UTAM Autogen Final Report

Team Members

Daniel Devine, Ian Nara, Andrew Taylor

Client

Matt Butland: Lightning Web Developer

June 15, 2021

Introduction

The UI Test Automation Model (UTAM), is a project developed by Salesforce with the goal of streamlining the process of testing web components. This system takes as input a JSON file that specifies to the UTAM system what we need to test and what methods we need to manipulate the web components. UTAM then automatically generates a javascript interface based on this input containing the methods necessary for testing, which effectively decouples and abstracts the functionality needed for testing from the actual HTML, Javascript, and CSS code of the developer's project. This interface makes it simple to test functional web components because a developer can simply use the provided methods to perform actions on the webpage, rather than self-authoring tests containing the detailed logic abstracted by UTAM. Currently these JSON specifications are authored by hand as a web developer examines their project and determines what they want UTAM to generate. Our project aims to automate this process by parsing the developer's HTML code and automatically constructing the JSON input file.

Requirements

Functional Requirements

The project should be able to generate UTAM JSON files for most web page objects. The composition of the JSON files should be similar to what someone would generate if they were writing the file by hand. Specific functionality includes:

- Ability to generate the JSON for the example web page in the repo for the project
- Generates at least 60-70% of the UTAM grammar syntax needed for a complete JSON file
- Able to generate root element, basic element, custom element, element selector, selector properties, listen index, compose method, basic actions, and argument types.
- Be able to generate mostly complete UTAM JSON files for larger projects
- The output must be accepted as valid input by UTAM.

Non-functional Requirements

- The program must be written in TypeScript.
- The generator must work well for and be stable for the basic web components and their corresponding UTAM input.
- UTAM JSON output should have a consistent format.
- The generator output should be consistent with what someone would create manually.

System Architecture

Overview

Our program accepts a webpage project directory as a command line input. We then recursively search this directory to locate the HTML files, and construct a Document Object Model (DOM) tree for each file using the ‘htmlparser2’ package. We then construct a JSON tree for each file based on the DOM. These trees are our way of representing the UTAM JSON before it is outputted to the file. This process consists of two steps.

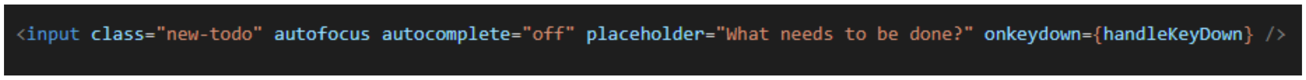
1. Extract interactable elements from the HTML by traversing the DOM tree.
2. Determine the methods and filters that must be constructed based on the extracted elements and their properties.

After the JSON trees are constructed, elements in the trees are grouped together based on sections in the HTML. For larger websites, the trees will need to be split up logically to ensure readability of output files. See Split by Section for more details. Once the trees are split, each tree is output to its corresponding UTAM JSON file. These files are constructed line by line from top to bottom by traversing the UTAM JSON root’s elements and methods as well as their corresponding sub-properties.

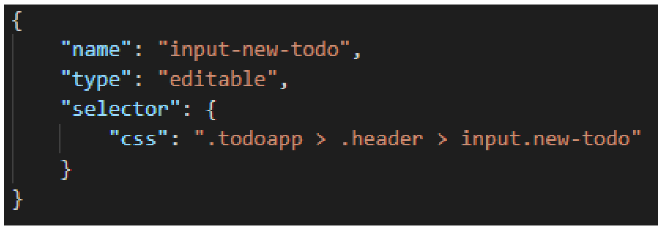
Extracting Elements

To determine whether a node is interactable, we pull properties found in the DOM tree such as the HTML tag and other attributes: the DOM event, if the node has a link specified by href, if the node has a type. Here is a list of the tags/attributes and their corresponding UTAM types:

- Tags: “button”: clickable, “input”: clickable or editable, “a” with attribute “href”: clickable
- Attributes: “onclick”: clickable, “onkeydown”: editable, “onchange”: clickable, “ondblclick”: editable
- Custom HTML tag (tag containing a dash): Extract package name from package.json file of developer’s project and extract component name from the tag. The type is defined as: “[package name]/pageObjects/[component name]”



```
<input class="new-todo" autofocus autocomplete="off" placeholder="What needs to be done?" onkeydown={handleKeyDown} />
```



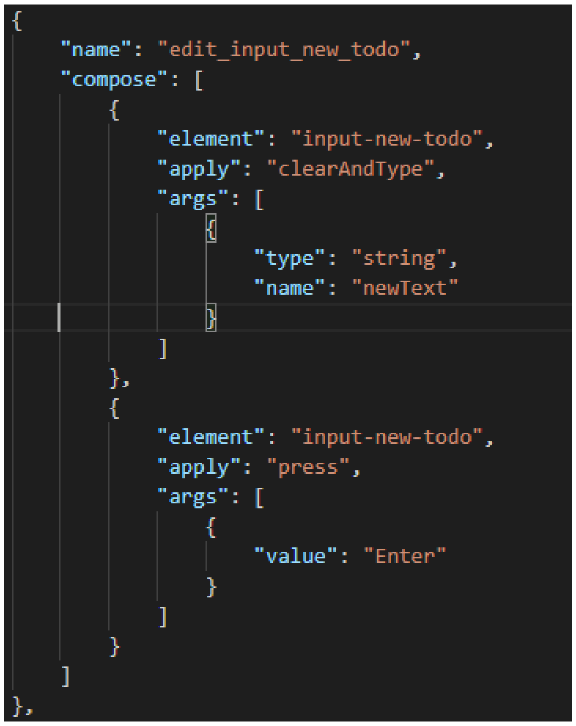
```
{
    "name": "input-new-todo",
    "type": "editable",
    "selector": {
        "css": ".todoapp > .header > input.new-todo"
    }
}
```

Example of a HTML element with the tag “input” and the “onkeydown” attribute and its corresponding UTAM element. The CSS selector is generated by pulling tag/class names from the HTML while recursively going down the DOM tree.

Making Methods

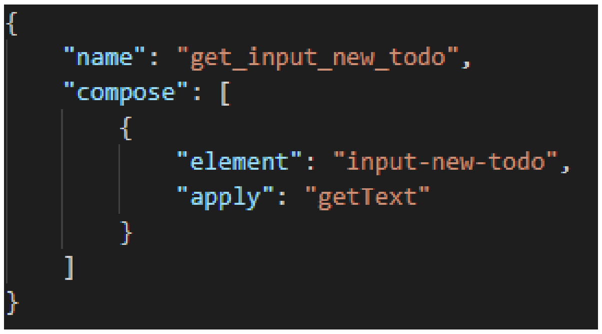
After creating a list of elements from the HTML, we examine these elements to determine what methods (if any) we need to generate for them. The methods are generated mainly by examining the types of the elements.

- clickable: click method which applies “click” to the element
- editable: edit method which applies “clearAndType” with some input text and presses Enter, getter method which applies “getText” to the element
- Custom: no method needed as methods will be generated in UTAM output for the HTML for that custom element



```
{
    "name": "edit_input_new_todo",
    "compose": [
        {
            "element": "input-new-todo",
            "apply": "clearAndType",
            "args": [
                {
                    "type": "string",
                    "name": "newText"
                }
            ]
        },
        {
            "element": "input-new-todo",
            "apply": "press",
            "args": [
                {
                    "value": "Enter"
                }
            ]
        }
    ]
},
```

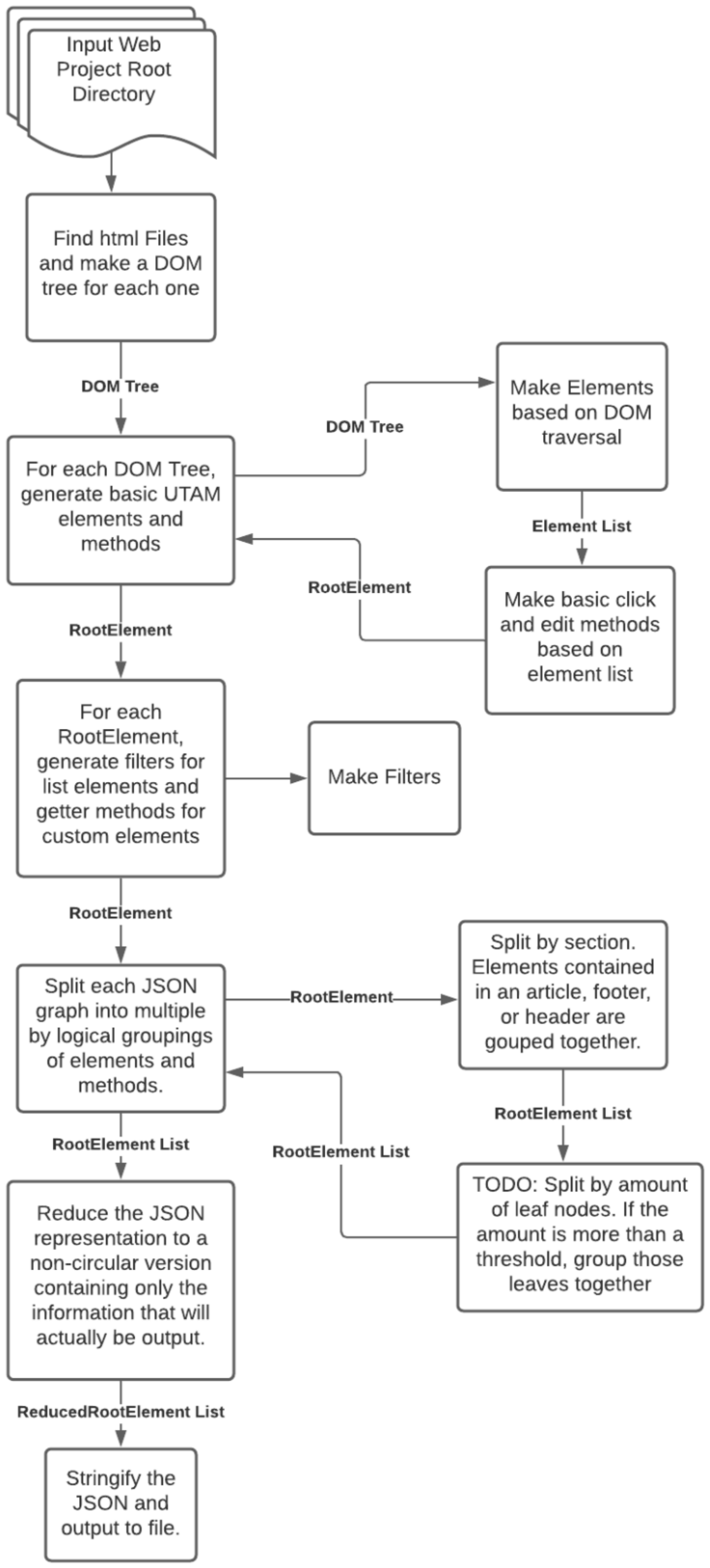
Edit method generated from the UTAM element above



```
{
    "name": "get_input_new_todo",
    "compose": [
        {
            "element": "input-new-todo",
            "apply": "getText"
        }
    ]
}
```

Getter method generated from the UTAM element above

System Architecture Flowchart


Input Web Project Root Directory
Find html Files and make a DOM tree for each one
DOM Tree
For each DOM Tree, generate basic UTAM elements and methods
DOM Tree
Make Elements based on DOM traversal
Element List
Make basic click and edit methods based on element list
RootElement
RootElement
For each RootElement, generate filters for list elements and getter methods for custom elements
Make Filters
RootElement
Split each JSON graph into multiple by logical groupings of elements and methods.
RootElement
Split by section. Elements contained in an article, footer, or header are gouped together.
RootElement List
TODO: Split by amount of leaf nodes. If the amount is more than a threshold, group those leaves together
RootElement List
RootElement List
Reduce the JSON representation to a non-circular version containing only the information that will actually be output.
ReducedRootElement List
Stringify the JSON and output to file.

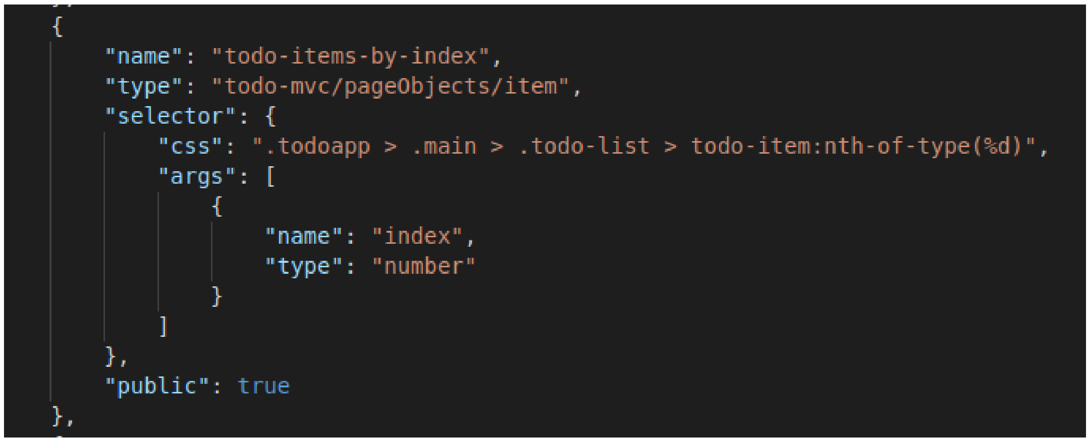
Technical Design

Filter Elements

For HTML elements that are lists, we need to specify to UTAM that we would like methods that allow us to filter the list for specific elements. In particular, we would like to be able to locate list elements based on a given index, and based on the contents of any list element. This would, for example, allow us to search a list of animals for ‘dog’, or to get element five out of a list. The ability to do this is helpful for testing because it allows us to check the edges of a list or to make sure a list has been ordered properly. It can also be used to check to make sure only certain items have been put in a list. To construct these elements, we are borrowing many properties from the original list element, but we also add several more specifications that tell UTAM how to compare and filter based on the list elements’ contents, and to filter based on a list elements’ index.

Example Filter Output

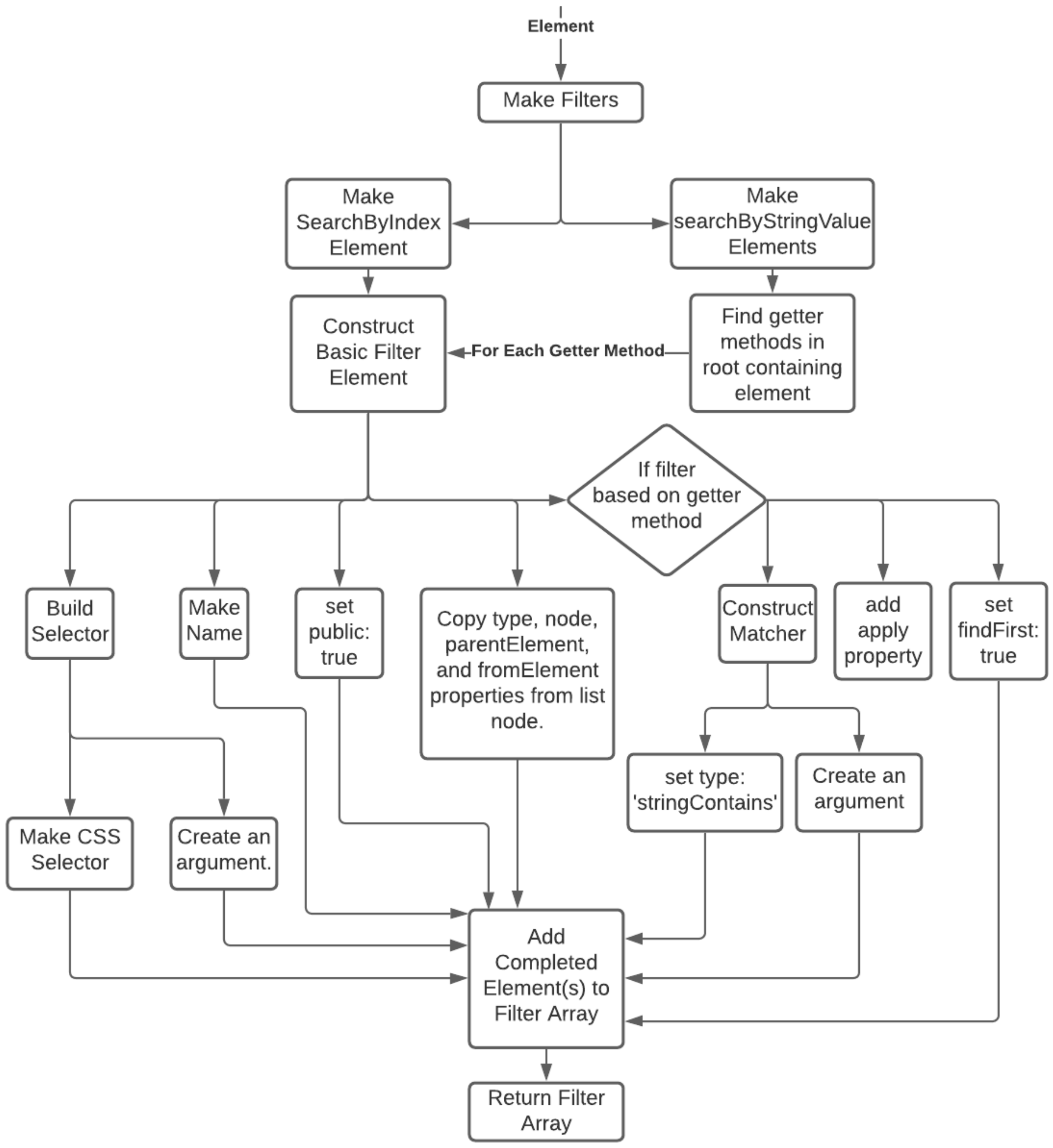
The example output below corresponds to a list of “item” elements in our HTML code. We are specifying to UTAM that we would like a javascript method to filter the list based on an item’s index. This information is contained in the “selector” property block. In our CSS selector, we specify that we would like to get a particular element in our list based on its index by appending “:nth-of-type(%d),” to the list’s CSS selector “... > todo-item”. We also specify that we would like the method to take in a number as a parameter, which will be called index.



```
    {
        "name": "todo-items-by-index",
        "type": "todo-mvc/pageObjects/item",
        "selector": {
            "css": ".todoapp > .main > .todo-list > todo-item:nth-of-type(%d)",
            "args": [
                {
                    "name": "index",
                    "type": "number"
                }
            ]
        },
        "public": true
    },
```

Example Filter Output

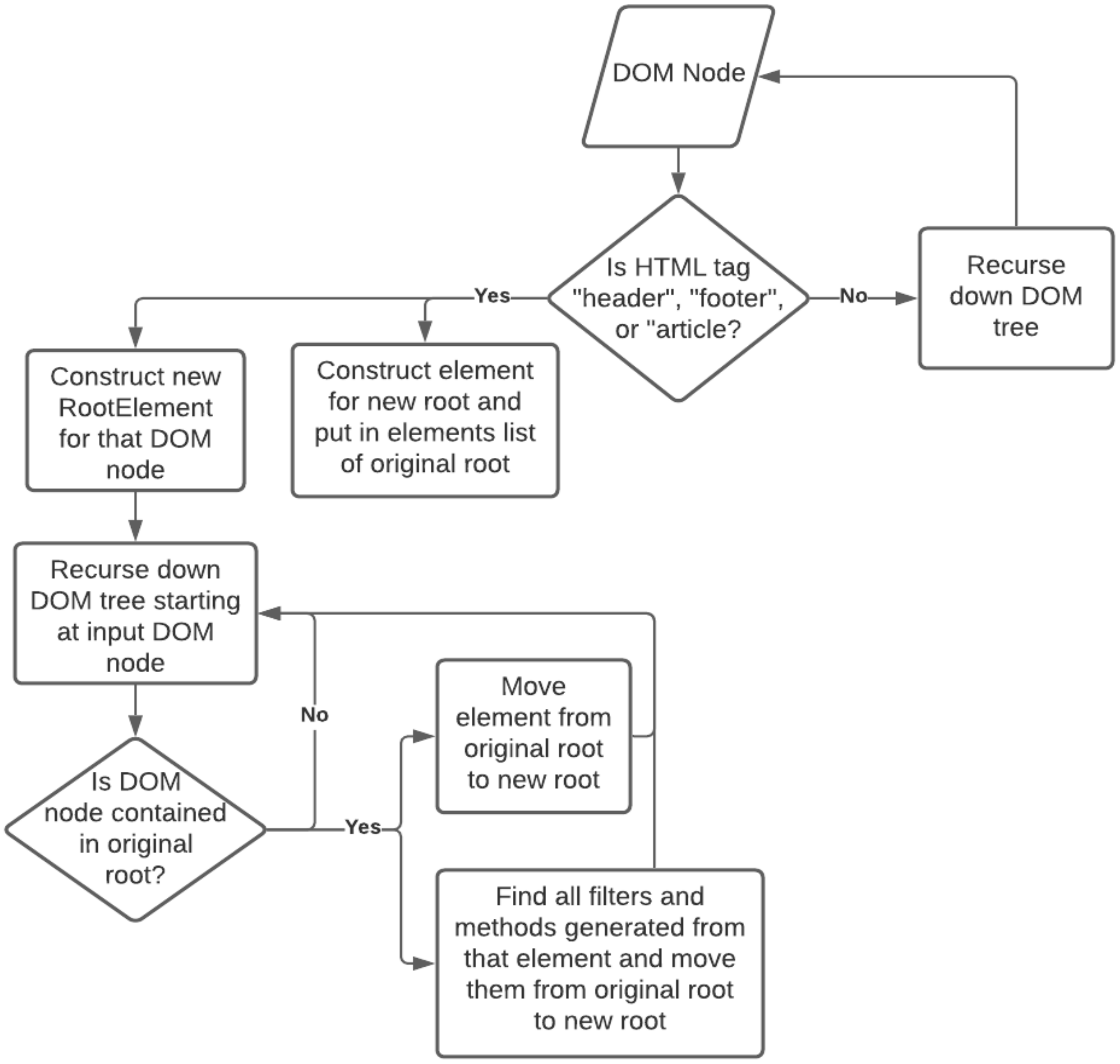
Filter Element Flow Chart


Element
Make Filters
Make SearchByIndex Element
Make searchByStringValue Elements
Construct Basic Filter Element
For Each Getter Method
Find getter methods in root containing element
If filter based on getter method
Build Selector
Make Name
set public: true
Copy type, node, parentElement, and fromElement properties from list node.
Construct Matcher
add apply property
set findFirst: true
Make CSS Selector
Create an argument.
set type: 'stringContains'
Create an argument
Add Completed Element(s) to Filter Array
Return Filter Array

Split By Section

For larger websites, multiple UTAM output files are needed for readability. The end user should not have to navigate through thousands of lines of UTAM JSON to find the element they are looking for. For this reason, logic was needed to split the output files in a way that makes sense. A simple way to do this is to search for HTML tags that indicate a large portion of the website is underneath them, specifically the tags “header”, “footer”, and “article”. The reason we chose these tags specifically is because they often will contain a sizable amount of interactable elements within them without containing the main part of the website. The main part of the website, usually contained under the tag “body” should be outputted into the main JSON file for less confusion.

Split By Section Flowchart


DOM Node
Is HTML tag "header", "footer", or "article?
Yes
No
Recurse down DOM tree
Construct new RootElement for that DOM node
Construct element for new root and put in elements list of original root
Recurse down DOM tree starting at input DOM node
No
Move element from original root to new root
Is DOM node contained in original root?
Yes
Find all filters and methods generated from that element and move them from original root to new root

Quality Assurance

Quality Assurance Activities

- End to End Testing: Testing was performed over the entire UTAM lifecycle, beginning from an input web project and ending with tests authored using the UTAM generated javascript interface.
- Code Reviews: Code was reviewed by each member of the team before a completed feature could be merged into the project's main branch.
- Pair Programming: We did pair programming by having 1 or 2 people coding at a time and screen sharing so the other(s) can watch.
- Output Reviews: Program outputs were reviewed by all team members to assure the naming and syntax look alike to what a developer would manually write.
- Client Acceptance Testing: The client examined the UTAM generation ability for each new feature and determined if the output was acceptable compared to what a developer would be expected to produce by hand.

Edge Cases

1. The HTML code of the developer's project is invalid, resulting in failure of DOM creation.
2. The HTML code of the developer's project contains characters/symbols that are not valid in javascript. (This includes things like the '-' character that would be interpreted as subtraction in javascript).
3. The developer's project does not contain .html files, or only contains empty .html files.
4. The developer's project has some valid .html files and some empty .html files.
5. The developer's project has no interactable elements.
6. The developer's project has references to a custom element that does not exist in their project directory.

Addressing Our Edge Cases

1. Our project is not concerned with handling syntactical or logical errors in a developer's input project, regardless of how minor they may be. Our approach to handling such inputs is to hard-fail with a message alerting the user that their input was not found to be valid HTML. We are able to do this early into execution as construction of the DOM is one of the first steps in the project. If any one HTML file is not valid, the entire program will hard-fail, as we may need access to the DOM of that HTML file in order to construct the JSON of another file.
2. To handle information extracted from the HTML containing symbols that would result in the generated javascript being invalid, we are performing substitution of invalid symbols with valid ones, such that no non-syntactic information is lost. (for example, test-html -> test_html). This will require some research and care to ensure that no invalid symbol patterns are missed.
3. If a project has no HTML files or has only empty HTML files, our program is unable to perform any useful work for the developer. Our response in this case is to alert the

developer that no HTML was found in the supplied directory and terminate, which occurs immediately into execution.

4. In the case that some .html files are empty and some are not, it is safe for our program to simply generate the UTAM JSON for those HTML files that contain HTML code. If a .html file is empty, no corresponding .json file will be produced for that file.
5. In the case that our program finds no elements in need of functional testing, our program will simply generate no .json file for that particular HTML file.
6. In the case that a custom element whose HTML code is not provided is referenced, the program will alert the developer about the missing information, and then hard-fail.

Results

Our project's goal was to create a tool that can convert a web developer's project into a UTAM JSON input file, increasing the convenience and ease of use of Salesforce's automated web component testing system. The complexity of this task turned out to be much greater than anticipated due to both the customizability of HTML and the customizability of UTAM. Our initial goal was to be able to generate 60-70% of the possible UTAM features for a given web project, including capability for: root element, basic element, custom element, element selector, selector properties, list and index, compose method, basic actions and argument types. As we began testing our project on a larger set of webpages, however, we found that in HTML, developers can achieve an end result in a multitude of different ways. These discrepancies all translate into the same UTAM, requiring the addition of more program logic for each UTAM feature to accommodate these different styles. Facing this problem, we had the option to either make our efforts intensive, and produce the essential elements for a larger set of webpages, or extensive, and cover a greater degree of the UTAM features, but for a smaller set of webpages and HTML coding styles. Our choice was to try and strike a balance between the two, but with the edge towards producing a tool that does a solid job on the basics.

Achieved Functionality

We can claim to have some generation capability for all of the UTAM features listed above. We cannot, however, claim to have full functionality for all of these. The functionality we have implemented includes:

- Basic Actions (partial functionality): We are able to generate UTAM for some, but not all of the possible basic actions.
- Argument Types (complete functionality): We are able to generate UTAM for the type, value, and name fields of an argument array.
- Compose Method (complete functionality): We are able to generate UTAM for the name, element, apply, and args fields.
- List and Index (complete functionality): We are able to generate the returnAll property and a filterByIndex element for list elements.
- Selector Properties (complete functionality): We are able to generate the UTAM for the CSS, args, and returnAll fields of a selector.

- Custom element (partial functionality): We have partial functionality for representing nested components.
- Basic Element (partial functionality): We are able to generate the UTAM for the elements, filter, name, public, selector, and type fields.
- Element Selector (complete functionality): We are able to generate the UTAM for an element selector.
- Splitting HTML files (partial functionality): HTML files are split into multiple JSON files based on logical groupings of components. Splitting is done by grouping elements contained in a header, footer, or article

It is important to note that we are not able to generate these UTAM features for every possible HTML input, due to the large number of ways a developer could represent components in their HTML. This is true both for features which we have implemented full functionality for and features which we have implemented partial functionality for. Full or partial functionality is merely a measure of how much of the possible JSON properties for a feature have been captured by logic in our program. It does not indicate that this logic is going to effectively capture every possible HTML representation of that feature.

Summary of Testing

For the first half of our project, we were given a sample input directory and sample output files. Our testing consisted of running our script on the input directory, and then comparing our output files with the sample output files. Our output file did not have to match the sample directly as the sample was manually created while ours was automatically generated, but it was required that our output be formatted the same and reference the same HTML elements. Once we were able to match the output file, for the rest of the project we cycled through: pulling a random major website such as facebook, google, or stack overflow, running our script on that website and then adapting our script to be able to handle the new elements and style of that website, all while making sure it did not break our handling of past websites. To consider our testing for a website to be a pass, we made sure that 100% of clickable and editable elements were located and had created methods.

What We Learned

Some notable things we learned during our time working on this project include:

- Handling package managers and project dependencies is an important skill. We spent a considerable amount of our total working time dealing with project dependency and package manager issues. Getting good at handling the way your project builds allows for more time spent actually coding.
- One should read the documentation carefully first before deciding what libraries to use. Although we ended up using the 'htmlparser2' package in the end as our parsing tool, this was only after trying to use the 'node-html-parser' and 'dom-parser' libraries. The dom-parser library turned out to be intended for a purpose different from our own, and the node-html-parser did not offer the degree of functionality we assumed it did when we started coding with it.
- Typescript and javascript.

- Some HTML and css.
- Don't build something that already exists, and it can be helpful to ask an experienced developer what they think about your plan to achieve something before you start coding. We created a rather complex and unscalable system for converting our data structure into a proper json file. After asking our client for advice on this subject, we found that the system we had been building already existed in a built-in, one-line command.

Future Work

Our project has been developed with the understanding that Salesforce is going to continue development after Field Session has ended. For this reason, in addition to leaving behind documentation on how the program functions, we have also created notes on things that will likely need to be added or modified in the code as more capabilities are added. Some important additions include:

- The algorithm for splitting a JSON tree into multiple trees can be expanded on to include more criteria for constructing logical groupings of elements and methods. In particular, one would likely want to extract leaf node HTML elements with a large number of siblings into their own cluster.
- The CSS selectors currently being generated for UTAM elements are highly specific. To make the generated interface more resistant to changes in the HTML, one could create an algorithm for generating CSS selectors that only uses the level of specificity necessary to uniquely identify the element.
- More testing needs to be performed to verify the system behaves correctly on inputs that require nested elements.

Because of the high degree of variability in HTML, which depends on a developer's personal preferences and opinions, it is unlikely that expansion of the project via building the base of rules will allow for the achievement of 100% accuracy in UTAM generation. For this reason, it may be more effective to explore a solution that utilizes a neural network to generate UTAM that is more accurate to what a human would come up with.

Appendices

Installation Instructions

1. Clone the github repository
2. `cd` into the repository and run `yarn` at the root to automatically install dependencies.

Usage Instructions

`utam-autogen <project root directory>` will execute the script. To build and run the project, `yarn br` will build the code in the utam-autogen package and execute it on its todo-mvc folder.

Coding Style

Coding style generally follows camelCase. Some other important conventions we are using include:

```
# Single space before opening brackets for a block.
for(let i = 0; i < x; i++) {

}

#  Else block is on same line as closing bracket of if block
if(x == y) {

} else {

}

#One line if statements have no brackets
if(x == y) return x;

#type specifiers have a space after the colon
Var test: boolean = true;

#math operations have spaces between them
Correct: x + y = i
Incorrect : x+y=i
```