# Datava User Interface Testing: Final Report

Client: Datava Software Solutions

---

**Team sqUId squad**

Shane Cranor, Tracy Pantleo, Neil Walsh, John Walter

June 16th, 2021

## 0. Table of Contents


0. Table of Contents 1

1. Introduction 2

1.1. Datava 2

1.2. Product Vision 2

2. Requirements 3

2.1. Functional Requirements 3

2.2. Non-Functional Requirements 3

3. System Architecture/Overall Architecture 4

4. Technical Design 5

5. Quality Assurance 9

6. Results 10

7. Future Work 11

8. Appendix 12

# 1. Introduction:

## 1.1. Datava:

Datava is a company that builds cloud based systems for corporations, groups, and individuals. Datava builds Customer Relationship Management (CRM), Member Relationship Management (MRM), and Business Intelligence solutions which means they look at the interactions between their clients and the clients' customers and use data analysis to help them conduct that relationship. Their software is a custom website interface that allows for customization specific to each client. Their products predict what features will produce the best customer reaction, manage customer interactions, identify the root cause of any issues and much more. Datava values honesty, communication, excellence, innovation, and dedication through their product and company while aiming to deliver reliable software to all of their clients.

## 1.2. Product Vision:

The vision of this project is to build a framework that can record user interface data and to create robust tests that will simulate a user's behavior on Datava's cloud-based user interface. This includes writing tests to ensure User Interface (UI) elements like buttons, text boxes, and sliders work properly. These tests must be robust enough to accurately test each element properly every time. More specifically, the project focused on making it easier to create these robust tests as it was not previously feasible to create such a large quantity of tests manually without tweaking Datava's system. Smaller steps between beginning this project and reaching the final vision for the product include finding a way to add data attributes to elements on the interface and creating a logical guide for writing tests.

# 2. Requirements:

## 2.1. Functional requirements:

- Example tests for the web-based Datava user interface (built in ExtJS) written in Javascript
- The product should use the Cypress Javascript library to assist with user interface testing
- A framework to output useful information on test failure
- Modification of Datava source code to add Cypress tags. Elements with said tags are highlighted in blue in Figure 1.

## 2.2. Non-functional requirements:

- Documentation on building reliable Cypress tests for the Datava UI
- Documentation of elements to implement to make user interface testing more efficient for future developers
- Documentation explaining Datava source code modification
- Suggestions on where to move forward with improving and building upon this project
- High quality well commented code that follows ExtJS conventions
- Code functional on both macOS and Windows

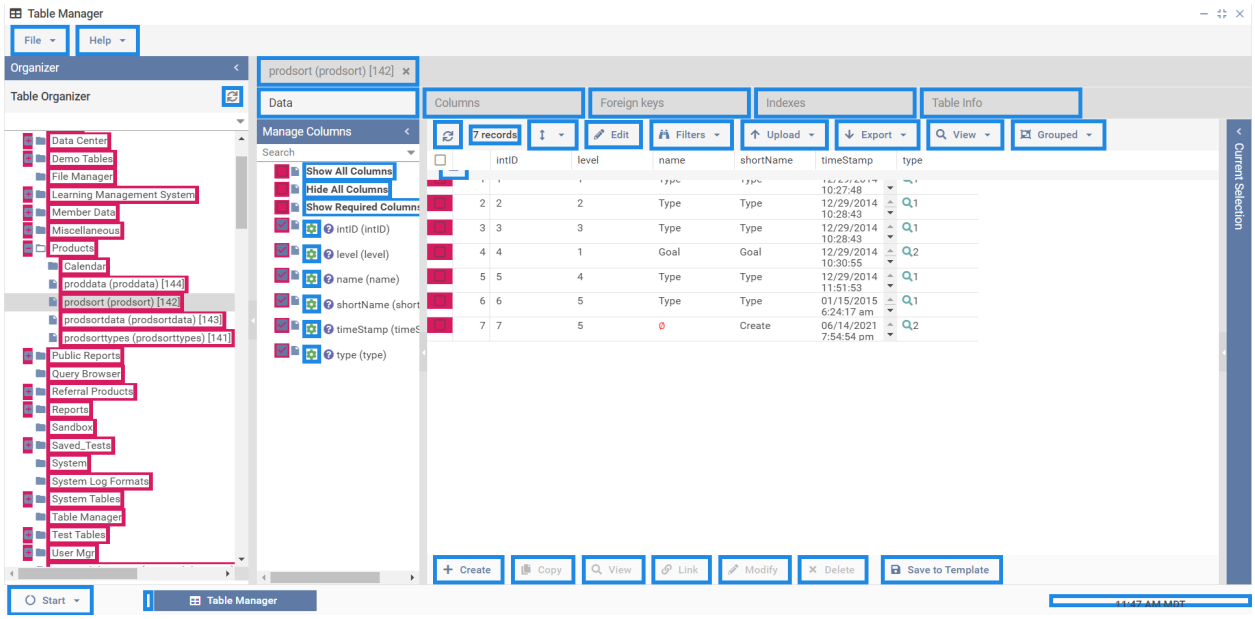



Figure 1: Highlights of tagged (blue) and untagged (red) elements in Datava application

## 3. System Architecture/Overall Architecture:

As shown in Figure 2, the architecture of our interface testing application is built around Datava's ExtJS application code and the Graphical User Interface (GUI)/App interface (light grey) that it generates. The ExtJS source code creates every element present on the application's interface, including buttons, tabs, and drop down menus. This project was not heavily involved with ExtJS source code as it mainly dealt with user interface tests. The tests only accessed the UI that the source code created. Our usage of the ExtJS source was mostly limited to adding tags to UI elements to make them easier to test. Otherwise, the source code was unchanged.

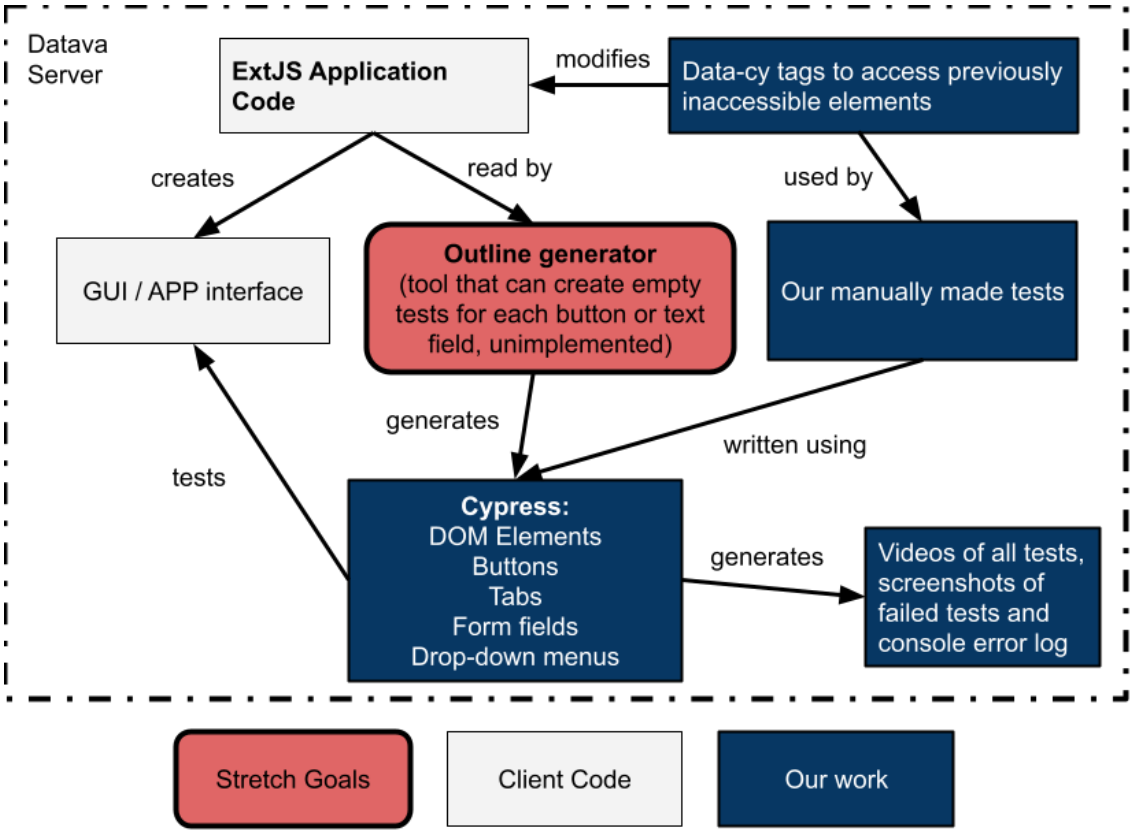



Figure 2: Diagram of system

The nodes pictured in dark blue are parts that we implemented in our project. The Cypress interface provided the framework for writing tests. Cypress has the ability to simulate user behavior and interact with interfaces. Our manually written tests give Cypress DOM elements and user actions and Cypress determines if the elements are visible and accessible and then performs the actions, see appendix 8.3 for logic behind writing Cypress tests. While writing tests, it was observed that some elements within the interface were difficult to access because they did not contain static attributes. To resolve these issues 'data-cy' tags were added to the ExtJS application code. The data-cy tags allow for previously inaccessible elements to have

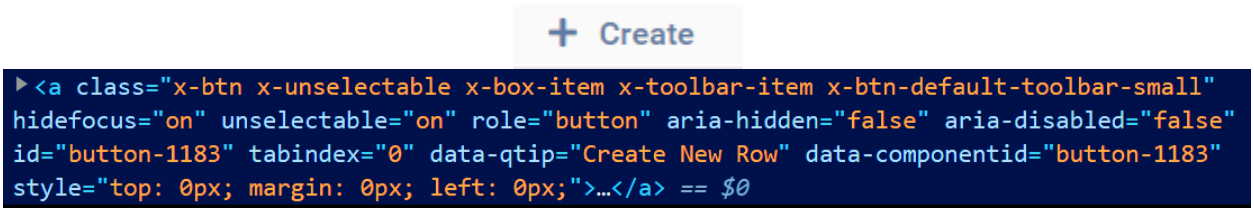
static labels for use in testing. Cypress outputs generated error logs which included thrown exceptions, screenshots of before and after a test, and videos of Cypress performing all executed tests.

Lastly, highlighted in light red with rounded edges is our stretch goal for the project. The outline generator would be a part of the user interface testing application and would be able to auto generate skeleton tests for any application after reading the source code or looking at all the data-cy tags. The outline generator would make testing more efficient for developers creating new applications.

Other than the physical structure itself, another important part of the overall architecture was understanding Cascading Style Sheets (CSS) selectors and HyperText Markup Language (HTML). Learning about the different ways elements are stored within each other was a very important part of the project: it proved to be vital when specifying what element Cypress is supposed to be testing. HTML uses a Document Object Model (DOM structure), which is a tree model where each node represents elements and branches going off that node represent child elements within the parent element. Cypress uses CSS selectors to select elements, so a basic understanding of CSS was important as well. Having an understanding of how to walk through the HTML code when inspecting an element provided insight on how Cypress should be guided to the correct element.

## 4. Technical Design:

Our technical design revolved around the implementation of Cypress: the library used for UI testing in this project. One of the first steps of the project was using online resources, like demos, official documentation, and videos to learn how to use Cypress. While having a baseline understanding was useful in terms of learning how to access and test elements, the tutorials we found focused on Javascript and there were practically no resources available for implementing Cypress with ExtJS. Because of this, we experimented a lot during the first few weeks, trying to get a solid understanding of Cypress in the context of ExtJS. Take for example this ‘Create’ button, which resides in Datava’s ‘Table Manager’ application, and its HTML code, as shown in Figures 3 and 4.




Figures 3 and 4: Create button for Table Manager application and its HTML code

When we looked at the HTML code for the button, our initial thought was to use the generated ID to access it. However, we quickly realized this was not a viable solution, as the ID

was generated dynamically by ExtJS. There was no guarantee that the number assigned, in this case 1183, would be the same every time the test was run (in fact it was different every time we reloaded the page). We realized that we would need to find a different method to describe elements we wanted to access.

```
//initial attempt at getting button by ID
it('Create button test: ID', () => {
     cy.get('#button-1183').click();
});
```

The next solution came in the form of regular expressions, or “regex”. Regex is a sequence of characters that specifies a search pattern. For example, ‘`/^Create$/`’ when used in Cypress would find text on the screen that matched exactly. The button element in the Create button is directly underneath the text, so if you click the text you click the button. This technique worked well for a while, as many of the buttons were named, however, this method did not work for every required element. There were many objects that still needed to be accessed in some way that did not have text that could be searched for, such as the refresh button, or the check boxes that appear in tables.

```
//First successful attempt at clicking a button
it('Create button test: regex', () => {
    cy.get(/^Create$/).click();
});
```

To solve this problem, Cypress allowed us to search for specific classes. For some elements, this technique worked extremely well, such as the refresh button, but for the majority, this still was not the best solution. An issue arose when there existed multiple elements that had the same class or classes. For example, most buttons had the same set of classes, and only differed deep in the children of the HTML code, which proved difficult to access. The workaround was to use the `.eq(n)` function which can pick the nth element when a Cypress function returns a list of elements. However, this was tedious and it was difficult to specify the exact element needed.

```
//using classes to select buttons
it('Create button test: class', () => {
  cy.get('.x-btn.x-unselectable.x-box-item...')
     .eq(2).click();
});
```

Eventually, we decided that we would need to add our own tags to be able to access the elements that needed to be tested. These tags would be static, meaning that every time the page is loaded, the element would have the same tag, so that we would know the element found by Cypress would be the correct one. To achieve this, we searched through Datava’s ExtJS source code, looking for the code that created the element we wanted to tag. Once the element was found, we added our tag, shown in Figure 5.

```
var btn = Ext.create('Ext.Button',{
    itemId: 'refreshBtn',
    tooltip: ESP.getLocale('refreshTheGrid'),
    overflowText: ESP.getLocale('refresh'),
    iconCls: Ext.baseCSSPrefix + 'tbar-loading',
    disabled: false, // XXX Always keep this as true since if something goes wrong, you want to be abl
    tooltip: ESP.getLocale('These are charts of data from this table this shows up'),
    autoEl: {
        'data-cy':'refresh-button-table'
    },
    handler: this.refresh,
    scope: this
    });
this.refreshBtn = btn;
tbar.add(btn);
```

Figure 5: Example of tag being implemented for refresh button

We could not find the Ext.create function for every element, so we looked into overriding the ExtJS components themselves instead of modifying their creation. We modified existing overrides and created new override files that would add tags to their respective elements. This was done by adding listeners that modify the DOM elements after they are rendered as shown in Figure 6. This also had the added benefit of using the non-dynamic part of the object's ID to specify more clearly what kind of element it was. As shown in Figure 8, there are now data-cy tags in the HTML compared to the original HTML in Figure 7.

```
datava2_cypress > js > overrides > button > JS Button.js > ...
1   Ext.define('ESP.overrides.button.Button',{
2
3       override: 'Ext.button.Button',
4       listeners: {
5           afterrender: function(cmp) {
6               let elText = cmp.text;
7               if(!elText){ //if elText is null, call it unnamed
8                   elText = "unnamed"
9               }
10              //don't overwrite an existing tag if it is more useful than the text is contains
11              if(!cmp.getEl().getAttribute("data-cy")){ //checks if tag already exists
12                  //set data-cy tag
13                  cmp.getEl().set({
14                      "data-cy": elText.replace(/\s+/g,"_") + '-button'
15                  });
16              }
17          }
18      },
```

Figure 6: Example of tags being added with override file

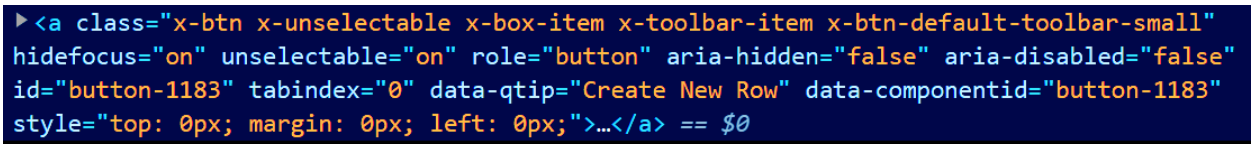



Figure 7: For comparison, before data-cy tag added

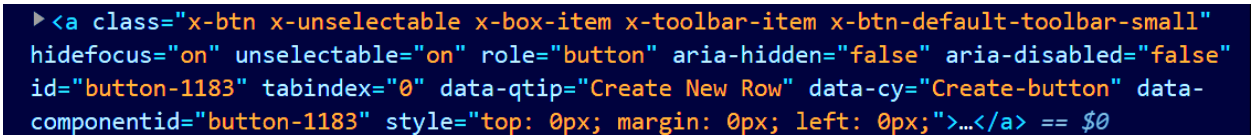



Figure 8: Create button HTML after data-cy tag was added

```
it('Create button test: class', () => {
  cy.get("div[data-cy-panel=ext-container]")
     .find("div[data-cy-panel=toolbar-container]")
     .find("a[data-cy=Create-button]").click();
});
```

While buttons were the most prominent element in the Table Manager application, as shown in Figure 1, there are many elements that need to be tagged such as drop down menus, containers, and toolbars. Our focus for this project was to identify these elements, find their source or override code, and add a tag for easier access for our testing and tests by future developers.

## 5. Quality Assurance:

Quality assurance was unique for our project, as we did not need to apply traditional unit or integration tests because we were writing user interface tests. That said, quality tests help to create quality products, so we needed to ensure that our tests were held up to the same standard as the code that was being tested. To achieve this standard, we implemented several code quality assurance techniques as we made our way through the class.

The first and most involved technique was effective pair programming. We did this when we first began testing the waters of Cypress and as we wrapped up our final tests to submit to our client. The type of pair programming used depended on the problem that was presented. In the first few weeks, we used the driver-navigator method, as it was beneficial to have another pair of eyes try to help stumble through Cypress and ExtJS code. It was also beneficial to have a partner who could focus on gathering information needed to help overcome an obstacle. The most common type, however, was ping-pong pair programming. When we started modifying ExtJS source, only one group member was able to understand Datava's code enough to implement the tags, either in the core code or through writing ExtJS overrides. To ensure the quality of their code, the rest of the team worked to continue testing the UI of applications to ensure the tags were applied to the correct elements, as well as search for new elements that needed said tags.

Another technique that worked in parallel with pair programming was our constant code review. Because of early troubles with Git, we decided it was easiest to all work out of the same directory through Visual Studio Code. While this did lead to some occasional issues where someone would accidentally run a test at the same time as someone else, it allowed members to view others' code at any time, rather than having to have to wait for a push. This allowed for fast reviewing and updating of code that was causing issues, or simply optimizing and changing tests whenever a better technique for accessing elements was achieved.

An additional tool was the Cypress user interface. The Cypress UI allows the user to step through their code while observing what the simulated Cypress user is doing on the website. The tool also produces before and after screenshots of individual test execution. These tools allowed us to ensure that test results do not have false positives, where a test technically passes but the user interface does not respond as predicted, or false negatives, where a test fails but the user interface responds correctly. Early on in the project a test passed when clicking the 'Create' button, however, when watching the video output, the expected behavior did not occur. The cypress UI allowed us to catch this false positive and helped us correct our test.

Finally, we met with Datava every day for 5 to 30 minutes for our daily standup. We used this time to keep the client informed of our progress, as well as to discuss any problems that we encountered. This was beneficial to our code quality as it allowed the client to propose solutions or their thoughts on how to approach and solve the issue at hand. The daily standup also ensured the project goals and objectives were understood by all parties and allowed for any alterations to the project to be made if needed. These meetings ensured that our team and our client were confident in the ability to meet project goals in the given timeframe.

## 6. Results:

The goal of this project was to make a framework for Datava software engineers to easily implement tests for their user interface. We accomplished this by creating in-depth documentation and by editing Datava's source code. The documentation includes instructions on how to implement and write tests in Cypress, explanations on how to properly change the source code, examples of Cypress tests, and common errors that can happen and how to fix them.

- **Performance Testing Results**
  - User interface tests often take a long time to run. Some of this is attributed to test design, however, much of the runtime comes from waiting for webpages or DOM elements to load. Cypress by default waits 4 seconds for a target element to load. This had to be changed to up to 20 seconds in some cases to allow for the loading of larger databases. This issue can be alleviated by running the server and database locally or on a faster connection to speed up loading times.

- **Summary of Testing / Results of Usability Tests**
  - Since the only code implemented in the project is user interface testing, there are no traditional unit tests or integration tests. Quality assurance was done mainly through peer programming, code review, and Cypress UI.
    - Cypress has a UI that allows the user to step through each part of the test which allows for easy debugging. Through this, it is possible to ensure that tests are functioning correctly.
  - Elements Tested:
    - Buttons: Tabs, drop down menus, non-labeled buttons, and file trees
    - Text boxes: Data, paragraph, and password input
    - Check boxes: Selecting and deselecting items
    - Sliders: Resizing Tables
    - Text: Making sure that the correct text is displayed and checking that text is correctly updated.
    - Containers: Used to narrow down which element is being clicked when there are multiple options with the same text or tag
    - Grids: Wrote a function to assert that a cell in a grid identified by row and column contains an expected value.

- **Features we did not have time to implement**
  - Adding data-cy tags for all elements of the user interface. By adding tags to all elements, it will allow for easier testing by specifying the element to be tested.
  - Many elements have not had the data-cy tags implemented:
    - Check boxes
    - Expand menu arrows
    - Rows and columns in tables
  - Robust tests for all elements:
    - Currently tests are app specific in function and in the future tests should be easily shareable across applications and elements.

## 7. Future Work:

Before we discuss future work it is important to talk about the lessons we learned from this project. These lessons learned are the reason that the project's objectives and goals changed, and they help outline ideas for future work on the project.

Lessons Learned:

- It is difficult to test dynamically generated user interfaces without useful static DOM attributes.
- Sometimes Cypress can click faster than a user and reveal bugs that are nonissues when a human is navigating the site.
- Cypress is very powerful.
- Our objective shifted as we learned and experimented with our client's system and Cypress. It is good to have clear communication so all parties know what is expected and what can be achieved.
- Cypress executes code faster than web pages are able to load so wait statements were hard coded into a few tests to make Cypress slow down. Wait statements are not good because if the server decides to be faster or is upgraded, the tests still run just as slow. We replaced wait by increasing the default timeout for finding elements and by adding waitUntil statements that relied on waiting for elements to show up instead of arbitrary wait times.

For future work on this project the following elements if implemented would allow a more complete user interface testing system. The elements listed below will support creating reliable and robust tests for all elements of Datava's user interface. These elements would also help achieve the final goal of creating a completed application for testing all of Datava's current applications and any future applications they may implement.

Future work:

- Create unique and static data-cy tags for all elements present which will span all elements to be specified during testing and will add reliability to testing
- Create outline tests for all classes of elements such as buttons or check boxes. This feature will allow for easy future implementation of tests for new features by giving developers an outline of a test.
- Create an application that generates a skeleton test script for developers. This application gives developers a starting point for test creation by automatically reading the UI's source code and creating a test outline for every element that needs to be tested.
- Give Cypress more direct access to the server so it does not need to log in manually every time

# 8. Appendix:

## 8.1. Product Installation Instructions

For Datava's server each developer had their own directory to work in. With this each developer must install Cypress in their own directory in order to perform testing. Guide for installing Cypress can be found on the Cypress webpage. All updated source code can be accessed by copying the files or by Git.

## 8.2. Coding conventions

While writing a Cypress test it is important to note every step a user would have taken to reach any element. Cypress must perform every action a user would in order to test the element properly to ensure Cypress is testing the same user behavior. This coding convention allows for Cypress to behave as a user would on the desired interface. Another convention that is important during testing is to add documentation of what element is being tested, that steps were taken to reach and test that element. This documentation can be done with commenting on each test and their steps or in a different document that records information about the tests.

It is also important to ensure no false positives or false negatives are being produced by the Cypress tests. To ensure the correct behavior Cypress provides screenshots of the interface before and after a test, a video that shows all steps being taken, and a GUI that allows the developer to step through each test. This convention helps ensure the desired behavior is being performed.

Coding conventions such as well documented and well thought out tests that perform their desired behavior add credibility to the code. With this credibility our code will be able to be used for testing on existing and future applications. Our documents allow for easy use by future developers to understand and expand on the tests if needed. Coding conventions are important to ensure good tests are being written.

## 8.3. User Interface Testing Process

Before writing a test it is best to go through the web application and take note of every action taken

- For example, if you wanted to create a test to ensure the Datava demo login page for the web application was working you first need to know all of the steps taken when trying to log in to the page. (Reference Figure 9)
- Enter the login URL
  Type in the:
  - Database name
  - User ID
  - Password
- Assert information displayed matches input information
- Click log in
- In order for Cypress to successfully login on to the web application it has to be told to take the five steps above.
- Another good practice is to reset the desktop after every test using a beforeEach method and 'Start > Reset' to ensure a failed test will not affect the functionality of future tests.

- This logic can be applied to any test, all steps a user would take to interact with an element cypress must take as well.

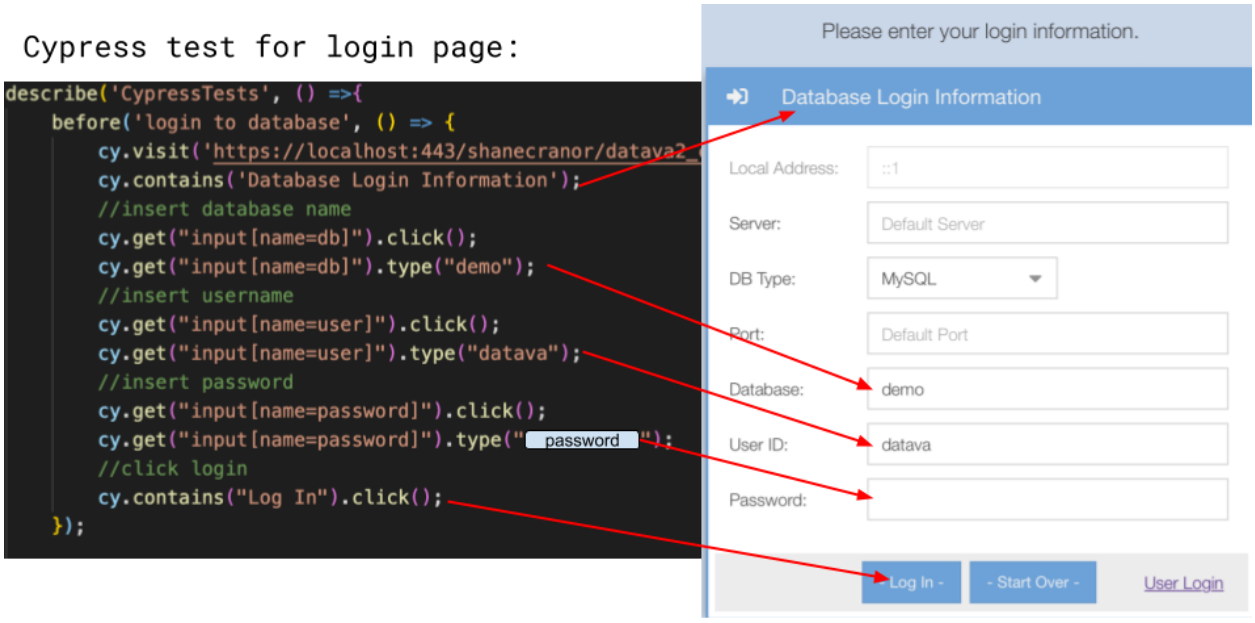



Figure 9: Example of basic Cypress function to log into database

## 8.4. Overall Team process

The initial idea behind the project was to create a testing interface/generator for each of Datava's applications. This eventually changed into creating individualized tests for each application manually to make sure that each one functioned properly. Once the writing of said tests began, two main problems emerged: too much time was spent writing in-depth tests for all of the different features in an app and it was not possible to select all of the desired elements because the only identifying features were the IDs that were dynamically generated by ExtJS, meaning they changed, and we could not reliably use them. Due to these problems, the focus of the project shifted to figuring out how to make it easier to create tests for elements that were difficult to identify and document everything the team learned about writing tests to assist others when writing tests in the future. To make it easier to access unlabeled objects on the user interface, the team worked on changing the source code and overriding files to add static tags to elements to access them during tests. The team created a document explaining our approach and techniques when writing Cypress tests.