CSC/ECE 517 Fall 2019 - M1952. Missing DOM features project

From PG_Wiki

Servo (https://servo.org/) is a modern, high-performance browser engine designed for both application and embedded use. The current version of Servo has a couple of issues. The first issue is the absence of the capability to parse the srcdoc (https://bocoup.com/blog/third-party-javascript-development-future#iframe-srcdoc) attribute in an iframe (https://www.w3schools.com/tags/tag_iframe.asp) tag in the HTML code. The second issue is that Servo does not have a named getter implemented in HTMLFormElement (https://html.spec.whatwg.org/multipage/forms.html#dom-form-nameditem) to reference the form elements by their id. The goal of this project is to implement these two functionalities in the current version of Servo.

Contents

Introduction

Servo

Servo (https://en.wikipedia.org/wiki/Servo_(software)) is an experimental browser engine that seeks to create a highly parallel environment, in which components such as rendering, layout, HTML parsing, image decoding, etc. are handled by fine-grained, isolated tasks. It leverages the memory safety properties and concurrency features of the Rust programming language.

Rust

Rust (https://en.wikipedia.org/wiki/Rust_(programming_language)) is a multi-paradigm systems programming language primarily developed focused making the browser safe and concurrently operable. Rust has been the "most loved programming language" in the Stack Overflow Developer Survey every year since 2016.

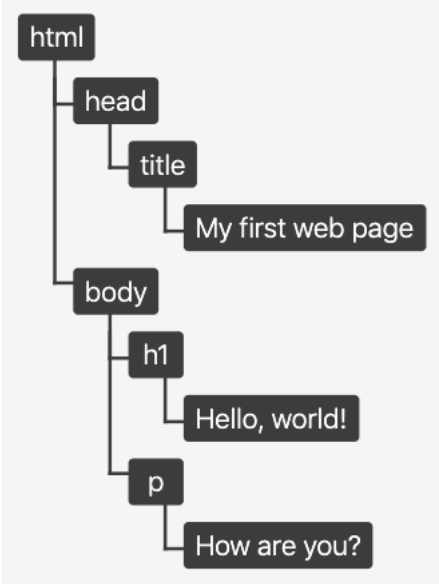
DOM

DOM, short for Document Object Model, is an interface and a way to how programs treat web pages. It parses the web pages in a structured order so that programs can read and manipulate the web page's content, structure, and style. When an HTML page is parsed, the programs build, what is called, a DOM tree and this lists all the HTML tags as nodes while maintaining the scope under which these tags might be defined.

bitsofcode (https://bitsofco.de/what-exactly-is-the-dom/) has an excellent read on the basics of DOM and here is a quick snapshot from the same: [Left - HTML page content; Right - DOM tree]

```
<!doctype html>
<html lang="en">
 <head>
   <title>My first web page</title>
  </head>
 <body>
    <h1>Hello, world!</h1>
    <p>How are you?</p>
  </body>
</html>
```

<----------->



Note to Reviewers

You would find that our code doesn't contain many comments. The current maintainer for the project advised us to remove comments which only read the code further and hence, to follow Servo's formatting and style guidelines, we removed these comments.

Setup

We need to compile and build Servo on our local machines to work on the code and check whether the tests pass. Servo's GitHub page has an excellent starting guide to set up the environment for Servo here (https://github.com/servo/servo/blob/master/README.md#setting-up-your-environment). It also mentions the other dependencies that need to be installed specific to an operating system.

Final Project

Problem Statement

We are working on the subsequent steps listed on the project page (https://github.com/servo/servo/wiki/Missing-DOM-features-project) which is the **named getter issue** (https://github.com/servo/servo/issues/16479). Currently, Servo is unable to submit forms on web pages since it is not able to fetch the form elements by their ID. Now, in terms of DOM and HTML, the HTMLFormElement (https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement) is the interface to the <form> tag in HTML. Hence, we need to implement the named element getter function in HTMLFormElement files.

Scope

The **named getter** issue has been worked upon as the subsequent steps.

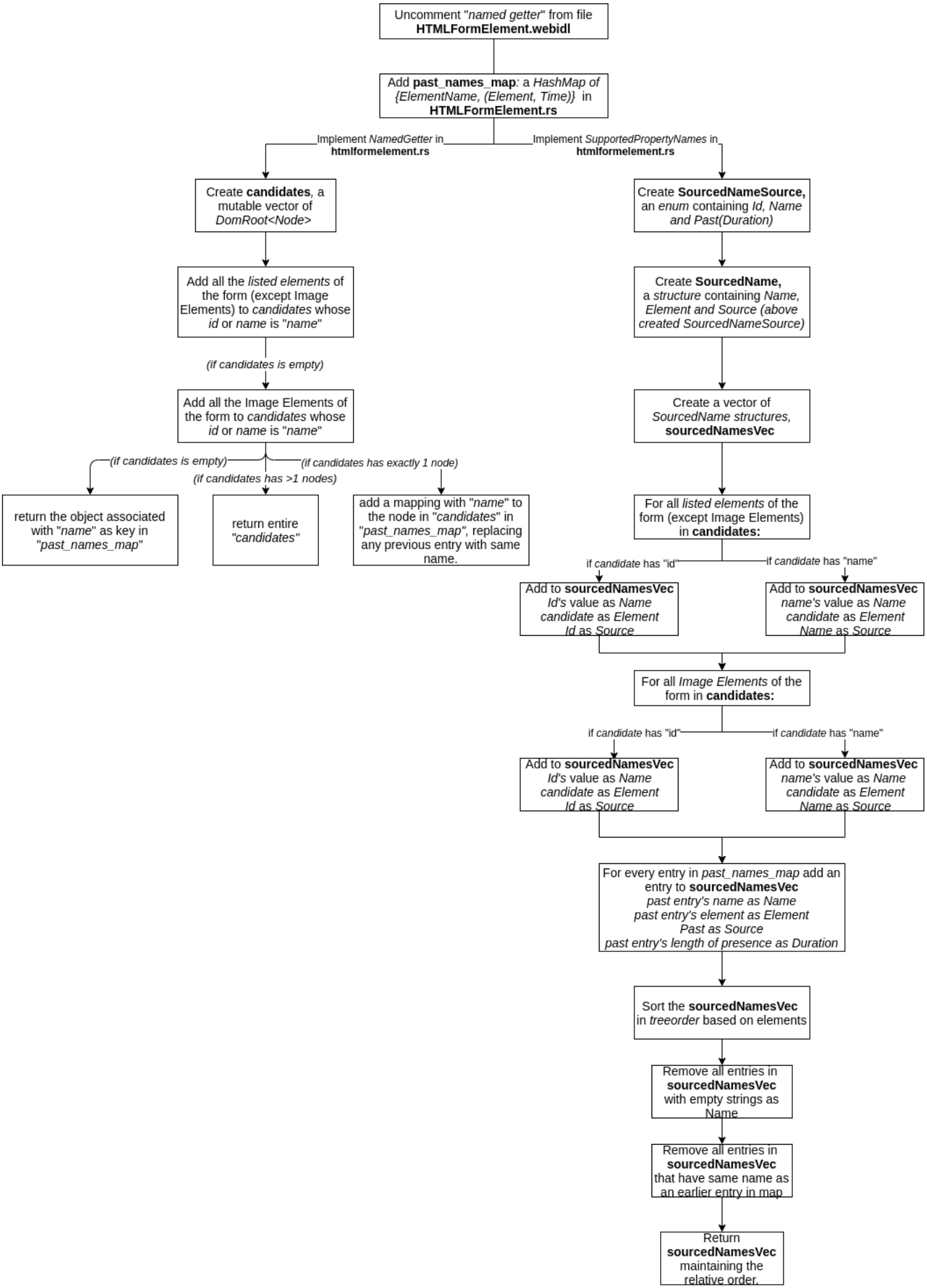
- Uncomment the **named getter** from HTMLFormElement.webidl file.
- The previous step yields two new methods that need to be implemented in htmlformelement.rs - **SupportedPropertyNames()** and **NamedGetter()**.
- **SupportedPropertyNames()** is used to get the list of all meaningful **property names** for a **HTMLFormElement** object.
- **NamedGetter()** gets the value of a specific property name.
- Both **NamedGetter()** and **SupportedPropertyNames()** are expected to read from the past names map, but only NamedGetter() is expected to modify it.
- Implement a **HashMap** from a **DOMString** (which holds an element's id/name value) to **Dom<Element>**. This Hashmap will be used in **NamedGetter()** to fetch the form node which has the given value in its **name** attribute. We will also update this Hashmap when we find a new node with this **name**. This Hashmap will also be used in **SupportedPropertyNames()** to extract the names of each form HTML element.

Design Patterns

Design patterns are not applicable as our task involves the implementation of methods and modifying various files. However, the Implementation section below provides details of why certain steps were implemented the way they were.

Flowchart

Given below is a high-level flowchart for the proposed solution:



Implementation

We have worked on the subsequent steps mentioned on the project page here (https://github.com/servo/servo/wiki/Missing-DOM-features-project).

Step 1: Uncomment the named getter from HTMLFormElement.webidl

The NamedGetter method was already declared. We uncommented those lines in the file **HTMLFormElement.webidl**. The function has the element's name as the attribute which is of type DOMString which stores a Rust String. It returns a **RadioNodeList** or **Element** based on the type of node returned.

2 components/script/dom/webidls/HTMLFormElement.webidl Viewed

```
@@ -29,7 +29,7 @@ interface HTMLFormElement : HTMLElement {
    [SameObject] readonly attribute HTMLFormControlsCollection elements;
    readonly attribute unsigned long length;
    getter Element? (unsigned long index);
-   //getter (RadioNodeList or Element) (DOMString name);
+   getter (RadioNodeList or Element) (DOMString name);

    void submit();
    [CEReactions]
```

Step 2: Add the missing NamedGetter and SupportedPropertyNames methods to htmlformelement.rs

We added the method definition for **NamedGetter()** based on the line uncommented in the previous step. **Option<RadioNodeListOrElement>** is Servo-specific syntax of returning either a RadioNodeList or Element from the function.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+    // https://html.spec.whatwg.org/multipage/#the-form-element%3Adetermine-the-value-of-a-named-property
+    fn NamedGetter(&self, name: DOMString) -> Option<RadioNodeListOrElement> {
```

We added the **SupportedPropertyNames()** method definition. It returns a vector of element names which are of type **DOMString** as in Rust.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+    // https://html.spec.whatwg.org/multipage/#the-form-element:supported-property-names
+    fn SupportedPropertyNames(&self) -> Vec<DOMString> {
```

Step 3: Implement SupportedPropertyNames() per the specification

(1) We define an Enum **SourcedNameSource** of sources from where elements are inserted into the vector. If the element entry is made when we find the **id** attribute, then the source will be selected as **Id** and similarly for **Name** as well. The **past** option is used when the entry is fetched from **past names map**.

Per the specification, we need to maintain an ordered list of tuples called **sourced names (string, element, source(duration))**. We incorporate this by maintaining a vector of struct called **SourcedName** which stores name, element and its source.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+    // https://html.spec.whatwg.org/multipage/#the-form-element:supported-property-names
+    fn SupportedPropertyNames(&self) -> Vec<DOMString> {
+        // Step 1
+        enum SourcedNameSource {
+            Id,
+            Name,
+            Past(Duration),
+        }
+
+        struct SourcedName {
+            name: DOMString,
+            element: DomRoot<Element>,
+            source: SourcedNameSource,
+        }
+
+        let mut sourcedNamesVec: Vec<SourcedName> = Vec::new();
```

(2) We loop over the **child elements** of form by calling **controls.iter()**. We first insert entries into the vector for **children** that are **listed elements** (non-image elements). We check whether that the child contains the **id** attribute and **name** attribute and insert the entry by defining a new structure object and pushing it to the vector. The value of **id** and **name** are fetched using **get_string_attribute()** function.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
353 +         // Step 2
354 +         for child in controls.iter() {
355 +             if child
356 +                 .downcast::<HTMLElement>()
357 +                 .map_or(false, |c| c.is_listed_element())
358 +             {
359 +                 if child.has_attribute(&local_name!("id")) {
360 +                     let entry = SourcedName {
361 +                         name: child.get_string_attribute(&local_name!("id")),
362 +                         element: DomRoot::from_ref(&*child),
363 +                         source: SourcedNameSource::Id,
364 +                     };
365 +                     sourcedNamesVec.push(entry);
366 +                 }
367 +                 if child.has_attribute(&local_name!("name")) {
368 +                     let entry = SourcedName {
369 +                         name: child.get_string_attribute(&local_name!("name")),
370 +                         element: DomRoot::from_ref(&*child),
371 +                         source: SourcedNameSource::Name,
372 +                     };
373 +                     sourcedNamesVec.push(entry);
374 +                 }
375 +             }
376 +         }
```

(3) We repeat the same process as mentioned in (2) but now for the image elements in the form.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
378 +         // Step 3
379 +         for child in controls.iter() {
380 +             if child.is::<HTMLImageElement>() {
381 +                 if child.has_attribute(&local_name!("id")) {
382 +                     let entry = SourcedName {
383 +                         name: child.get_string_attribute(&local_name!("id")),
384 +                         element: DomRoot::from_ref(&*child),
385 +                         source: SourcedNameSource::Id,
386 +                     };
387 +                     sourcedNamesVec.push(entry);
388 +                 }
389 +                 if child.has_attribute(&local_name!("name")) {
390 +                     let entry = SourcedName {
391 +                         name: child.get_string_attribute(&local_name!("name")),
392 +                         element: DomRoot::from_ref(&*child),
393 +                         source: SourcedNameSource::Name,
394 +                     };
395 +                     sourcedNamesVec.push(entry);
396 +                 }
397 +             }
398 +         }
```

(4) We borrow a reference to the **past names map** and iterate over the hashmap. We push the entry of **past names map** into the **sourcedNamesVec** by defining a new structure with **key value** as string, **HTML element** as element and **Past** as source with **duration** calculated.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
400 +         // Step 4
401 +         let past_names_map = self.past_names_map.borrow();
402 +         for (key, val) in past_names_map.iter() {
403 +             let entry = SourcedName {
404 +                 name: key.clone(),
405 +                 element: DomRoot::from_ref(&*val.0),
406 +                 source: SourcedNameSource::Past(now()-val.1), // calculate difference now()-val.1 to find age
407 +             };
408 +             sourcedNamesVec.push(entry);
409 +         }
```

(5) We sort the sourced names vector by comparing the element in tree order by using the **sort_by()** function of vector and the **cmp()** method in **PartialOrd trait** in Servo. We are able to sort entries with same element in order of **id**, **name** and at the end, put older entries before when the source is **Past**.

222 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+        // Step 5
+        // TODO need to sort as per spec.
+        // if a.CompareDocumentPosition(b) returns 0 that means a=b in which case
+        // the remaining part where sorting is to be done by putting entries whose source is id first,
+        // then entries whose source is name, and finally entries whose source is past,
+        // and sorting entries with the same element and source by their age, oldest first.
+
+        // if a.CompareDocumentPosition(b) has set NodeConstants::DOCUMENT_POSITION_FOLLOWING
+        // (this can be checked by bitwise operations) then b would follow a in tree order and
+        // Ordering::Less should be returned in the closure else Ordering::Greater
+
+        sourcedNamesVec.sort_by(|a, b| {
+            if a.element
+                .upcast::<Node>()
+                .CompareDocumentPosition(b.element.upcast::<Node>()) ==
+                0
+            {
+                if a.source.is_past() && b.source.is_past() {
+                    b.source.cmp(&a.source)
+                } else {
+                    a.source.cmp(&b.source)
+                }
+            } else {
+                if a.element
+                    .upcast::<Node>()
+                    .CompareDocumentPosition(b.element.upcast::<Node>()) &
+                    NodeConstants::DOCUMENT_POSITION_FOLLOWING ==
+                    NodeConstants::DOCUMENT_POSITION_FOLLOWING
+                {
+                    std::cmp::Ordering::Less
+                } else {
+                    std::cmp::Ordering::Greater
+                }
+            }
+        });
```

(6) As per the spec, we remove the elements which have the empty string as their name from the **sourcedNamesVec**. This is implemented by doing the **inverse operation**: **retain** only those elements which don't have **empty string** as their name.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+        // Step 6
+        sourcedNamesVec.retain(|sn| !sn.name.to_string().is_empty());
```

(7-8) We remove the entries in **sourcedNamesVec** that have the same name as an earlier entry in the map. We return just the vector of element names. Since our **sourcedNamesVec** consists of the **structure**, we just extract the element names from the structure vector and push it to a new vector which stores the **DOMString**s.

222 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+        // Step 7-8
+        let mut namesVec: Vec<DOMString> = Vec::new();
+        for elem in sourcedNamesVec.iter() {
+            if namesVec
+                .iter()
+                .find(|name| name.to_string() == elem.name.to_string())
+                .is_none()
+            {
+                namesVec.push(elem.name.clone());
+            }
+        }
+
+        return namesVec;
+    }
```

Step 4: Implement NamedGetter() function per the specification

(1) We need **candidates** to be a live **RadioNodeList**. However, operations are better defined for a vector and hence we define a vector that stores the Node itself and will convert it to a RadioNodeList when we return from the function.

We iterate over the **form** children by borrowing the **controls** member and check if the child is a **listed element (non-image element)**. If yes, we check whether the child has an **id** attribute or a **name** attribute equal to **name passed as parameter**. If yes, we push this child to the **candidates** vector.

```
+    // https://html.spec.whatwg.org/multipage/#the-form-element%3Adetermine-the-value-of-a-named-property
+    fn NamedGetter(&self, name: DOMString) -> Option<RadioNodeListOrElement> {
+        let mut candidates: Vec<DomRoot<Node>> = Vec::new();
+
+        let controls = self.controls.borrow();
+        // Step 1
+        for child in controls.iter() {
+            if child
+                .downcast::<HTMLElement>()
+                .map_or(false, |c| c.is_listed_element())
+            {
+                if child.has_attribute(&local_name!("id")) ||
+                    (child.has_attribute(&local_name!("name")) &&
+                        child.get_string_attribute(&local_name!("name")) == name)
+                {
+                    candidates.push(DomRoot::from_ref(&*child.upcast::<Node>()));
+                }
+            }
+        }
```

(2) If the vector **candidates** is empty, we repeat the same thing as we did in step 1 but now for image elements.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+        // Step 2
+        if candidates.len() == 0 {
+            for child in controls.iter() {
+                if child.is::<HTMLImageElement>() {
+                    if child.has_attribute(&local_name!("id")) ||
+                        (child.has_attribute(&local_name!("name")) &&
+                            child.get_string_attribute(&local_name!("name")) == name)
+                    {
+                        candidates.push(DomRoot::from_ref(&*child.upcast::<Node>()));
+                    }
+                }
+            }
+        }
```

(3) If the **candidates** vector is empty, we infer that the element we are seeking is not in the current form DOM tree and hence, we return the **element** associated with name from **past names map** by formatting it as a **Element**.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+        // Step 3
+        if candidates.len() == 0 {
+            if past_names_map.contains_key(&name) {
+                return Some(RadioNodeListOrElement::Element(DomRoot::from_ref(
+                    &*past_names_map.get(&name).unwrap().0,
+                )));
+            }
+            return None;
+        }
```

(4) If **candidates** vector contains more than 1 node, we return the **candidates** itself by formatting it as a **RadioNodeList**.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
+        // Step 4
+        if candidates.len() > 1 {
+            let window = window_from_node(self);
+
+            return Some(RadioNodeListOrElement::RadioNodeList(
+                RadioNodeList::new_simple_list(&window, candidates.into_iter()),
+            ));
+        }
```

(5) At this point, **candidates** has exactly one node. We insert the (name, element) pair into past names map and update the entry with the same name if it exists.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
318 +         // Step 5
319 +         let element_node = &candidates[0];
320 +         past_names_map.insert(
321 +             name,
322 +             (
323 +                 Dom::from_ref(&*element_node.downcast::<Element>().unwrap()),
324 +                 now(),
325 +             ),
326 +         );
```

(6) We return the **single node** in **candidates** by formatting it as a **Element**.

174 components/script/dom/htmlformelement.rs 100755 → 100644 Viewed

```
328 +         // Step 6
329 +         return Some(RadioNodeListOrElement::Element(DomRoot::from_ref(
330 +             &*element_node.downcast::<Element>().unwrap(),
331 +         )));
332 +     }
```

Test Plan

The tests for **named getter issue** have already been written. We need to check whether the modifications we make to the code can still pass these tests.

The tests will be run using the mach utility (https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/mach) commands:

```
./mach test-wpt tests/wpt/web-platform-tests/html/semantics/forms/the-form-element/form-elements-nameditem-01.html
./mach test-wpt tests/wpt/web-platform-tests/html/semantics/forms/the-form-element/form-elements-nameditem-02.html
./mach test-wpt tests/wpt/web-platform-tests/html/semantics/forms/the-form-element/form-nameditem.html
```

Before our implementation, 6/17 tests were passing. While, After our implementation, 11/17 tests are passing:

Results - Before:

Summary

Harness status: OK

Found 17 tests

☑ 6 Pass
☑ 11 Fail

Details

Result	Test Name	Message
PASS	Forms should not have an item method	
PASS	Forms should not have a namedItem method	
FAIL	Name for a single element should work	assert_equals @http://web Test.protot test@http:/ @http://web
FAIL	Calling item() on the NodeList returned from the named getter should work	form.radio is @http://web Test.protot test@http:/ @http://web
FAIL	Indexed getter on the NodeList returned from the named getter should work	form.radio is @http://web Test.protot test@http:/ @http://web
PASS	Invoking a legacycaller on the NodeList returned from the named getter should not work	
FAIL	All listed elements except input type=image should be present in the form	assert_equals @http://web Test.protot test@http:/ @http://web
FAIL	Named elements should override builtins	assert_equals function "func @http://web @http://web Test.protot test@http:/ @http://web

Results - After:

Summary

Harness status: OK

Found 17 tests

☑ 11 Pass
☑ 6 Fail

Details

Result	Test Name	Message
PASS	Forms should not have an item method	
PASS	Forms should not have a namedItem method	
FAIL	Name for a single element should work	assert_ec undefine @http:/ Test.pr test@ht @http:/
PASS	Calling item() on the NodeList returned from the named getter should work	
PASS	Indexed getter on the NodeList returned from the named getter should work	
PASS	Invoking a legacycaller on the NodeList returned from the named getter should not work	
FAIL	All listed elements except input type=image should be present in the form	assert_ec input> @http:/ Test.pr test@ht @http:/
FAIL	Named elements should override builtins	assert_ec (function @http:/ @http:/ Test.pr test@ht

To test whether the code change works, follow the steps as outlined.

1. Install the pre-requisites required for servo as mentioned here (https://github.com/servo/servo/blob/master/README.md)
2. Clone our GitHub repo: `git clone https://github.com/cagandhi/servo`
3. Navigate to servo's directory: `cd servo`
4. Checkout the git branch **iframe-srcdoc**: `git checkout named-form-getter`
5. Check if code follows style guidelines: `./mach test-tidy`
6. Check if code has no compilation errors: `./mach check`
7. Check if servo is built successfully: `./mach build --dev --verbose`
8. Check if tests pass, i.e. servo can process **named form getter**: `./mach test-wpt tests/wpt/web-platform-tests/html/semantics/forms/the-form-element/form-elements-nameditem-01.html`

You will see that the servo build is successful but currently, the tests might fail.

Pull Request

Here is the link to our pull request (https://github.com/servo/servo/pull/25070). We have attached the code snippets for the changes made in files in the PR. The pull request has been merged into the master branch of Servo.

OSS Project

Problem Statement

We have worked on the initial steps of the project page (https://github.com/servo/servo/wiki/Missing-DOM-features-project) which is the **srcdoc iframe issue** (https://github.com/servo/servo/issues/4767). In HTML, there is a tag called **<iframe>** which allows you to embed a web page into another web page. This attribute has attributes like **src** and **srcdoc** which can be used to embed web pages. However, the uses of both attributes are different.

To embed a web page using **src** attribute, we need to provide a URL of the web page to be embedded. This works in Servo.

To embed a web page using **srcdoc** attribute, all we need to provide is just HTML content and it works even without adding **<html>** and **<body>** tags. This does not work in Servo. We have worked upon this issue for our OSS project.

Scope

The **srcdoc iframe** issue is to be done as initial steps.

- Uncomment the **srcdoc WebIDL attribute** and implement the attribute getter.
- Add a field to structure **LoadData** for storing the srcdoc contents when loading a srcdoc iframe.
- Add a new method to **script_thread.rs** which loads the special **about:srcdoc** URL per the specification.
- Call this new method from **handle_new_layout** when it's detected that a srcdoc iframe is being loaded.
- In **process_the_iframe_attributes**, implement the srcdoc specification so that LoadData initiates a srcdoc load.
- In **attribute_mutated**, ensure that changing the srcdoc attribute of an iframe element follows the specification.

Design Patterns

Design patterns are not applicable as our task involves the implementation of methods and modifying various files. However, the Implementation section below provides details of why certain steps were implemented the way they were.

Implementation

We have worked on the initial steps mentioned on the project page here (https://github.com/servo/servo/wiki/Missing-DOM-features-project).

Step 1: Uncomment srcdoc WebIDL attribute and implement the attribute getter

The **srcdoc** attribute was already declared. We simply uncommented those lines in the file HTMLIFrameElement.webidl (https://github.com/servo/servo/compare/master...jaymodi98:iframe-srcdoc#diff-375e9b33977b4ed4d088e14bbb752bb3).

components/script/dom/webidls/HTMLIFrameElement.webidl

```
@@ -9,8 +9,8 @@ interface HTMLIFrameElement : HTMLElement {
     [CEReactions]
             attribute USVString src;
-    //[CEReactions]
-    //         attribute DOMString srcdoc;
+    [CEReactions]
+             attribute DOMString srcdoc;

     [CEReactions]
     attribute DOMString name;
```

We implemented the attribute getter in the file htmliframeelement.rs (https://github.com/servo/servo/blob/df9bc08e33155b0a12f39c4674e5a8bd1df56d99/components/script/dom/htmliframeelement.rs). It basically defines a new Element which stores the srcdoc String in its attribute and its value is returned by the getter. The lack of a semi-colon in the last line of a **Rust function** denotes that the value of the variable be returned from the function.

```
110 -    fn get_srcdoc(&self) -> String {
111 -        let element = self.upcast::<Element>();
112 -        String::from(element.get_string_attribute(&local_name!("srcdoc")))
113 -    }
114 -
```

Since this attribute getter is called only at one place in the entire codebase in **process_the_iframe_attributes()** function, it was suggested to us that we make the function inline and we did the change in lines 245, 246 in our latest commit.

```
245 +            let element = self.upcast::<Element>();
246 +            load_data.srcdoc = String::from(element.get_string_attribute(&local_name!("srcdoc")));
```

Step 2: Add a field to LoadData for storing the srcdoc contents when loading a srcdoc iframe

We added a public field **srcdoc** of String type in the line 170 in file lib.rs (https://github.com/servo/servo/compare/master...jaymodi98:iframe-srcdoc#diff-bf561a46a499a0c2dc837bea89df1be0). We declared srcdoc of type DOMString in the webidl file and we are mapping the same field in the rust file. The data type DOMString is inherently a Rust String as can be seen here (https://doc.servo.org/script/dom/bindings/str/struct.DOMString.html).

6 components/script_traits/lib.rs

```
@@ -163,6 +163,11 @@ pub struct LoadData {
163 163         pub referrer: Option<Referrer>,
164 164         /// The referrer policy.
165 165         pub referrer_policy: Option<ReferrerPolicy>,
166     +
167     +     /// add a field to store srcdoc contents - init step 3; since srcdoc is a DOMString which is
168     +     /// inherently a Rust String - https://doc.servo.org/script/dom/bindings/str/struct.DOMString.html
169     +     /// https://doc.rust-lang.org/rust-by-example/std/str.html#targetText=Strings,in%20Rust%3A%20String%20and%20%26str%20.&target
170     +     pub srcdoc: String,
166 171   }
167 172
168 173   /// The result of evaluating a javascript scheme url.
@@ -194,6 +199,7 @@ impl LoadData {
194 199               js_eval_result: None,
195 200               referrer: referrer,
196 201               referrer_policy: referrer_policy,
202     +             srcdoc: "".to_string(),
197 203           }
198 204       }
199 205   }
```

Step 3: Add a new method to script_thread.rs which loads the special about:srcdoc URL per the specification

We defined a method **page_load_about_srcdoc** which is based on the method **start_page_load_about_blank** in the file script_thread.rs (https://github.com/servo/servo/compare/master...jaymodi98:iframe-srcdoc#diff-1879ac6bd5d567e2aa43529d33474677) and handles the loading of iframe tag with srcdoc property.

Effectively, we parse the about:srcdoc URL and set the URL in the context of the response which we load. Modern web browsers send responses in chunks and this is why we send the srcdoc content (an HTML string) in the chunk of the response.

24 components/script/script_thread.rs

```
3829 3832         }
3830 3833
3834     +     /// Synchronously parse a srcdoc document from a giving HTML string.
3835     +     fn page_load_about_srcdoc(&self, incomplete: InProgressLoad, src_doc: String) {
3836     +         let id = incomplete.pipeline_id;
3837     +
3838     +         self.incomplete_loads.borrow_mut().push(incomplete);
3839     +
3840     +         let url = ServoUrl::parse("about:srcdoc").unwrap();
3841     +         let mut context = ParserContext::new(id, url.clone());
3842     +
3843     +         let mut meta = Metadata::default(url);
3844     +         meta.set_content_type(Some(&mime::TEXT_HTML));
3845     +
3846     +         let chunk = src_doc.into_bytes();
3847     +
3848     +         context.process_response(Ok(FetchMetadata::Unfiltered(meta)));
3849     +         context.process_response_chunk(chunk);
3850     +         context.process_response_eof(Ok(ResourceFetchTiming::new(ResourceTimingType::None)));
3851     +     }
3852     +
```

Step 4: Call this new method from handle_new_layout when it's detected that a srcdoc iframe is being loaded

We already defined the method **page_load_about_srcdoc** in the above step. This function **handle_new_layout** is responsible for loading new data and redirecting the navigation to the relevant function based on the URL. If the structure **LoadData** has **about:srcdoc** in its **url** parameter, we pass in the new load and srcdoc string stored in LoadData.

35 components/script/script_thread.rs Viewed

```
@@ -2429,6 +2429,8 @@ impl ScriptThread {
            );
            if load_data.url.as_str() == "about:blank" {
                self.start_page_load_about_blank(new_load, load_data.js_eval_result);
+           } else if load_data.url.as_str() == "about:srcdoc" {
+               self.page_load_about_srcdoc(new_load, load_data.srcdoc);
            } else {
                self.pre_page_load(new_load, load_data);
            }
```

Step 5: In process_the_iframe_attributes, implement the srcdoc specification so that LoadData initiates a srcdoc load

We added the processing of srcdoc specification in **process_the_iframe_attributes()** function in this file htmliframeelement.rs (https://github.com/servo/servo/compare/master...jaymodi98:iframe-srcdoc#diff-498ce74d806ab54484e768d9237a53b1) by referring the specification (https://html.spec.whatwg.org/multipage/iframe-embed-object.html#process-the-iframe-attributes) and with help from Josh (https://github.com/servo/servo/pull/24576/#discussion_r340083953).

49 components/script/dom/htmliframeelement.rs Viewed

```
@@ -227,7 +227,30 @@ impl HTMLIFrameElement {

    /// <https://html.spec.whatwg.org/multipage/#process-the-iframe-attributes>
    fn process_the_iframe_attributes(&self, mode: ProcessingMode) {
-       // TODO: srcdoc
+       if self
+           .upcast::<Element>()
+           .has_attribute(&local_name!("srcdoc"))
+       {
+           let url = ServoUrl::parse("about:srcdoc").unwrap();
+           let document = document_from_node(self);
+           let window = window_from_node(self);
+           let pipeline_id = Some(window.upcast::<GlobalScope>().pipeline_id());
+           let mut load_data = LoadData::new(
+               LoadOrigin::Script(document.origin().immutable().clone()),
+               url,
+               pipeline_id,
+               Some(Referrer::ReferrerUrl(document.url())),
+               document.get_referrer_policy(),
+           );
+           let element = self.upcast::<Element>();
+           load_data.srcdoc = String::from(element.get_string_attribute(&local_name!("srcdoc")));
+           self.navigate_or_reload_child_browsing_context(
+               load_data,
+               NavigationType::InitialAboutBlank,
+               HistoryEntryReplacement::Disabled,
+           );
+           return;
+       }
```

We first check if the HTML element has the srcdoc attribute or not. In our case, we are processing the iframe HTML element and so `self.upcast::<Element>()` returns the iframe element's ID. We fetch the document to be shown on the window and store the ID of the incomplete process which we are currently executing. This is required since the browser processes are highly parallel. Next, we define a new LoadData instance and set its srcdoc property to that fetched by the **attribute getter** we implemented in **Step 1**. We then set the browsing context with the new attribute values.

Step 6: In attribute_mutated, ensure that changing the srcdoc attribute of an iframe element follows the specification

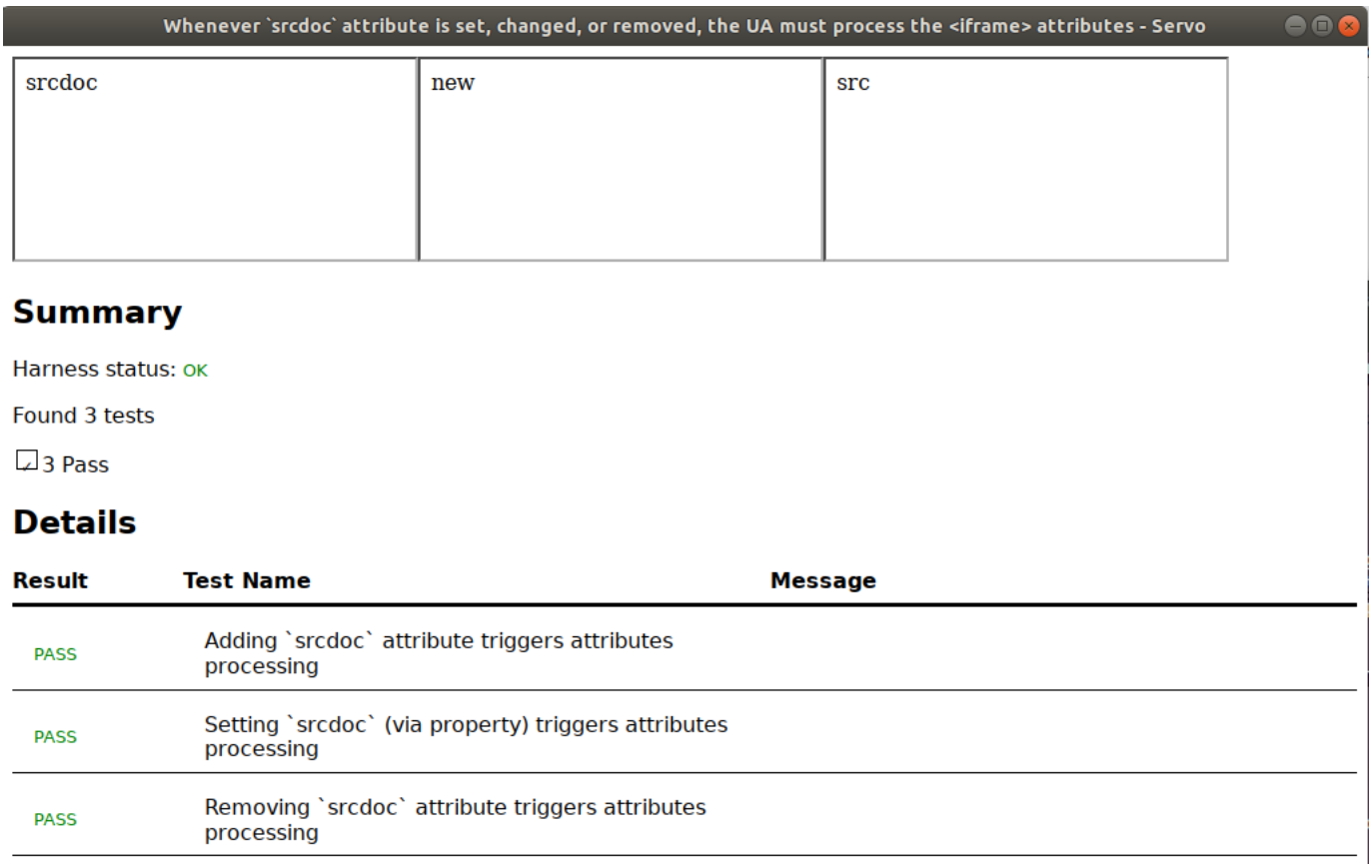
We added a code to fire the **process_the_iframe_attributes** method when **srcdoc** attribute of an iframe element is changed in the file htmliframeelement.rs (https://github.com/servo/servo/compare/master...jaymodi98:iframe-srcdoc#diff-498ce74d806ab54484e768d9237a53b1).

```
610 +            &local_name!("srcdoc") => {
611 +                // https://html.spec.whatwg.org/multipage/#the-iframe-element:the-iframe-element-9
612 +                // "Whenever an iframe element with a non-null nested browsing context has its
613 +                // srcdoc attribute set, changed, or removed, the user agent must process the
614 +                // iframe attributes."
615 +                // but we can't check that directly, since the child browsing context
616 +                // may be in a different script thread. Instead, we check to see if the parent
617 +                // is in a document tree and has a browsing context, which is what causes
618 +                // the child browsing context to be created.
619 +
620 +                // trigger the processing of iframe attributes whenever "srcdoc" attribute is set, changed or removed
621 +                if self.upcast::<Node>().is_connected_with_browsing_context() {
622 +                    debug!("iframe srcdoc modified while in browsing context.");
623 +                    self.process_the_iframe_attributes(ProcessingMode::NotFirstTime);
624 +                }
625 +            },
```

Test Plan

To test if the engine is able to process iframe tag with srcdoc with the command, run: `./mach test-wpt tests/wpt/web-platform-tests/html/semantics/embedded-content/the-iframe-element/srcdoc_process_attributes.html`.

The result of the test is:



We have successfully completed all the initial steps and the tests pass. Our pull request has been merged into the Servo repo.

Pull Request

Here is the link to our pull request (https://github.com/servo/servo/pull/24576). We have attached the code snippets for the changes made in files in the PR. This issue is now solved and our code has been merged into the **master branch of Servo**.

References

[1] https://servo.org/
[2] https://bocoup.com/blog/third-party-javascript-development-future#iframe-srcdoc
[3] https://www.w3schools.com/tags/tag_iframe.asp
[4] https://html.spec.whatwg.org/multipage/forms.html#dom-form-nameditem
[5] https://en.wikipedia.org/wiki/Servo_(software)
[6] https://en.wikipedia.org/wiki/Rust_(programming_language)
[7] https://github.com/servo/servo/blob/master/README.md#setting-up-your-environment
[8] https://bitsofco.de/what-exactly-is-the-dom/
[9] https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement

[10] https://github.com/servo/servo/wiki/Missing-DOM-features-project

[11] https://github.com/servo/servo/issues/16479
[12] https://github.com/servo/servo/issues/4767
[13] https://github.com/servo/servo/pull/24576
[14] https://github.com/servo/servo/pull/25070