# Web Example Generator Design Doc

~~(Better name pending)~~

A design document for detailing all the components involved in creating automatic web demos. The end goal will be to usurp ICU4X's existing web demo by generating markup from Diplomat bridges.

Currently there's a sample demo live at http://ambiguous.name/diplomat/

## 07/19/2024 Notes:

Current ICU4X demo is slow to load.

Make into separate HTML files, all using the same WASM file?

Note from Manish: We can catch demo_gen by having it catch JS attributes as well.

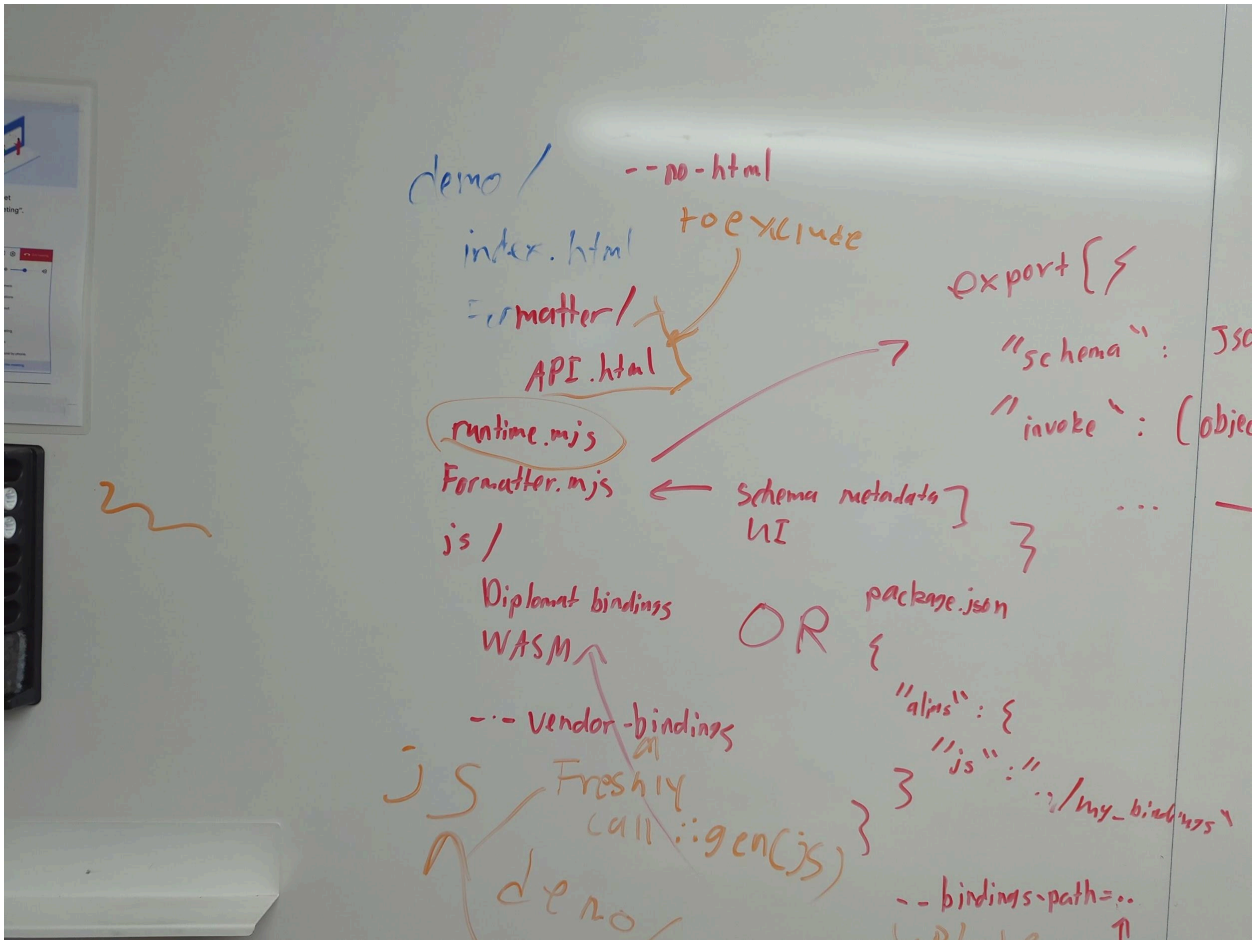
NPM fixes: rendering mode? Here is a path mode?

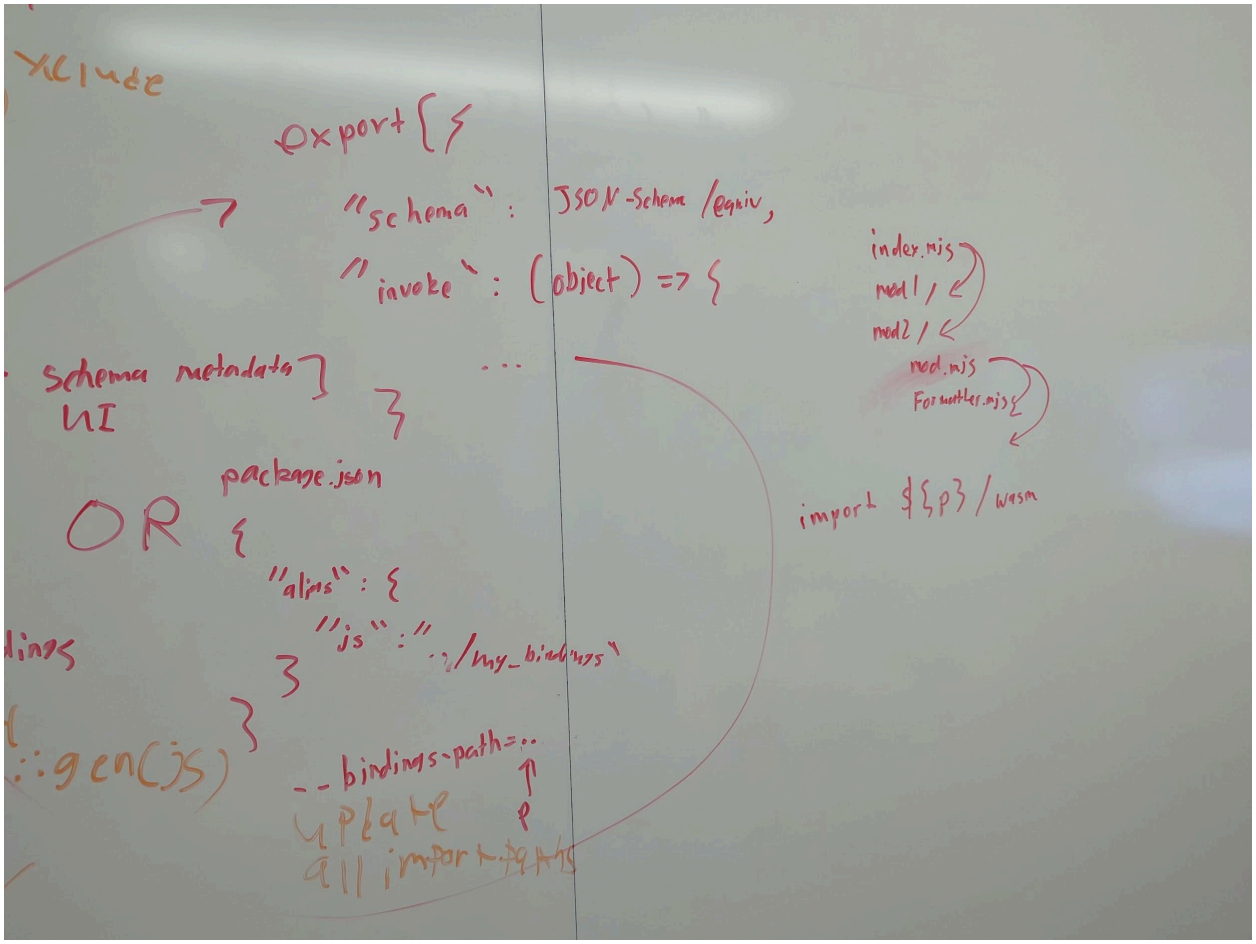
One gotcha - for local package dependencies you have to have an npmrc file.

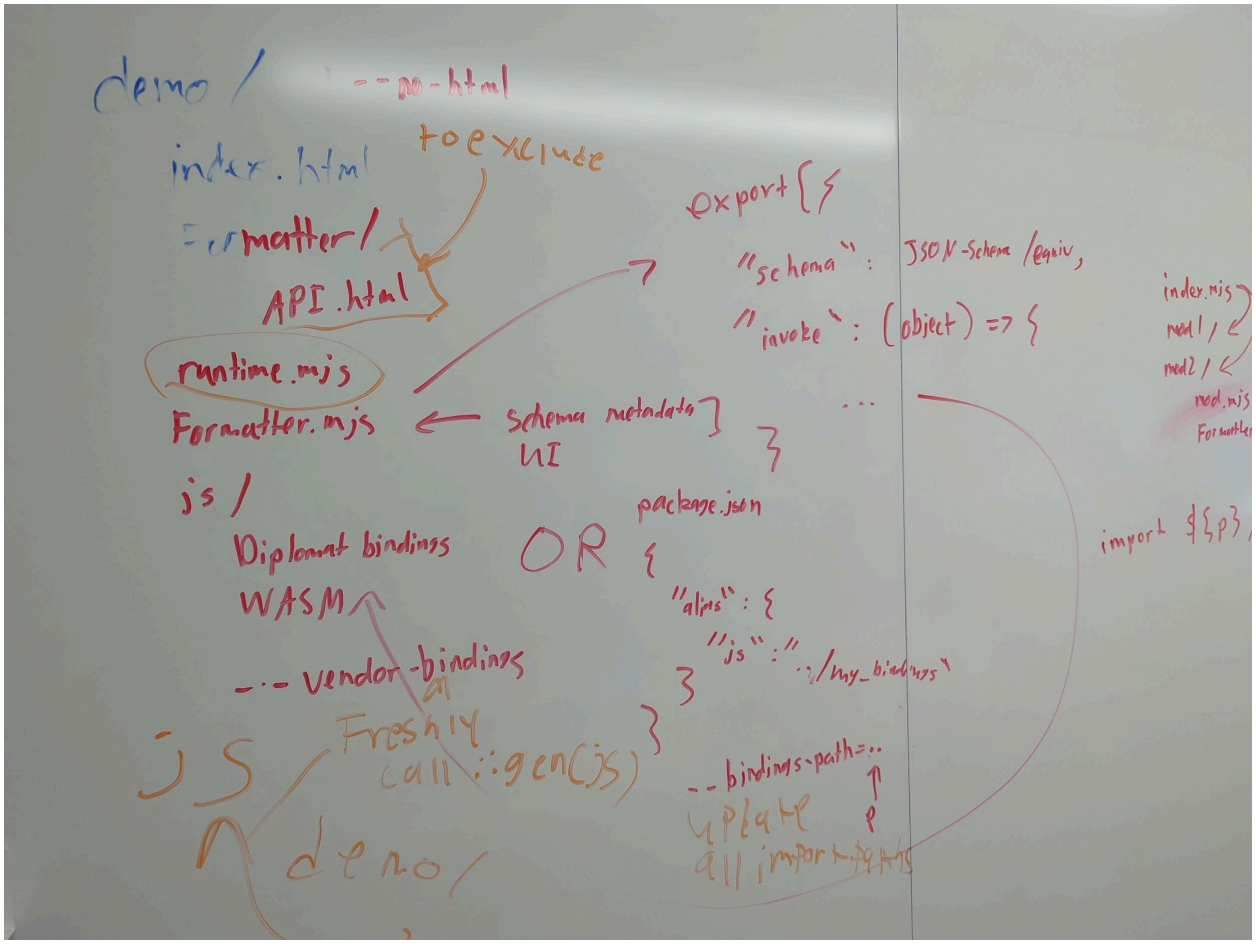
Actionables for file structure:

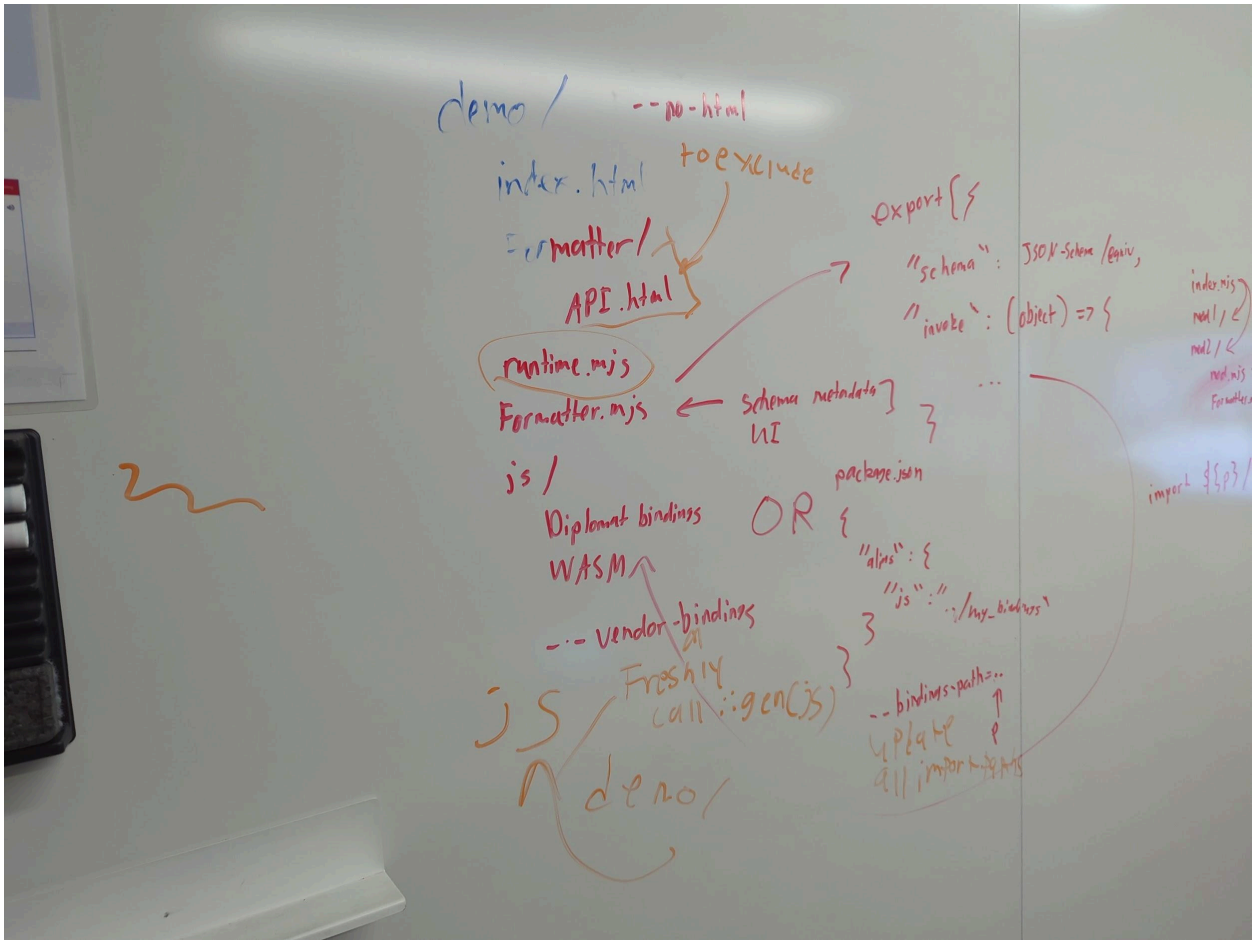
- demo_gen remains its own backend, make sure it recognizes JS as a valid attribute to allow for renames and disables.
- Follow up with Shadaj about potential package setups and demo_gen CLI flags

Add debug assertions in the JS backend? Asserting that a type is an integer?


demo /
--no-html
to exclude
index.html
formatter/
API.html
runtime.mjs
Formatter.mjs
schema metadata
UI
export {
"schema" :
"invoke" : (objec
...
js /
Diplomat bindings
WASM
OR
package.json
{
"alias" : {
"js" : "./my_bindings"
}
-- vendor-bindings
JS
Freshly
call ::gen(JS)
deno
--bindings-path=..

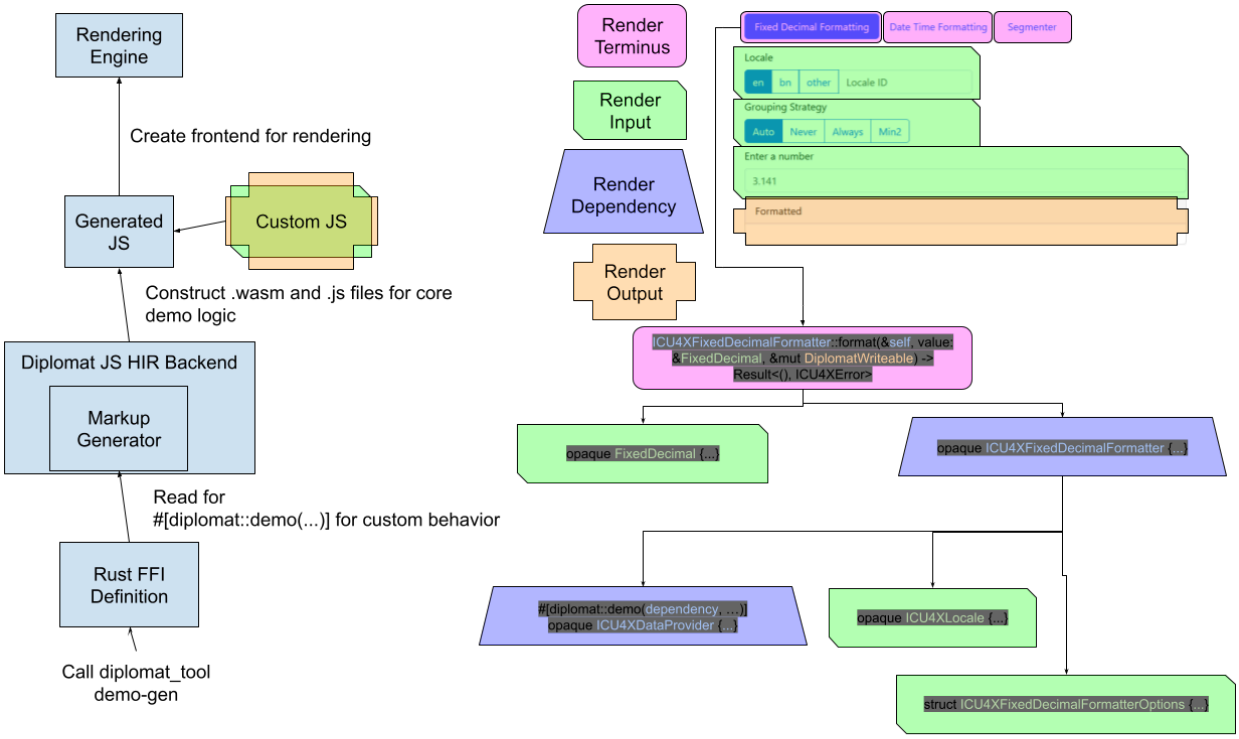

xclude
export {
"schema" : JSON-Schema /equiv,
"invoke" : (object) => {
schema metadata ]
UI
}
...
index.mjs
mod1 /
mod2 /
mod.mjs
Formatter.mjs
package.json
OR
{
"alias" : {
"js" : "./my_bindings"
}
dings
::gen(js)
--bindings-path=..
update
all import paths
import ${p} /wasm


demo/
--no-html
to exclude
index.html
Formatter/
API.html
runtime.mjs
Formatter.mjs
schema metadata
UI
export {
"schema": JSON-Schema /equiv,
"invoke": (object) => {
...
}
index.mjs
mod1/
mod2/
mod.mjs
Formatter
import ${p}
js/
Diplomat bindings
WASM
OR
package.json
{
"alias": {
"js": "./my_bindings"
}
}
--vendor-bindings
Freshly
call ::gen(JS)
--bindings-path=..
update
all imports
JS
deno/




# Rough Sketch

method_lifetimes_map


Rendering Engine
Create frontend for rendering
Generated JS
Custom JS
Construct .wasm and .js files for core demo logic
Diplomat JS HIR Backend
Markup Generator
Read for #[diplomat::demo(...)] for custom behavior
Rust FFI Definition
Call diplomat_tool demo-gen
Render Terminus
Render Input
Render Dependency
Render Output
Fixed Decimal Formatting
Date Time Formatting
Segmenter
Locale
en
bn
other
Locale ID
Grouping Strategy
Auto
Never
Always
Min2
Enter a number
3.141
Formatted
ICU4XFixedDecimalFormatter::format(&self, value: &FixedDecimal, &mut DiplomatWriteable) -> Result<(), ICU4XError>
opaque FixedDecimal {...}
opaque ICU4XFixedDecimalFormatter {...}
#[diplomat::demo(dependency, ...)]
opaque ICU4XDataProvider {...}
opaque ICU4XLocale {...}
struct ICU4XFixedDecimalFormatterOptions {...}

# Invoking Demo-Gen

The current process is:

1. Run diplomat-tool js2 some/folder/here
2. Run diplomat-tool demo-gen some/folder/here
    a. This should be the exact same folder as for JS2.
    b. demo-gen generates a /demo subfolder where all of the demo files go.
    c. You treat it all as one package.

## Discussion Points

Should this be a separate backend, or an arg that you pass to demo-gen?

Separate Backend

- Pros
    - Allows for separation of attributes
        - We might want things that the demo is able to read but JS2 is unable to. Or vice-versa.
        - For instance, using #[diplomat::attr(disable)] only for demo-gen but NOT JS2.
- Cons
    - Slightly harder for the end-user to set up.

Part of JS2 Backend

- Pros
    - Simpler to just set a flag
        - Most people who run JS2 are also going to want to run demo-gen as well.
- Cons
    - There are no really backend-specific flags

# Rust FFI Definition

Note for any of these attributes, we plan to have custom attributes that can configure how they might be interpreted by Diplomat.

## Attributes

There is one #[diplomat::demo(...)] attribute.

Right now, it only supports #[diplomat::demo(default_constructor)] (see Opaques) and #[diplomat::demo(external)], which currently does nothing.

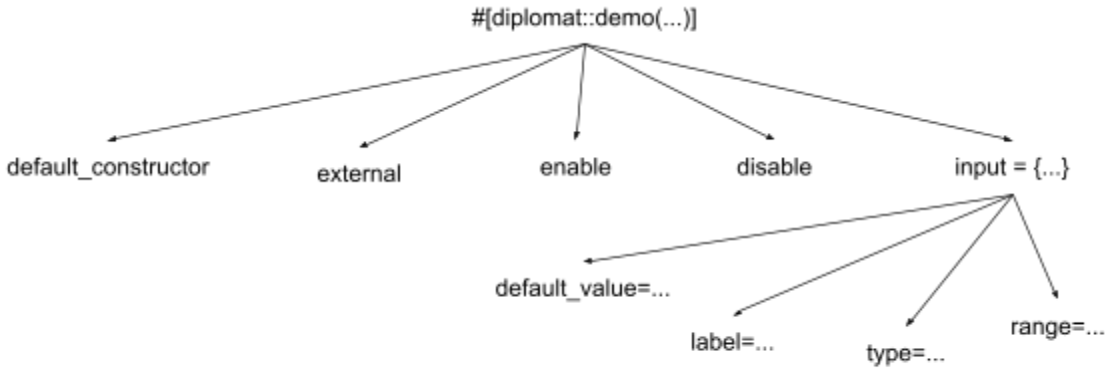
External is meant to be used in cases where a variable is provided as an argument to the render function.

Attribute details need to be discussed further before expanding on any attributes though.

### Discussion Points

1. Attribute setup
    a. Are we comfortable with it?
    b. Right now there are three spots where attributes are implemented for demo-gen:
        i. diplomat/core/src/ast/attrs.rs
            1. This finds #[diplomat::demo()] attributes and creates DemoBackendAttr structs to pass on.
            2. All DemoBackendAttrs are is a syn::Meta struct. These are actually parsed in demo-gen/attrs.rs
        ii. diplomat/core/src/hir/attrs.rs
            1. This takes DemoBackendAttr and just passes it along to demo-gen.
        iii. diplomat/tool/src/demo-gen/attrs.rs
            1. This creates MarkupOutCFGAttr from DemoBackendAttrs.
            2. This is where the actual logic for evaluating these attributes goes.
    c. It feels weird to me to create specific allowance for attributes for one single backend in two files that are meant to be more generic across backends.
2. Attribute Syntax
    a. How do we structure these?

## Attribute Syntax




Sample:

```
func output(
#[diplomat::demo(input(label = “Input Integer Here”, default_value = 0,
range = [0, 1, 0.01]))]
in : i32,
#[diplomat::demo(external)]
some_struct : SomeStruct,
out : &mut DiplomatWrite);
```

Things we want to specify:

- Terminus Information
    - Enable/Disable generation (#[diplomat::demo(enable/disable)])
- Input information (#[diplomat::demo(input(...))])
    - Since these are represented either in structs or function arguments, need special syntax for this.
        - I’m thinking we can take advantage of https://rust-lang.github.io/rfcs/2565-formal-function-parameter-attributes.html, which has been in Rust since 2019.
    - Default value (default_value = “0”)
    - Input Label (label = “Label Here”)
    - Input Type (type = “checkbox”)
        - I’m thinking just allow for all the HTML input types, plus textarea but minus button
    - Possible range (range = [0, 1, 0.01])
        - String
            - Something like ICU4XLocale
            - Autocomplete for lots of possibilities?
        - Number
            - Min, max, step
- Dependency information
    - Opaques
        - Default constructor (#[diplomat::demo(default_constructor)])
        - External/Provided by rendering engine (#[diplomat::demo(external)])

- Structs
    - Optional default constructor (#[diplomat::demo(default_constructor)])

## Render Termini

Any function that returns &mut DiplomatWrite is a render terminus. It represents a function that will be called by the rendering engine after user input has been provided.

The “Fixed Decimal Formatting” tab would be an example of a render terminus.

Render termini encapsulate one demonstration: given a specific function, create a dependency tree of the given parameters and prepare them to be exported. The possible parameters are listed below.

#[diplomat::demo(terminus, …)] would be a configuration macro for non-default behavior.

### Discussion Points

How do we want to evaluate render termini?

We currently automatically search for &mut DiplomatWrite, but we could have other default behaviors.

Some users may want to just set for themselves which functions are termini. Some may want to do it automatically. Which should be default? How do we let them configure this?

## Render Inputs

Render Inputs represent a structure that can be created with user input, that is then passed into the render terminus.

For instance, we require ICU4XFixedDecimal as input to format(). ICU4XFixedDecimal has multiple constructors for strings or integers. Any of these can be called with some user input asking them to input a number.

I imagine we will have to split Render Inputs into two separate definitions: one for the struct, and one for the actual constructor that takes input to make the struct.

#[diplomat::demo(input, …)] would be a configuration macro for non-default behavior.

For creating HTML from HIR types, some approaches are listed below.

## Primitives

Any primitive (or close to primitive) types, like strings, enums, integers, chars, etc., have pretty easy mappings to HTML forms:

- Bool - Checkbox
- Char - Text input with maxlength of 1
- Int/IntSize/Int128 - Number input
- Float - Number input (step of some arbitrary precision?)

## Enums

This would be a simple dropdown.

## Slices

Can be represented through a text field, either for a string slice or primitive slice.

Primitive slices are very dependent on what the developer intends to use them for in any given function call, so I think we could leave the door open for how exactly they intend to pass input into &[u8], &[u16], &[u32], etc.

For a default implementation however, we have some options:

**Approach One**
We could take all text passed into the text field, move it into binary form, and chop it up into the integer type required. Then add extra 0s at the end if there's a remainder of bits.

So if our input is "Hello World", and we wish to map it onto [u16], that would have us merge the binary form:

0100100001100101011011000110110001101111001000000101011101101111011001001101100110001100100

Which we then split up into slices of 16 (marked by _), and append 0s:
0100100001100101 _ 0110110001101100 _ 0110111100100000 _ 0101011101101111 _ 0111001001101100 _ 0110010000000000

- Advantages
    - Maps easily onto [u8]
    - No input validation needed
- Disadvantages
    - Maps less easily for other integer types.

**Approach Two**
We could ask the user to create the buffer for us by asking them to provide a list of comma separated integers. You would format Hello World in [u8] form with:

72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100

- Advantages
    - Better precision when inputting values
    - Works for any integer types.
- Disadvantages
    - Requires either a more complex UI or more input validation

### Structs

We split structs into their component parts until we have enough component parts that we can craft from HTML input.

You can specify a default_constructor optionally, if you prefer that be used instead.

### Opaques

You must label a default constructor with #[diplomat::demo(default_constructor)] or #[diplomat::attr(constructor)].

## Render Outputs

An output is the result of any render terminus function.

To be renderable, we require any output structs to implement some sort of string conversion function. We may wish to add other output types in the future, so leaving the door open for other functions that can produce output is important.

The easiest way to handle string outputs with the existing ICU4X structure is any function that borrows a mutable DiplomatWriteable reference, which we assume that it is writing to.

#[diplomat::demo(output, …)] would be a configuration macro for non-default behavior.

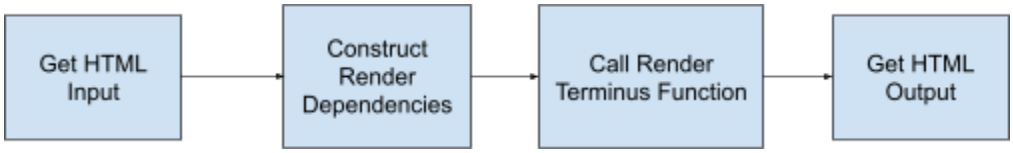
## Render Dependencies

Implicitly defined for any struct that does not have a #[diplomat::demo(...)] macro. These can also be explicitly defined with #[diplomat::demo(dependency, …)] for special options. For instance, being able to define that it is a structure that will be provided to the demo at runtime (like with ICU4XDataProvider).

# Generated JS

Takes every render terminus item and constructs a dependency tree for how best to create a Javascript interface for quickly constructing examples. Then we output a mix of Javascript/WASM for handling the core logic of our demo.

In the final webpage, we want the Javascript/WASM we produce to have the following straightforward flow:




The HTML input/output should ideally be configurable by whoever uses the tool, so the main focus of this section will be in step 2: constructing dependencies.

We construct Javascript files bundled along with the outputs of the JS2 backend, with a function based on each render terminus. The function takes inputs from the rendering engine and returns the writable that the terminus function writes to. That is then passed to the rendering engine for output. This is all compiled into web assembly, to be used in the next step.

The Javascript is simply a stack of .call functions, like so:

```
export function formatWrite(name, grouping_strategy, some_other_config, v) {
        return (function (...args) { return this.formatWrite(...args) }).call(
        ICU4XFixedDecimalFormatter.tryNew.call(
            null,
            ICU4XLocale.new_.call(
                null,
                arguments[0]
            ),
            ICU4XDataProvider.newStatic.call(
                null
            ),
            ((...args) => {
                let out = new ICU4XFixedDecimalFormatterOptions();

                out.groupingStrategy = args[0];

                out.someOtherConfig = args[1];

                return out;
            }).call(
                arguments[1],
                arguments[2]
            )
        ),
        ICU4XFixedDecimal.new_.call(
            null,
            arguments[3]
        )
    );
}
```

## RenderInfo

Along with each bit of generated javascript, we now need a way to present the JS render termini to whatever rendering engine needs them. We also need information like what sort of inputs any given terminus takes.

This is ideally configurable to a developer's specifications. We have a default implementation of frontend construction ourselves (see next section), but making sure that this is doable across frameworks is ideal.

This is where RenderInfo comes in.

So here's an example object, using the FixedDecimal formatting example:

```
export const RenderInfo = {
        termini: [
                {
                        func: ICU4XFixedDecimalFormatterDemo.formatWrite,
                        // For avoiding webpacking minifying issues:
                        funcName: "ICU4XFixedDecimalFormatter.formatWrite",
                        parameters: [

                                {
                                        name: "name",
                                        type: "string"
                                },

                                {
                                        name: "grouping_strategy",
                                        type: "ICU4XFixedDecimalGroupingStra
                                },

                                {
                                        name: "some_other_config",
                                        type: "boolean"
                                },

                                {
                                        name: "v",
                                        type: "number"
                                }

                        ]
                },
```

# Rendering Engine

This is where we provide our own default implementation for rendering HTML.

Currently it's specific to the example/ folder, but the rough idea is to use the Web Components set of features to create customizable templates that function regardless of framework.

The idea is to just be able to customize a provided .html file like so:

```
<template id="terminus">
        <div>
                <slot name="funcName"></slot>
                <slot name="parameters"></slot>
                <button onclick="submit()">Submit</button>
                <slot name="output">Output Shown Here</slot>
        </div>
</template>

<template id="enum">
        <label><slot name="param-name"></slot></label>
        <select oninput="valueChange()">
                <!-- Select doesn't evaluate anything other than option nodes, so to allow dropdow
                <option id="options"></option>
        </select>
</template>

<template id="enum-option">
        <option>
                <slot name="option-text"></slot>
        </option>
</template>

<template id="string">
        <label><slot name="param-name"></slot></label>
        <input type="text" oninput="valueChange()"/>
</template>

<template id="number">
        <label><slot name="param-name"></slot></label>
        <input type="number" oninput="valueChange()"/>
</template>

<template id="boolean">
        <label><slot name="param-name" for="bool"></slot></label>
        <input type="checkbox" id="bool" oninput="valueChange()"/>
</template>
```

Which is then converted into fully rendered HTML like you might see on the live demo page. Be sure to inspect element to see the templates in action.

This is all written in pure JS/HTML, so it allows for maximum compatibility.

# Names

Other thoughts for the project name. Sorted in no particular order:

- Diplomat Web Demo Generator
- Demo HTML Constructor
- Demo Tool
- Demo Markup Generator
- Render Demo
- Diplomat Tutor
- Embassy/Envoy???