend.
The details of how users interact with the frontend can be found in the User Manual, but a brief summary follows:

- Users can search for SEC filings using a search bar with settings like document type and filing date range.
- Alternatively, they can upload a .txt file specifying the filings they're interested in processing.
- The results of their search are displayed in a table, from which users can select documents to add to the 'document queue', a list of documents that will be processed as a batch (see 'The Back End').
- Users can press the 'Show Queue' button at the bottom of the screen, which will reveal a new tab (an `<Offcanvas>` element) displaying:
    - the full contents of the queue,
    - an option to select an output folder for the results of processing,
    - A checkbox for whether NER should be applied to the results,
    - and a button to begin processing the contents queue.

The *only* business-logic-relevant state handled by the frontend are the contents of the search bar, the search results, the output folder, and the information in the 'Show Queue' screen.

Communicating with the Backend

The heavy lifting of HTTP requests, document processing, and filesystem interaction is done by the Python backend. The frontend acts as a ZeroRPC client, making `async` function calls to the `window.requestRPC.procedure()` function, which triggers function calls on the Python backend (see the 'ZeroRPC Server' section of 'The Back End' for more information).

Managing the Backend's Lifecycle

As a Python application, the backend server is its own headless process that should only be running when the frontend is also running. The frontend is therefore responsible for launching the backend at app startup and killing it when the frontend is killed. The code for this is found in `main/src/mainWindow.ts` and `main/src/platform-specific.ts`.

When the backend is launched, it looks for an open port on which to host its ZeroRPC server. When it finds one, it prints that information to its stdout, which the frontend reads from to determine what port to connect its client instance to. When it is time to kill the backend, the frontend writes the kill message to the backend's stdin, which triggers the backend's cleanup

process. This occurs whether the frontend is killed by crash, kill-signal, or simply pressing the close button.

The Back End

The back end of this application is divided into three sections: the API connection to the SEC server, writing output data and extracting sections from 10-K forms. These three code components are wrapped around a singular class which is the ZeroRPC server. The server class is the singular point of communication between the user interface and the back end process. This part is written in Python.

SEC EDGAR API Connection

The SEC EDGAR API connection is handled by the `APIConnection` class in `api/connection.py` in the `backend` folder of the root directory.

The class provides two features it provides:

- Autocomplete functionality to allow users to search for entities or businesses by their name. This is done by sending a `POST` request to an unofficial SEC EDGAR API endpoint: `https://efts.sec.gov/LATEST/search-index` with a key parameter `keysTyped` which is the search key.
- Form retrieval of any entity with the Central Index Key (CIK) number registered in the SEC EDGAR database. Instructions for making such request is found here on the SEC website: `https://www.sec.gov/edgar/sec-api-documentation`

10-K Form Document Processing

The 10-K form document parsing is handled by the `Parse` class in `parse/parse.py` in the `backend` folder of the root directory.

This class provides two functionalities. The first is parsing 10-K documents and extracting all itemized sections in the 10-K document. The extraction is done through regular expression (regex) matching of patterns indicating sections of most 10-K documents. The solution is imperfect, but it covers most cases.

Standard 10-K filings consist of the following sections (Sections extracted by the app bolded):

- **Item 1**
- **Item 1A**
- Item 1B
- **Item 2**
- **Item 3**
- Item 4
- Item 5
- **Item 6**

- **Item 7**
- **Item 7A**
- Item 8
- Item 9
- Item 9A
- Item 9B
- **Item 10**
- Item 11
- **Item 12**
- **Item 13**
- Item 14
- Item 15
- Item 16

The second functionality is applying Named Entity Recognition to the text of the extracted sections. This is done through spaCy's small english language model, described by spaCy's excellent documentation.

Writing Output Files

Raw filing HTML documents are simply downloaded from the SEC site with urllib and written to disk.
Results of document processing are written to Excel by the `DataWriter` class in `writer/write_to_excel.py` in the `backend` folder of the root directory. It works by loading the destination Excel spreadsheet as a `pandas` DataFrame (creating it if it does not already exist), manipulating its contents, and using `DataFrame.to_excel()` to write the modified contents back to disk.

ZeroRPC Server

To serve function calls from the frontend process, the backend presents a ZeroRPC server over TCP on 127.0.0.1 for the frontend to connect to, negotiating a port as described in 'The Front End'. The details of ZeroRPC's implementation matter greatly, so we highly recommend that you read the docs and, as necessary, the code. There are three main points to know:

- The ZeroRPC server exposes the methods of whatever single object you pass into it when you construct the Server object - in our case, a `BackendServer` instance, which inherits from `APIConnection`, `DataWriter`, and `Parse`.
- All information passed over gevent (function args and return values) is serialized for transport using msgpack. Natively, msgpack only plays well with de/serializing dictionaries, which limits the utility of mypy's type-checking. To get strong types that are still serializable, have your remotely callable functions return Dataclasses that extend `` `mashumaro.DataClassDictMixIn` ``, and annotate any functions that return those dataclasses with `@serializable_dataclass.remotely_callable_returns_dataclass`.

- The server handles function calls using an internal `gevent` Queue. Gevent uses a cooperative scheduling model, so make sure that any long operations regularly yield control using `gevent.sleep()`. Learn more about gevent's concurrency model here.
- For extremely long remote function calls (>5 seconds), consider using a state-polling model like what is used for `BackendServer.process_filing_set()` in which the initial remote call starts a background operation on the backend, and the frontend periodically polls that operation's state. This prevents the long-running call from hogging the scheduler and starving other requests from the frontend.

Unit Tests

Back End Testing

Testing on the back end is handled by the `unittest` Python library. Unit tests for all methods of classes handling 10-K document parsing, communication with the SEC EDGAR database and ZeroRPC rate limiting features have been implemented.

Testing for writing output files and server communication however not been implemented but manual testing done suggests that these components are functional.

Front End Testing

Unit testing for most of the front end logic has not been implemented due to complications with testing libraries not integrating well with TypeScript React combined with Electron.

An end-to-end behavioral driven test was implemented using WebDriverIO to automate all the functionalities described in the application however most of these tests were flaky (could not identify UI components or register changes in UI components). This was due to issues with the resq (https://github.com/baruchvlz/resq) library being used by WebDriverIO to drive this UI component identification. However we did leave the instructions for testing in the README of the code repository and instructions on how to continue that work of testing.

Running Locally

Compiling

In the root directory, run:
```
npm install
```
Once that command has finished, run:
```
npm run build:full
```

Running

In the root directory, run the following command based on your system:
Windows (INTEL):
```
.\ui\dist\win-unpacked\fractracker-sec-ui.exe
```

Mac (ARM):
`./ui/dist/mac-arm64/fractracker-sec-ui.app/Contents/MacOS/fractracker-sec-ui`
Linux:
`./ui/dist/linux-unpacked/fractracker-sec-ui`

Unit Tests

How to run the unit tests.
Under the backend directory, run:
`poetry run pypyr pypyr/python-quality-check`

End-to-End Tests

In the ui directory, run:
`npm run test:e2e:with_compile`

Note: For Mac Intel users there might be an issue with the WebDriverIO not figuring out the directory for the application in `ui/dist`. Just update Line 14 in `ui/wdio.conf.js`. Replace `mac-arm64` with `mac` or whatever main subfolder the application is in.

Creating an Installer

In the root directory, run:
`npm run build:dist`
This will create an installer file, for your system type (Linux, Windows, Mac) under the ui/dist directory.

Extending the System

Limitations of the Current System

Currently, our system only parses and runs NER analysis on 10-K forms. 20-F and 10-Q forms are only supported for download. Adding support for additional file types can be done by adding functionality to the `Parse` class in `backend/parse.`

The file upload feature is intended to speed up the parsing and NER analysis process, and thus only supports 10-K forms. If parsing and/or NER are enabled for other file types, modify the function `handleFileUpload` in `App.tsx` found in the `ui/packages/renderer/ts` directory to add these new files in bulk for processing.

For adding functionality for parsing and NER for additional files, this logic could be added as a function to the `Parse` class in `backend/parse.`

Suggestions for System Expansion

Table Extraction

One feature that the FracTracker team has indicated would be useful would be to extract the tables found in the HTML files into separate Excel spreadsheets. BeautifulSoup's parser already does much of the heavy lifting for this.

Multi Form Analysis

Another useful feature would be the ability to extract sections from 20-F and 10-Q forms. This code could be based on the form processing for 10-Ks, but reliably detecting sections is more of an art than a science, based on designing appropriate regexes. Once section text can be reliably extracted, applying NER to it is trivial.

Resources and Credit

Resources We Used to Write Code

Documentation Resources:

Typescript
React
Bootstrap (via React-Bootstrap)
ZeroRPC
Gevent (powers ZeroRPC behind the scenes)
Electron
Automated Testing with WebdriverIO Electron

Open Source/ Other Resources:

Search bar dropdown:
“Typeahead with React hooks and Bootstrap” by Alexander Rusev on DevRespices.net
https://devrecipes.net/typeahead-with-react-hooks-and-bootstrap/
File upload tutorials:
https://www.pluralsight.com/guides/how-to-use-a-simple-form-submit-with-files-in-react
https://thewebdev.info/2021/11/26/how-to-read-a-text-file-in-react/
https://www.youtube.com/watch?v=-AR-6X_98rM&ab_channel=KyleRobinsonYoung
End-to-End testing:
https://www.electronjs.org/docs/latest/tutorial/automated-testing