Template Instantiation Explainer

justinfagnani@google.com
Last Update: 2019-01-23
Status: Draft

What is Template Instantiation?

Template Instantiation is a proposal* to allow developers to parameterize <templates> to allow interpolation of data and easy updates to DOM cloned from <template> contents.

Template Instantiation is a solution to the problem of creating an updating non-trivial chunks of DOM. The vast majority of client-side rendered pages and template & DOM creation libraries need to create large chunks of DOM derived from app data, and then update them later. The DOM has no well-lit path for doing this. It has lower-level APIs like createElement(), setAttribute(), cloneNode(), and querySelector(), but it is a very difficult problem to use these together to create a fast-booting, fast-updating, small, and easy-to-use, system.

Template Instantiation is also in many ways a completion of the arguably incomplete <template> feature. It is extremely rare that an application needs to copy a completely static, predefined fragment of DOM, and not make updates to it after the fact. <template> is only minimally useful because it doesn't offer any new ability to actually parameterize or update DOM - the critical feature of template systems.

* The Template Instantiation proposal was initially from Apple, but Google was concurrently working on a near-identical proposal.

Overview

Template Instantiation is actually a relatively simple idea. The key innovation is to allow <template>s to contain a number of "Template Parts" that mark dynamic sections of the template, and then to add an API to clone both the template and the Parts together such that the cloned Parts point to the correct cloned nodes.

Currently the platform has no way to make a reference to a node inside a fragment, then point that reference into a different, identically-shaped fragment. Template Parts add that capability. This capability saves userland template systems from the work of walking cloned fragments to find the nodes referenced in templates.

Additionally, the Template Instantiation proposal includes the idea of adding syntax for Parts to save template libraries from walking the template tree to add Parts programmatically. This

should improve boot time of client-side rendered pages, and rehydration time of server-side rendered pages. This syntax is both intended to be used directly in some cases, and as a transformation/compile target for existing and new template syntaxes, including JavaScript-based systems like JSX, lit-html, etc.

So the proposal is divided into two levels:

- Level 1: Programmatic API for attaching Parts, and creating and updating TemplateInstances
- Level 2: A syntax for describing Parts
- Level 3+: Ideas like a built-in TemplateProcess that allows simple expressions

Note: We will use the syntax occasionally in this document to help illustrate the location of Parts within markup, even if the example is not actually using the syntax layer. It's simply a convenient way to visually represent a location in DOM.

The Template Lifecycle

Definition

Parts

In the definition phase, a template element is annotated with markers that denote the dynamic parts of a template. These are called simply *Parts*.

There are two main types of Parts:

- NodeParts describe a location within the text content of a template. ie, `<div>{{}}</div>`
- AttributeParts describe a location associated with an attribute. ie, `<p class={{}}></p>`

Parts can be *attached* to a template.

NodeParts are attached, somewhat like a Range, to a start and end location. Unlike a Range a Part must start and end with entire Nodes (no offsets) and the start and end Nodes must be siblings. This allows a NodePart to set and replace the Nodes contained within its extent without changing the structure of the surrounding DOM.

AttributeParts are attached to a node and an attribute name. AttributeParts support controlling part of an attribute value, so they also support being attached as a group, including static strings between grouped parts.

When a Part is attached to a template it is also added to a private list of Parts owned by the <template>.

The attach API has a few possible variations. The current idea is that NodePart and AttributePart will have static methods:

```
// Fully control an Element
NodePart.attach(element: Element): NodePart;

// Control a range between two nodes
NodePart.attach(startNode: Node, endNode: Node): NodePart;

// Fully control an attribute
AttributePart.attach(element: Element, name: string): AttributePart

// Control one or more ranges of an attribute value
AttributePart.attach(element: Element, name: strings: string[]):
AttributePart[]
```

TemplateProcessors

At definition time it's possible to assign a TemplateProcessor to a template

```
template.processor = new MyTemplateProcessor();
```

A TemplateProcessor can customize the work done in each lifecycle phase.

Instantiation

Once a template has been annotated with Parts, the template and Parts can be cloned together, along with data to be interpolated in one step:

```
const instance = template.createInstance(data);
```

createInstance() returns a TemplateInstance, which contains the cloned DocumentFragment and a list of Parts for setting their values.

By default (with no user-provided TemplateProcessor), createInstance() delegates to the update phase, which handles interpolating data.

Once a TemplateInstance is created it can be attached to the document:

```
container.append(instance.content);
```

Update

The update phase is when dynamic data is merged with the static DOM.

Updates are done in two phases: set-value and commit. The set-value phase updates the current value of Parts, but does not write the value to DOM. The commit phase writes the values to DOM. This is done for several reasons:

- We would like updates to be atomic, especially attributes with multiple Parts
- For performance, we want to defer side-effects of DOM mutation like Custom Element reactions, Mutation Events, etc.

Ideally the set-value/commit separation propagate through nested templates such that an entire DOM tree created from nested templates can defer side-effects all the way down the tree.

By default (with no user-provided TemplateProcessor) data is assigned to Parts by index. Sata is assumed to be an array of the same length as the Parts array and each item is passed to `Part#setValue()`.

Nesting Templates

Note: Very drafty below…

Templates will need to be nested to allow composition, control flow, etc. The simplest way to do this is to allow some form of "template" as a value that can be passed to a NodePart.

The question becomes: what exactly is the type of this value? It needs to combine both the template to be rendered and the data to render, and needs to allow for in-place updates.

TemplateInstance has a template and data and is updatable, so it fits the requirements. The workflow however needs to be such that on the first render a new TemplateInstance is created, and on updates the existing TemplateInstance is simply updated. This conditional behavior must be implementable at the library-level, via a TemplateProcessor.

Note: lit-html has an additional and convenient type used for nesting called TemplateResult which is simply a reference to a template and the current data to render. It's not clear this should be part of the native implementation.

In order to enable the commit phase down through nested TemplateInstances, we'll likely need some downward reference from NodePart to a "updateable/committable" object, which TemplateInstance will implement.

Interfaces

```
class HTMLTemplateElement {
  processor?: TemplateProcessor;
  createInstance(data): TemplateInstance;
}

class TemplateInstance {
  template: HTMLTemplateElement;
  content: DocumentFragment;
  parts: Part[];
  update(data);
}

interface TemplateProcessor {
  prepare(template: HTMLTemplateElement, partLocations: PartLocationp[]):
    PartLocation[];

  create(template: HTMLTemplateElement, partLocations: PartLocationp[]):
    TemplateInstance;

  update(instance: TemplateInstance, data);
}

class Part {
  node: Node;
  setValue(value: any);
  currentValue: any;
  commit();
}

class NodePart extends Part {
  startNode: Node;
  endNode: Node;
}

class AttributePart extends Part {
  name: string;
  committer: AttributePartCommitter;
}
```

```
class AttributePartCommitter {
  parts: AttributePart[];
  commit();
}

interface PartLocation {
  type: 'node' | 'attribute';
}

interface NodePartLocation {
  type: 'node';
  parentNode: Node;
  startNode: Node;
  endNode: Node;
}

interface AttributePartLocation {
  type: 'attribute';
  node: Element;
  name: string;
  offset?: number; // open question on how to represent this
}
```

Examples

First, given a template, we want to mark certain parts of it as dynamic:

```
const template = document.createElement('template');

// How a library marks dynamic parts in template DOM is up to the library
template.innerHTML = `
  <div id="foo"></div>
`;
template.processor = {
  prepare(templateContent, partLocations) {
    // Until Level 2, partLocations will be null/empty, after it may
    // contain Parts parsed from syntax
    const dynamicNode = templateContent.querySelector('#foo');
    dynamicNode.id = null;
    return [NodePart.attach(dynamicNode)];
  }
};
```

Then we can create instances:

```
const instance = template.createInstance(['bar']);
document.body.append(instance.content);
```

And the document will contain:

```
<div>bar</div>
```

We can update data:

```
instance.update(['qux']);
```

And the document will contain:

```
<div>qux</div>
```

Security Considerations

TemplateInstiation should naturally be robust against XSS because values will be written into DOM as escaped textContent, not innerHTML.

Work needs to be done for sanitization of attributes, <style>, and <script> content. lit-html is undergoing security review and security API additions. This work can be repurposed to help possibly make Template Instantiation fully XSS proof by default.

Applicability to Existing Template Libraries

Most existing template systems should be able to benefit from these APIs. From a quick analysis, at the very least Mustache/Handlebars, Angular, Soy, Vue, Polymer, lit-html, and many others should be implementable on top of them. Deeper analysis and prototyping is needed.

Alternatives

TODO, but discuss at least "native VDOM" and DOMChangeList

Links

Some older revisions of the proposal:

- Original Apple Proposal

- Template Instantiation Level 1
- Layering and additional use cases (GitHub issue)