DrillingInfo Transform Launcher

Derek Foundoulis, Jake Meister, Cat Wylie

CSCI370 - Advanced Software Engineering

Final Report

June 20, 2017

Contents

1 Introduction

This paper covers the 2017 Field Session for DrillingInfo at the Colorado School of Mines.

DrillingInfo is a company based out of Austin, Texas that provides a wide array of services, which provide data on drilling for its 2,500 global clients. Their products range from a large *JavaFX* program, *Transform*, to a widget that populates spreadsheets with up to date drilling information. Transform is a large program that displays and interprets geological, geophysical, and other engineering data to its users.

For this Field Session they asked us to create a program launcher for *Transform* that allows DrillingInfo to control the client use and update frequency of their product on the client's computer. The design requested of us was a massively multiplayer online game style launcher programmed in *Electron* that launches, updates, technically manages, and helps to sell *Transform* to customers of DrillingInfo.

The *Electron* framework provides an easy way to display a graphical user interface and to communicate with DrillingInfo's authentication and updating services, (*Electron* is a specialized *Chromium* Browser that displays web-pages as regular desktop applications.) which are hosted with *Amazon Web Services (AWS)*. The design contains two windows, the Login Screen and the Main Screen. These screens work in tandem, allowing for a simple and easy to use user interface that is discretely connected to the web.

2 Requirements

2.1 Definition of Project

We built an Electron based launcher for DrillingInfo's *Transform* application. The application handles the updating, maintaining, user permissions, and launching of the program itself. Electron provides a clean interface for users to update when they prefer, unless DrillingInfo fores the update. The Electron framework provides smooth JavaScript interpretation and a way to hide the use of the Internet from users who prefer their data and tools exist offline.

The launcher also displays a blog and other tools from DrillingInfo; alternatively, this could be provided by the client. The program will also be modifiable to add more tools through a server side file provided to the launcher. Other parts are adaptive and easy to expand to allow more functionality without changing the program.

2.2 Functional Requirements

- The program launches, updates, removes, and otherwise protects the jars for Transform.
- The program implements voluntary updates, and displays a status bar while doing so. The update information is stored in JSON files.
- There is a display for user permissions that prevents them from launching anything that they do not have permission for.
- The permission system uses cookies to hold information for 15 days, allowing the user to use the program without Internet access.
- Furthermore the program allows for the implementation of tools that may be defined by the local user or by DrillingInfo. These will also use jars.
- The program displays a blog held by AWS and the logo.

2.3 Non-Functional Requirements

- The program uses Electron.
- The program has a version drop-down, which allows the user to switch between versions of Transform.
- The program has a news feed for DI news that will be updated continuously.
- The program as an update button and corresponding progress bar for the update.

- The program has a launch button.
- The program has a login page where the user will enter their credentials.
- The program shows entitlements.
- The program displays the username.
- The program allows for the addition of tools in a toolbox, local or foreign.

3 System Architecture

3.1 Electron Basics

Electron is a framework for building desktop applications using web tools (HTML/C-SS/JavaScript). The framework was originally designed for the editor *Atom*, under the name *atom-shell*. Under the hood *Electron* is a modified *Chromium* browser for offline use.

The framework is comprised of two basic parts, the main process and the renderer process. The main process manages the browsers, and allows the designer to open more windows. Each of these windows is a renderer process that can fail without taking down the whole program. These traits are inherited from *Chromium* Everything outside of `main.js` is in a renderer process. If main fails the entire program fails with it. Each of the renderer processes can communicate with the main process through an inter process communications (IPC) channel, this allows coordination between the windows. In the launcher that we built, there is only one renderer process.

3.2 Starting the Application

When the application starts, it enters the `main.js` file. This is the primary point of entry is defined in the `package.json` file provided by the Node Package Manager (*npm*). The `main.js` handles the interaction between the window and *Electron*.

Imports

The script first imports packages that allow the program to run, including 'electron', 'path', 'url', and 'request'. These are stored into constant variables for later use.

Declarations

The `main.js` script then allocates space for global variables that are visible to processes outside of the main process. The primary window variable is declared.

Cookies

Because *Electron* is a modified browser and has the ability to store cookies, however, these are stored in page as they are in a modern browser they are stored in the main process. This part of the script enables the main process to listen for requests for cookies in the main browser.

Application variable

The next declarations tell *Electron* what to do when the framework is ready to launch and close. There is also special handling for *macOS*.

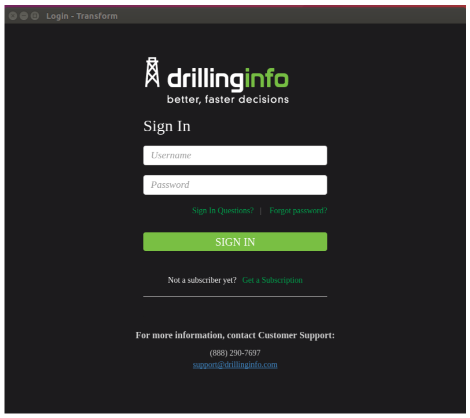
Creating the window

The application launches, when ready, into the function `createWindow()` which attaches a `browserWindow` to the variable `win`. This function also deallocates `win` when the program exits. This causes the window to vanish really fast rather than waiting for the entire process to exit.

3.3 Login Screen

`login.html` is the first page that is loaded when the window displays. This page takes in the user's username and password and returns it to the main process. This page checks these credentials against hard coded user-names and passwords, due to client security concerns. This is a huge security vulnerability, but the accounts we were given do not exist outside of a bucket on DrillingInfo's Amazon Web Services (*AWS*) server. When the sign in button is pressed the page redirects to `mainscreen.html`.

The login page also has links to support, forgotten passwords, and a place to register to buy transform products. These are more selling points for DrillingInfo.



For more information, contact Customer Support:

(888) 290-7697

support@drillinginfo.com

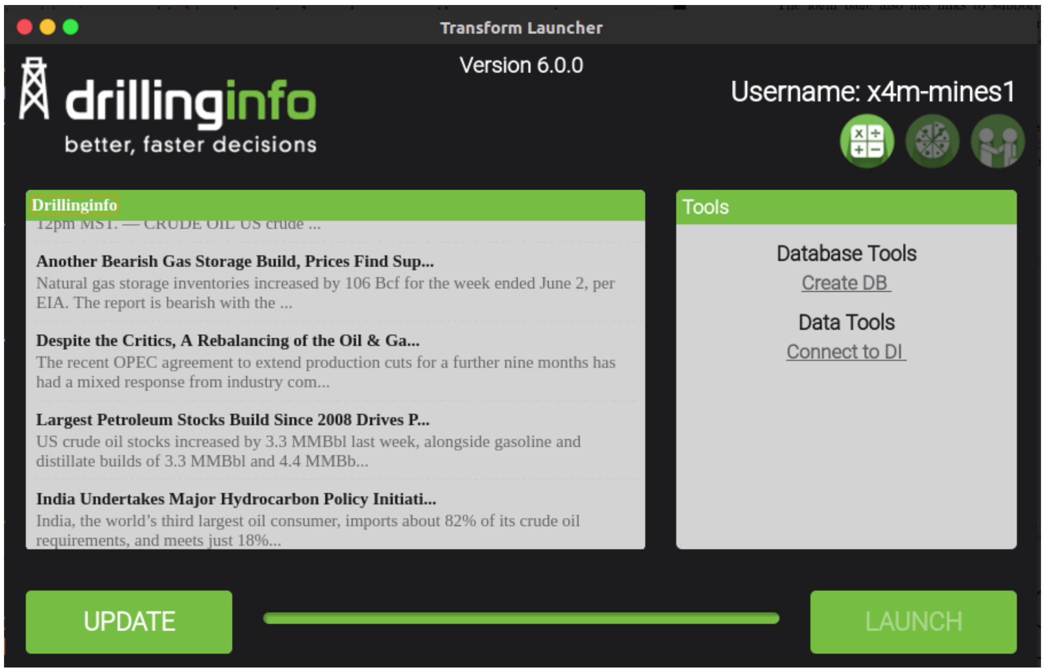
Figure 1: Log In Screen

3.4 Main Screen

The main screen, which is located in `mainscreen.html` provides most of the functionality for the program. The window is broken into seven parts: version, user-name, permissions, RSS feed, tools, update, and launch. The main script is located inside `mainscreen.html`. It controls and manages the seven parts.

The version window is currently hard coded. The user-name comes from the global variable in the set from the login screen. Permissions are loaded from the server and the icons are changed accordingly. The RSS feed is connected to a XML file from DrillingInfo's feed and the cache is updated once a day. RSS feeds are built into JavaScript. The tools are found in the configuration JSON file, in the future, users will be able to personalize these tools with a default from DrillingInfo.

Update and launch buttons provide the primary functionality for the program. They are enabled and disabled based upon the user's permissions. The update button downloads the files from DrillingInfo's S3 server.



Transform Launcher

Version 6.0.0

Username: x4m-mines1

Drillinginfo

12pm MST. — CRUDE OIL US crude ...

Another Bearish Gas Storage Build, Prices Find Sup...
Natural gas storage inventories increased by 106 Bcf for the week ended June 2, per EIA. The report is bearish with the ...

Despite the Critics, A Rebalancing of the Oil & Ga...
The recent OPEC agreement to extend production cuts for a further nine months has had a mixed response from industry com...

Largest Petroleum Stocks Build Since 2008 Drives P...
US crude oil stocks increased by 3.3 MMBbl last week, alongside gasoline and distillate builds of 3.3 MMBbl and 4.4 MMBb...

India Undertakes Major Hydrocarbon Policy Initiati...
India, the world's third largest oil consumer, imports about 82% of its crude oil requirements, and meets just 18%...

Figure 2: Main Screen

The launch button first checks to make sure there is not a forced update, then it launches *Transform* by building the UNIX command to launch and attaching it to a child process. This launches the .jar application.

4 Technical Design

The Chromium browser, that *Electron* runs on, uses a separation of processes to maintain program integrity. In this way, if a renderer process fails and becomes non-responsive, the main process will remain healthy. However, programming in this interface requires inter process communication to transfer data between the main process and the renderer process. Modules, JavaScript's libraries, are specific to certain processes making it dependent on the programmer to understand this architecture for working code. For example, the module used to store and retrieve cookies is only held within the main process, and requires a data transfer from the renderer process to store any variable data. These processes *do not share variable values*. A way to get around this conundrum is exporting the file as a module and creating getter functions for the variables needed. This work around was used in several instances in the code, but a more secure and proper way of doing so is through IPC messages. Unfortunately, due to time constraints, these changes could not be implemented.

Splitting processes and automatic threading ensures that *Electron* runs quickly. JavaScript similarly uses asynchronous function calls to run quickly. While JavaScript is compiled during runtime, it looks at all the function calls available in the current scope and feeds them into the call stack in a seemingly random order. This presents several issues such as data being used before it is written. Because this behavior is counterintuitive, there were many instances of this asynchronous behavior ruining functional code.

JavaScript introduces the callback mechanism to deal with asynchronous behavior. Callbacks are function names passed by convention as the last argument in a function call, and are called when the function's content is finished (see listing 1). This paradigm mandates the first argument of a function call to be an error and if it is detected, the callback exits and the error is thrown. Because we were new to JavaScript, callbacks were not utilized or utilized improperly. Callbacks also allow for the propagation of errors and the paradigm of error-first functions. Instead, the *async* module was used to imitate a series chain of callbacks for synchronous function calls. The *async* module has nice features including readability and ease of use. As long as the function names are descriptive enough to give a broad understanding of what happens inside them, the *async* module is more friendly to read. The callbacks from each function are gathered in the final anonymous function. This creates easily debugged code (see listing 2).

```
function callbackExample(arg1, arg2, callback) {
  //return to callback function instead of caller
    callback(arg1 + arg2);
}
```

Listing 1: Callback Example

```
async.series([
    function(callback){
      retrieveToken(callback);
    },
    function(callback){
      tokenHandler(callback);
    },
    function(callback){
      updateUsername(require('electron').remote.getGlobal('username
         ').name, callback);
    },
    function(callback){
      updatePlatform();
    }
  ], function(err, results) {
    //optional callback function goes here
    if (err) console.log(err);
    else console.log(results);
  });
```

Listing 2: Async Module Example

There were complications in the implementation of the *async* module when requests to the server were made. JavaScript views these get requests as completed as soon as they are sent instead of when data is received, causing issues with exactly when the default callbacks are run. To mitigate this problem, the `setTimeout()` function was used to give the program a delay to wait for this information to get back from the server, but because it is not consistent between runs, periods of heavy server load cause errors. If the callback could be called upon receiving the data instead of automatically at the end of the function, we believe this problem could be eliminated.

The figure below is the process flow for the launcher, it shows the process the launcher takes when the user runs the desktop Transform application. The main screen has several path options, making the flow non-linear.

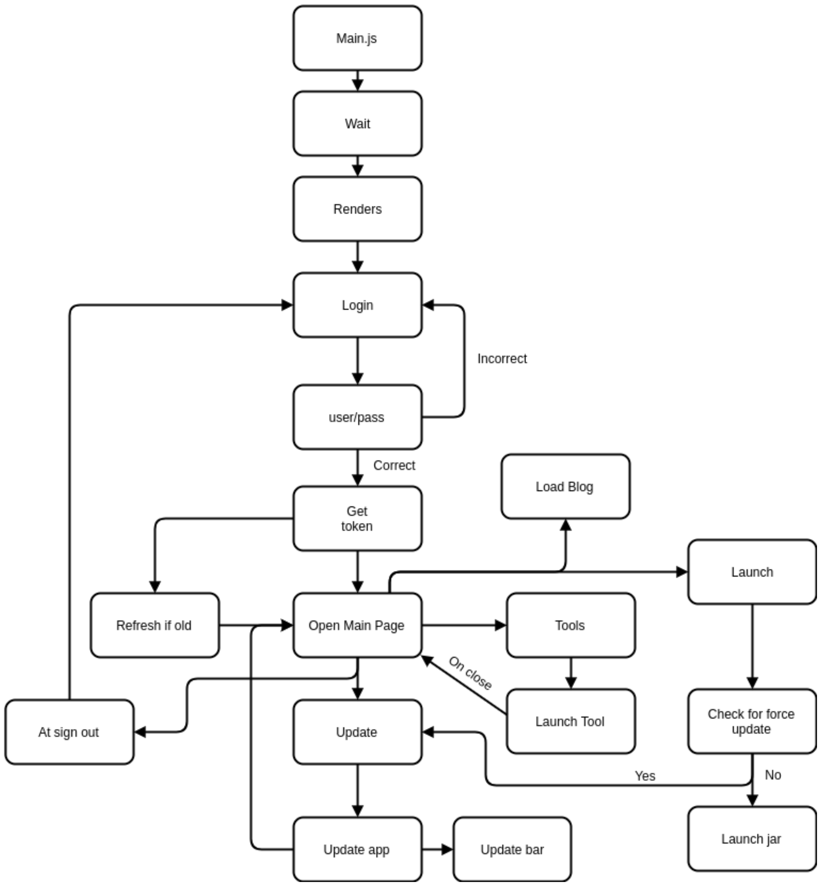


Figure 3: DI Launcher Work Flow

5 Decisions

5.1 Electron

The *Electron* framework forms the backbone of the application. It is a modified version of the *Chromium* browser that provides an easy format to make desktop applications that use HTML/CSS/JavaScript, but don't look like they are web-based. The client requested this platform for its portability, JavaScript's easy communication with online resources, ease of use, and overall look and feel.

5.2 HTML/CSS/JavaScript

These tools are required by the *Electron* framework. They are old and reliable. HTML adds elements to the application, CSS formats them and JavaScript gives them functionality.

5.3 Node.js

Electron provides use of the *Node.js* library, both on the 'front' end and the 'back' end. This means that you can use the library on the HTML side of the application. This increases functionality and removes security features as they aren't necessary when building a GUI.

5.4 RSS News Feed

The client requested a news feed on the application that would allow DrillingInfo to advertise through the launcher and get news about Transform. The RSS/XML format is easy to view and implement.

5.5 Login Screen Format

When initially designing the login screen, we decided to recycle the login screen from *info.drillinginfo.com/login*, this would both ensure that they would keep cohesion in their product design and save much needed time designing the page.

6 Results

The goal of this project was to create a launcher for DrillingInfo's Transform software in order to keep users from avoiding updating and using the software without having the proper entitlements, while still looking like a desktop app. Thus, much of our testing is based on user experience and usability. The client's design outlined a program with large buttons and section headers to create an easy to use interface. We found this design to be appropriate for our implementation. On a performance level, our launcher is smooth but takes about three seconds to load in it's entirety. This can occasionally feel slow, but we believe that the transition time is no longer than an average load time on a desktop app. In the future we would add a version selection option, this would allow the Transform development team to deploy special versions of the software to clients that need a feature that isn't offered in the normal Transform software. Login entitlements are going to be handled by the client, due to security concerns and the lack of time to complete this task. We hoped to have the tool section of the launcher launch other applications, but due to technical concerns, they currently link to the DrillingInfo website to show how they are populated by a JSON object. In the future, DrillingInfo hopes to use the launcher with other applications offered by DrillingInfo.

6.1 Lessons Learned

- JavaScript is asynchronous, so our processes all ran out of order, which was a huge block in our programming. Using synchronous processes was essential for returning the correct information at the correct time.

- Electron runs in two separate processes, main.js and renderer.js, where main sets up the window and renderer contains most of the working code and logic. This allows one process (similar to a tab in *Chromium*) to fail without affecting the others.

- JavaScript doesn't run as expected because it decides to run in the order it thinks is the most efficient.

- A scrum tool, like Pivotal Tracker is helpful for keeping tabs on where the project is and what needs to be implemented to achieve a workable product.

List of Figures