



DRILLINGINFO TRANSFORM LAUNCHER  
DEREK FOUNDOULIS, JAKE MEISTER, CAT WYLIE  
CSCI370 - ADVANCED SOFTWARE ENGINEERING

---

# Final Report

---

JUNE 20, 2017



# Contents

1	Introduction . . . . .	2
2	Requirements . . . . .	3
2.1	Definition of Project . . . . .	3
2.2	Functional Requirements . . . . .	3
2.3	Non-Functional Requirements . . . . .	3
2.4	Risks . . . . .	4
2.5	Definition of Done . . . . .	4
3	System Architecture . . . . .	5
3.1	Electron Basics . . . . .	5
3.2	Starting the Application . . . . .	5
3.3	Login Screen . . . . .	6
3.4	Main Screen . . . . .	6
4	Technical Design . . . . .	8
5	Decisions . . . . .	11
5.1	Electron . . . . .	11
5.2	HTML/CSS/JavaScript . . . . .	11
5.3	Node.js . . . . .	11
5.4	RSS News Feed . . . . .	11
5.5	Login Screen Format . . . . .	11
6	Results . . . . .	12
6.1	Lessons Learned . . . . .	12

# 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 for its 2,500 global clients. Their products range from their large *JavaFX* program, *Transform*, to a widget that populates spreadsheets with up to date drilling information.

For this Field Session they asked us to create a program launcher for *Transform* that allows Drilling Info to control their product on the clients' computer. The design laid out for us was a massively multiplayer online (MMO) game style launcher programmed in *Electron* that would launch, update, technically manage, and help 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 work in tandem, allowing a simple and easy to use GUI that is discretely connected to the web.

## 2 Requirements

### 2.1 Definition of Project

We will build an Electron based launcher for DrillingInfo's *Transform* application. The application will handle the updating, maintaining, permissions, and launching of the program itself. Electron provides a clean interface that allows users to update on their own time unless DrillingInfo wishes to force updates. The Electron framework provides smooth JavaScript interpretation and a way to hide the use of the Internet from users who prefer their data offline.

The Electron view will also display a blog and other tools given by DrillingInfo. The program will also be modifiable to add more tools, if the user wishes to. Other parts should be adaptive and easy to expand to allow more functionality without changing the program.

### 2.2 Functional Requirements

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

### 2.3 Non-Functional Requirements

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

- The program must have a launch button.
- The program must have a login page where the user will enter their credentials.
- The program must show entitlements.
- The program must show the username.
- The program must allow for the addition of tools in a toolbox, local or foreign.

## 2.4 Risks

### Technological Risks

- The launcher could load the incorrect version of Transform or pass incorrect permissions.
- It could be harder to store tokens (for verification of users) than previously thought. This is necessary for offline users acquiring access to Transform.
- Outdated jars are not updated/removed causing an error when Transform launches, conversely, new jars are not downloaded.

### Skill Risks

- No team member has ever used Electron, so all members will have to learn how to use it.
- One member is practiced in JS, the other two will need learn and practice it.
- All members need to learn how to access the /S3 server hosted by AWS through JS.

## 2.5 Definition of Done

- The launcher properly logs in users and gives them their permissions.
- Update button loads the proper jars.
- Code is commented, matches the client's style guide, and peer reviewed.
- Interacts properly with test server.
- The client approves of the design and user experience.
- Product has been delivered to the client.
- Documentation produced and updated for the product in the form of a final report.

## 3 System Architecture

### 3.1 Electron Basics

*Electron* is a framework for building desktop applications using web tools (HTML/CSS/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, allows the designer to open more windows. Each of these windows is a renderer process that can fail without taking down the whole program. 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 launch that we are building, there is only one renderer process.

### 3.2 Starting the Application

When the application starts, it enters the `main.js` file. This primary point of entry is defined in the `package.json` file provided by *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 nothing more than a modified browser, it 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 when ready launches 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 user-name and password and hands it back to the main process. As dictated by the client this page checks these credentials against hard coded user-names and passwords. This is a huge security vulnerability, but the accounts we were given don't actually exist according to the login server. When the signing button is pressed the page redirects to a new `browserWindow` session that is of different dimensions and loads `mainscreen.html`

The login page also has links to support, forgotten password pages, and a place to register to buy transform products providing more selling points for DrillingInfo.

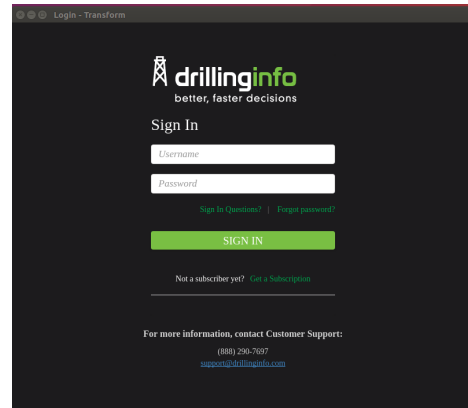


Figure 1: Log In Screen

### 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, each with their own functionality: version, user-name, permissions, RSS feed, tools, update, and launch. The main script is located inside `mainscreen.html` and 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 DrillingInfo's and the cache is updated every 24 hours. RSS feeds are built into JavaScript. The tools are found in the configuration JSON file, in the future, users would be able to personalize these tools with a default from DrillingInfo.

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

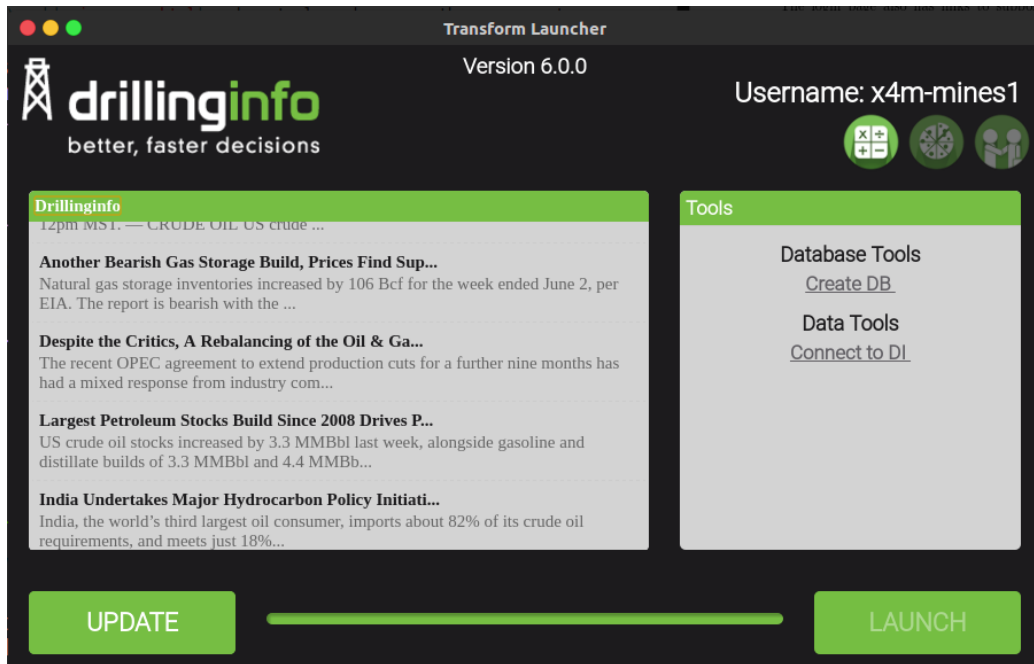


Figure 2: Main Screen

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.



## 4 Technical Design

The Chromium browser, on which *Electron* runs, 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 still be healthy. However, programming in this interface requires some specialized functions, IPC 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. The two *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 one would like to share. This work around was used in several instances in the code, but a more secure and proper way of doing so would be through IPC messages. Unfortunately, due to a lack of time, these changes could not be implemented.

Splitting processes and automatic threading ensures that *Electron* runs as fast as possible. 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 seemingly any order it so chooses. This presents several issues in practice such as data being used before it is written. Because this behavior is counterintuitive, there were many instances of this asynchronous behavior ruining seemingly 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 the team was 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 does have its upsides 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 code readers. The callbacks from each function are gathered in the final anonymous function, creating easily debugged code (see listing 2).

```
1 function callbackExample(arg1, arg2, callback) {  
2   //return to callback function instead of caller  
3   callback(arg1 + arg2);  
4 }
```

Listing 1: Callback Example

```

1  async.series([
2      function(callback){
3          retrieveToken(callback);
4      },
5      function(callback){
6          tokenHandler(callback);
7      },
8      function(callback){
9          updateUsername(require('electron').remote.getGlobal('username
10         ').name, callback);
11      },
12      function(callback){
13          updatePlatform();
14      }
15  ], function(err, results) {
16      //optional callback function goes here
17      if (err) console.log(err);
18      else console.log(results);
19  });

```

Listing 2: Async Module Example

Regrettably, there were complications in the implementation of the *async* module when requests to servers are 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 get around 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 constant 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.

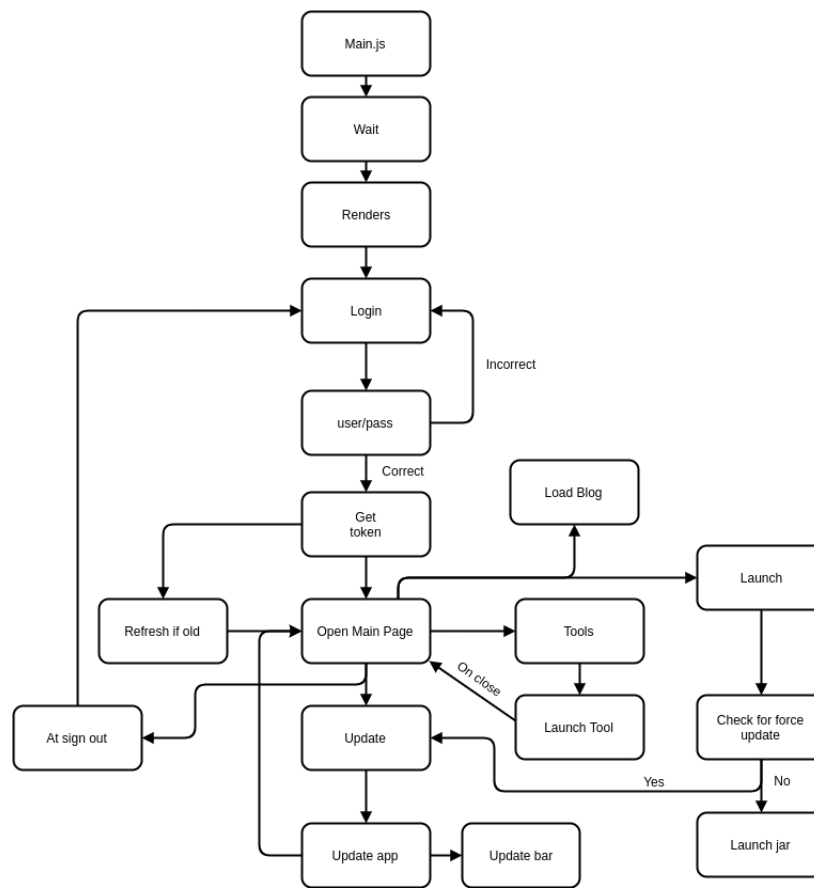


Figure 3: DI Launcher Work Flow

## 5 Decisions

### 5.1 Electron

The *Electron* framework formed the backbone of the application. It is a modified version of the *Chromium* browser that provides an easy format to make on and off line 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 to fill out the application and beautify the design.

### 5.5 Login Screen Format

When initially designing the login screen, we decided to take the login screen from *info.drillinginfo.com/login*, this would both ensure that they would look the same (keeping 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 implementation. On a performance level, our launcher is smooth but takes about 3 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 to run 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 (tab in *Chromium*) to fail without affecting the others.
- JavaScript doesn't do what you expect because it decides to run in what it thinks is the more 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

1	Log In Screen . . . . .	6
2	Main Screen . . . . .	7
3	DI Launcher Work Flow . . . . .	10