Ember Template Imports

My argument for `<template>`: a series in 5 parts.

Chris Krycho

Contents

Introduction

The Ember and Glimmer community is currently experimenting with designs for components being available in the same file as supporting JavaScript—sometimes described as "single-file components" (or SFCs). There are some working implementations in the ember-template-imports repository, and Ember's community and leadership has already committed to making *some* move in this space the strict mode templates RFC.

It's important to say before I jump in: these are *my* opinions. They're *not* official LinkedIn positions, and in fact I have a number of colleagues who disagree with me about some of these things! I'm writing this series to persuade any and all members of the Ember community, including other people at LinkedIn.

While each of these has its own upsides and downsides, I believe `<template>` is far and away the best choice, because of its wins for teaching and understanding, scaling, and testing. In this series, I will do my best to present an even-handed analysis that shows how and why I came to that conclusion over the last few years of thinking about it.

Part 1 – The Formats

Introducing the series and walking through the formats.

The Ember and Glimmer community is currently experimenting with designs for components being available in the same file as supporting JavaScript—sometimes described as "single-file components" (or SFCs). There are some working implementations in the ember-template-imports repository, and Ember's community and leadership has already committed to making *some* move in this space the strict mode templates RFC.

While each of these has its own upsides and downsides, I believe `<template>` is far and away the best choice, because of its wins for teaching and understanding, scaling, and testing. In this series, I will do my best to present an even-handed analysis that shows how and why I came to that conclusion over the last few years of thinking about it.

In this first post, I will introduce each of the options and give a high-level overview of what I take the tradeoffs in the design space to be. In the following posts, I will cover **Teaching and Understanding**, **Tooling**, and **Testing**. In a final post, I will summarize the tradeoffs once more.

It's important to say before I jump in: these are *my opinions*. They're *not* official LinkedIn positions, and in fact I have a number of colleagues who disagree with me about some of these things! I'm writing this series to persuade any and all members of the Ember community, including other people at LinkedIn.

The options

There are two currently-implemented proposals, and two other major designs which members of the community have advocated for:

- `<template>` tags with a custom file extension (currently `.gjs` and `.gts`)
- template literals using an `hbs` literal
- something like Svelte's and Vue's SFC format
- an imports-only extension of the current format

A few things which are *not* under discussion here:

- *whether* to adopt a format which supports "strict mode"—the RFC has already been approved
- the target compilation format: these all assume we are targeting the Glimmer VM and the existing, standard output formats for components on that VM
- whether we should solve some of these problems in other ways (e.g. having other syntax for object literals or the `and` helper etc.)

Finally, for *this* post I will not be offering any commentary on the tradeoffs in the space. Instead, I will leave the commentary to future posts, and here I will keep my commentary to explaining how each approach works.

Definitions

Throughout this discussion, I will use the following terms:

template-only components

Components which have no backing class and therefore no state of their own. Note that a "template-only" component may still be highly dynamic with the use of helpers, modifiers, and possible future extensions to the framework programming model such as Resources and Effects.

strict mode

The "strict" resolution rules for values in templates defined in Ember RFC #0496.

I will also use the same examples throughout:

- a template-only `Greeting` component
- a stateful component for setting a user's name
- a component which uses built-in helpers and modifiers
- a component which assembles all of these pieces

In these examples, I also assume the following:

- the ember-truth-helpers addon
- a `@glimmer/modifier` package, which supplies core functionality like the `on` modifier, and which I assume for the purposes of *this* post also supplies a `modifier` function to define new modifiers[1]

I am also not using the `@action` decorator: in an Ember Octane app, it doesn't do anything other than bind the `this` context of the class for the method to which it is applied. Leaving it out makes the code a bit briefer, and method-binding/action-handling is not the focus of this series.[2]

The *status quo*

Ember and Glimmer apps *work* today, so it's important to see what they do well and they do *less* well. What's more, whenever considering a change to a programming language or an API, we should have a strong bias *away* from change, and especially away from change for its own sake. Every change is a cost to existing users who have to migrate their code. Even when the change can be made mostly or entirely with automation, it still takes time—to build the automation, to *test* the automation, and ultimately to run the automation.

However, it is worth remembering that we have already decided as a community to adopt features in strict mode templates which the *status quo* does not support!

[1] In practice, I expect we will largely be authoring standalone functions courtesy of the open RFCs to that end!

[2] I'm also on record that I think "action" is a terrible name for what we're doing with that decorator anyway—and there's a high likelihood that the syntax there will have to change if the current decorators proposal is accepted.

Template-only

Here we have a simple component which just "greets" the user. In today's world, this is a standalone file containing just the content to render:

`greeting.hbs`:

```
<p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
```

Stateful/class-backed

This example introduces a stateful, class-backed component which uses a bound function to update some tracked state—a small form to let user input her name and use it to generate an avatar (from an imaginary avatar generator on `example.com`.

- `set-username.js`

  ```
  import Component from '@glimmer/component';
  import { tracked } from '@glimmer/tracking';

  export default class SetUsername extends Component {
    @tracked name;

    get nameValue() {
      return this.name ?? this.args.name;
    }

    updateName = ({ target: { value } }) => {
      this.name = value;
    }

    saveName = (submitEvent) => {
      submitEvent.preventDefault();
      this.args.onSaveName(this.name);
    };
  }
  ```

- `set-username.hbs`: the form itself, which is using `eq` to disable the button if there is no name set[3]

  ```
  <form {{on "submit" this.saveName}}>
    <label for='name'>Set username:</label>
    <input
      id='name'
      value={{this.nameValue}}
      {{on "input" this.updateName}}
  ```

[3] Pro tip: don't do this. There are better ways of handling accessibility, as described in this CSS Tricks post; I implemented the approach described there in this addon.

```
    />

    <button type='submit' disabled={{eq this.name.length 0}}>
      Generate
    </button>
  </form>
```

Using helpers and modifiers

We also need to see what it's like to work with helpers and modifiers—especially in the case where we only need them for one specific component. In this case, we'll imagine we're using an `iframe` and need to update its URL without affecting browser history (a real-world use case!)

- `replace-location.js`

```
import { modifier } from 'ember-modifier';

export default modifier((el, _ , { with: newUrl }) => {
  el.contentWindow.location.replace(newUrl);
});
```

- usage:

```
<iframe
  title='...'
  {{replace-location with=@src}}
/>
```

All features

Now we can assemble all of these together with a parent component, `GenerateAvatar`. I am including *all* the files here to give a more direct comparison between each of the approaches.

- `generate-avatar.js`:

```
import Component from '@glimmer/component';

export default class GenerateAvatar extends Component {
  @tracked name = "";

  get previewUrl() {
    return `http://www.example.com/avatars/${this.name}`;
  }

  updateName = (newName) => {
    this.name = newName;
```

```
  };
}
```

- generate-avatar.hbs:

```
<Greet @name={{this.name}} />
<SetUsername
  @name={{this.name}}
  @onSaveName={{this.updateName}}
/>

{{#if (gt 0 this.name.length)}}
  <iframe
    title='Preview'
    {{replace-location with=this.previewUrl}}
  >
{{/if}}
```

- replace-location.js

```
import { modifier } from 'ember-modifier';

export default modifier((el, _ , { with: newUrl }) => {
  el.contentWindow.location.replace(newUrl);
});
```

- greeting.hbs:

```
<p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
```

- set-username.js

```
import Component from '@glimmer/component';
import { tracked } from '@glimmer/tracking';

export default class SetUsername extends Component {
  @tracked name;

  get nameValue() {
    return this.name ?? this.args.name;
  }

  updateName = ({ target: { value } }) => {
    this.name = value;
  }

  saveName = (submitEvent) => {
    submitEvent.preventDefault();
    this.args.onSaveName(this.name);
```

```
  };
}
```

- set-username.hbs:

```
<form {{on "submit" this.saveName}}>
  <label for='name'>Set username:</label>
  <input
    id='name'
    value={{this.nameValue}}
    {{on "input" this.updateName}}
  />

  <button type='submit' disabled={{eq this.value.length 0}}>
    Generate
  </button>
</form>
```

`<template>` tags

The second proposed format uses `<template>` in files with a custom file extension (currently the proposal has `.gjs` and `.gts` for Glimmer JavaScript or Glimmer TypeScript files respectively).

A `<template>` tag contains the template content, which will be *compiled* to an appropriate value in the JavaScript context. There are two compilation outputs:

1. A `<template>` in module scope (that is, a `<template>` which is not in a class body) compiles to a template-only component.
   - A `<template>` may be assigned to a binding in the JavaScript module:

     ```
     const Greet = <template>Hello!</template>
     ```

   - These bindings may be exported or not, just as any other in a JavaScript module:

     ```
     export const Greet = <template>Hello!</template>

     const Conditional = hbs`{{#if @val}}, {{@val}}{{/if}}`;

     const Farewell = <template>
       Goodbye<Conditional @val={{@user}} />!
     </template>
     export default Farewell;
     ```

   - A top-level `<template>` with no binding is equivalent to writing `export default <template>...</template>`.

     Accordingly, you cannot have multiple unbound top-level `<template>`, and you cannot have *both* an explicit `export default` and an unbound

top-level `<template>`, because having multiple `export default` statements is not allowed in JS.

2. A `<template>` within a class body compiles to a template attached to the class (and bound to the class prototype, not to class instances).

```
class Example extends Component {
  respond = () => {
    alert("You clicked it!");
  };

  <template>
    <button type='button' {{on "click" this.respond}}>
      CLICK
    </button>
  </template>
}
```

3. These are the only locations a `<template>` can be created. You cannot create and return them from a function, for example.

Template-only

`greeting.js`:

```
<template>
  <p>
    Hello{{#if @name}}, {{@name}}{{/if}}!
  </p>
</template>
```

Stateful/class-backed

```
import Component from '@glimmer/component';
import { tracked } from '@glimmer/tracking';
import { on } from '@glimmer/modifier';
import { eq } from 'ember-truth-helpers';

export default class SetUsername extends Component {
  @tracked name;

  get nameValue() {
    return this.name ?? this.args.name;
  }

  updateName = ({ target: { value } }) => {
    this.name = value;
  }
```

```
  saveName = (submitEvent) => {
    submitEvent.preventDefault();
    this.args.onSaveName(this.name);
  };

  <template>
    <form {{on "submit" this.saveName}}>
      <label for='name'>Set username:</label>
      <input
        id='name'
        value={{this.value}}
        {{on "input" this.updateName}}
      />
      <button type='submit' disabled={{eq this.value.length 0}}>
        Generate
      </button>
    </form>
  </template>
}
```

Using helpers and modifiers

The *definition* of the modifier is identical, but it can appear and be used inline with the files where it is needed. See the next section!

All features

With all these features available, this can be a single file if there is no need to reuse the components.[4]

```
import Component from '@glimmer/component';
import { tracked } from '@glimmer/tracking';
import { on, modifier } from '@glimmer/modifier';
import { eq, gt } from 'ember-truth-helpers';

const Greeting = <template>
  <p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
</template>

const replaceLocation = modifier(
  (el, _ , { with: newUrl }) => {
    el.contentWindow.location.replace(newUrl);
  }
```

[4] And even if there *is*, if we think about them as related: `Greeting` could be a named export, for example.

```
);

class SetUsername extends Component {
  @tracked name = '';

  updateName = ({ target: { value } }) => {
    this.name = value;
  }

  saveName = (submitEvent) => {
    submitEvent.preventDefault();
    this.args.onSaveName(this.name);
  };

  <template>
    <form {{on "submit" this.saveName}}>
      <label for='name'>Set username:</label>
      <input
        id='name'
        value={{this.value}}
        {{on "input" this.updateName}}
      />
      <button type='submit' disabled={{eq this.value.length 0}}>
        Generate
      </button>
    </form>
  </template>
}

export default class GenerateAvatar extends Component {
  @tracked name = "";

  get previewUrl() {
    return `http://www.example.com/avatars/${name}`;
  }

  updateName = (newName) => {
    this.name = newName;
  };

  <template>
    <Greet @name={{this.name}} />
    <SetUsername
      @name={{this.name}}
      @onSaveName={{this.updateName}}
    />
```

```
    {{#if (gt 0 this.name.length)}}
      <iframe
        title='Preview'
        {{replaceLocation with=this.previewUrl}}
      >
    {{/if}}
  </template>
}
```

Tagged template literals with `hbs`

The second proposal, currently implemented in the GlimmerX experiment, is an extension of an existing feature in Ember: the use of a special `hbs` tagged template literals. (This is similar to how templates defined for tests work today.) In the proposed design, `hbs` is imported from `ember-template-imports` as a named import; in a final design it would presumably be imported from `@glimmer/component`:

```
import { hbs } from '@glimmer/component';
```

The rules are very similar to those for the `<template>` proposal:

1. The result of an `hbs` invocation in module scope (not in a class body) is a template-only component, bound to a name in a module. It can be a default export or a named export or not exported at all:
    - An `hbs` invocation may be assigned to a binding in the JavaScript module:

      ```
      import { hbs } from '@glimmer/component';

      const Greet = hbs`Hello!`;
      ```

    - These bindings may be exported or not, just as any other in a JavaScript module:

      ```
      import { hbs } from '@glimmer/component';

      export const Greet = hbs`Hello!`;

      const Conditional = hbs`{{#if @val}}, {{@val}}`;

      const Farewell = hbs`Goodbye<Conditional @val={{@user}} />!`;
      export default Farewell;
      ```

    - Unlike with `<template>`, there is no special case behavior for a single definition, so a default template-only component export would be written like so:

```
import { hbs } from '@glimmer/component';

export default hbs`Hello!`;
```

2. To define the template for a class-backed/stateful component, you assign it to the specially-named (effectively *reserved*) `static` field `template` on the backing class:

```
import Component, { hbs } from '@glimmer/component';

class Example extends Component {
  respond = () => {
    alert("You clicked it!");
  };

  static template = hbs`
    <button type='button' {{on "click" this.respond}}>
      CLICK
    </button>
  `;
}
```

3. These are the only locations the result of `hbs` can be used. You cannot create and return them from a function, for example.

4. `hbs` invocations are compiled out; they are *not* actual tagged template strings, and so cannot use `${...}` syntax for string interpolation.

Template-only

`greeting.js`:

```
import { hbs } from '@glimmer/component';

export default hbs`
  <p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
`;
```

Stateful/class-backed

`set-username.js`:

```
import Component, { hbs } from '@glimmer/component';
import { tracked } from '@glimmer/tracking';
import { on } from '@glimmer/modifier';
import { eq } from 'ember-truth-helpers';

export default class SetUsername extends Component {
  @tracked name;
```

```
  get nameValue() {
    return this.name ?? this.args.name;
  }

  updateName = ({ target: { value } }) => {
    this.name = value;
  }

  saveName = (submitEvent) => {
    submitEvent.preventDefault();
    this.args.onSaveName(this.name);
  };

  static template = hbs`
    <form {{on "submit" this.saveName}}>
      <label for='name'>Set username:</label>
      <input
        id='name'
        value={{this.value}}
        {{on "input" this.updateName}}
      />
      <button type='submit' disabled={{eq this.value.length 0}}>
        Generate
      </button>
    </form>
  `;
}
```

Using helpers and modifiers

As with `<template>`, the *definition* of the modifier is identical, but it can appear and be used inline with the files where it is needed. See the next section!

All features

Again, with all these features available, this can be a single file if there is no need to reuse the components.

```
import Component, { hbs } from '@glimmer/component';
import { tracked } from '@glimmer/tracking';
import { on, modifier } from '@glimmer/modifier';
import { eq, gt } from 'ember-truth-helpers';

const Greeting = hbs`
  <p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
`;
```

```
const replaceLocation = modifier((el, _ , { with: newUrl }) => {
  el.contentWindow.location.replace(newUrl);
});

class SetUsername extends Component {
  @tracked name = '';

  updateName = ({ target: { value } }) => {
    this.name = value;
  }

  saveName = (submitEvent) => {
    submitEvent.preventDefault();
    this.args.onSaveName(this.name);
  };

  static template = hbs`
    <form {{on "submit" this.saveName}}>
      <label for='name'>Set username:</label>
      <input
        id='name'
        value={{this.value}}
        {{on "input" this.updateName}}
      />
      <button type='submit' disabled={{eq this.value.length 0}}>
        Generate
      </button>
    </form>
  `;
}

export default class GenerateAvatar extends Component {
  @tracked name = "";

  get previewUrl() {
    return `http://www.example.com/avatars/${name}`;
  }

  updateName = (newName) => {
    this.name = newName;
  };

  static template = hbs`
    <Greet @name={{this.name}} />
    <SetUsername
```

```
      @name={{this.name}}
      @onSaveName={{this.updateName}}
    />

    {{#if (gt 0 this.name.length)}}
      <iframe
        title='Preview'
        {{replaceLocation with=this.previewUrl}}
      >
    {{/if}}
  `;
}
```

Svelte/Vue-style

For convenience, and following Svelte and Vue's example, I am using a `.glimmer` file extension for the following examples:

Template-only

`greeting.glimmer`:

```
<p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
```

Stateful/class-backed

`set-username.glimmer`:

```
<script>
  import Component from '@glimmer/component';
  import { tracked } from '@glimmer/tracking';

  export default class SetUsername extends Component {
    @tracked name;

    get nameValue() {
      return this.name ?? this.args.name;
    }

    updateName = ({ target: { value } }) => {
      this.name = value;
    }

    saveName = (submitEvent) => {
      submitEvent.preventDefault();
      this.args.onSaveName(this.name);
    };
```

```
  }
</script>

<form {{on "submit" this.saveName}}>
  <label for='name'>Set username:</label>
  <input
    id='name'
    value={{this.value}}
    {{on "input" this.updateName}}
  />
  <button type='submit' disabled={{eq this.value.length 0}}>
    Generate
  </button>
</form>
```

Using helpers and modifiers

Helpers and modifiers in a hypothetical Svelte/Vue-style template could be defined next to the component backing class, so I will demonstrate them in the next section that way. They could of course also be defined in other modules and imported.

All features

Unlike in the `<template>` and `hbs` scenarios, you cannot define multiple components in the same file with this format. Accordingly, here we *must* have three separate files:

- `greet.glimmer`:

  ```
  <p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
  ```

- `set-username.glimmer`:

  ```
  <script>
    import Component from '@glimmer/component';
    import { tracked } from '@glimmer/tracking';
    import { on } from '@glimmer/modifier';
    import { eq } from 'ember-truth-helpers';

    export default class SetUsername extends Component {
      @tracked name = '';

      updateName = ({ target: { value } }) => {
        this.name = value;
      }

      saveName = (submitEvent) => {
  ```

```
      submitEvent.preventDefault();
      this.args.onSaveName(this.name);
    };
  }
</script>

<form {{on "submit" this.saveName}}>
  <label for='name'>Set username:</label>
  <input
    id='name'
    value={{this.value}}
    {{on "input" this.updateName}}
  />
  <button type='submit' disabled={{eq this.value.length 0}}>
    Generate
  </button>
</form>
```

- generate-avatar.glimmer:

```
<script>
  import Component from '@glimmer/component';
  import { tracked } from '@glimmer/tracking';
  import { modifier } from '@glimmer/modifier';
  import { gt } from 'ember-truth-helpers';

  import Greet from './greet';
  import SetUsername from './set-username';

  const replaceLocation = modifier(
    (el, _ , { with: newUrl }) => {
      el.contentWindow.location.replace(newUrl);
    }
  );

  export default class GenerateAvatar extends Component {
    @tracked name = "";

    get previewUrl() {
      return `http://www.example.com/avatars/${name}`;
    }

    updateName = (newName) => {
      this.name = newName;
    };
  }
</script>
```

```
<Greet @name={{this.name}} />
<SetUsername
  @name={{this.name}}
  @onSaveName={{this.updateName}}
/>

{{#if (gt 0 this.name.length)}}
  <iframe
    title='Preview'
    {{replaceLocation with=this.previewUrl}}
  >
{{/if}}
```

Imports-only

The final option under consideration is a very small extension of today's baseline, which adds support for "front-matter" to templates, to allow them to specify imports explicitly. In this case, things work *exactly* as they do today, but all non-keyword functionality must be explicitly imported, including other components invoked within the component. For example, to define a template-only component which uses the `{{on}}` modifier, you would do this:

```
---
import { on } from '@glimmer/modifier';
---

<div {{on "mousenter" @isHovered}}></div>
```

Template-only

Template-only components with no imports look exactly as they do today:

`greeting.hbs`:

```
<p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
```

Stateful/class-backed

Class-backed/stateful components look much the same on the class definition side as they do today. The big difference is that the template side must separately define its imports as well.

- `set-username.js`

  ```
  import Component from '@glimmer/component';
  import { tracked } from '@glimmer/tracking';

  export default class SetUsername extends Component {
  ```

```
  @tracked name = '';

  updateName = ({ target: { value } }) => {
    this.name = value;
  }

  saveName = (submitEvent) => {
    submitEvent.preventDefault();
    this.args.onSaveName(this.name);
  };
}
```

- set-username.hbs

```
---
import { on } from '@glimmer/modifier';
import { eq } from 'ember-truth-helpers';
---

<form {{on "submit" this.saveName}}>
  <label for='name'>Set username:</label>
  <input
    id='name'
    value={{this.value}}
    {{on "input" this.updateName}}
  />
  <button type='submit' disabled={{eq this.value.length 0}}>
    Generate
  </button>
</form>
```

Using helpers and modifiers

In the imports-only world, helpers and modifiers must be defined in their own file and imported. Accordingly, the definition is as it is with today's *status quo*.

replace-location.js

```
import { modifier } from 'ember-modifier';

export default modifier((el, _ , { with: newUrl }) => {
  el.contentWindow.location.replace(newUrl);
});
```

All features

As with the rest of this section, this represents a fairly minimal change over the *status quo*. The only difference is the requirement for templates to explicitly

define their imports.

- `generate-avatar.js`:

```
import Component from '@glimmer/component';

export default class GenerateAvatar extends Component {
  @tracked name = "";

  get previewUrl() {
    return `http://www.example.com/avatars/${name}`;
  }

  updateName = (newName) => {
    this.name = newName;
  };
}
```

- `generate-avatar.hbs`:

```
---
import { gt } from 'ember-truth-helpers';
import Greet from './greet';
import SetUsername from './set-username';
import replaceLocation from '../modifiers/replace-location';
---

<Greet @name={{this.name}} />
<SetUsername
  @name={{this.name}}
  @onSaveName={{this.updateName}}
/>

{{#if (gt 0 this.name.length)}}
  <iframe
    title='Preview'
    {{replaceLocation with=this.previewUrl}}
  >
{{/if}}
```

- `replace-location.js`

```
import { modifier } from 'ember-modifier';

export default modifier((el, _ , { with: newUrl }) => {
  el.contentWindow.location.replace(newUrl);
});
```

- `greeting.hbs`:

```
<p>Hello{{#if @name}}, {{@name}}{{/if}}!</p>
```

- `set-username.js`

```
import Component from '@glimmer/component';
import { tracked } from '@glimmer/tracking';

export default class SetUsername extends Component {
  @tracked name;

  get nameValue() {
    return this.name ?? this.args.name;
  }

  updateName = ({ target: { value } }) => {
    this.name = value;
  }

  saveName = (submitEvent) => {
    submitEvent.preventDefault();
    this.args.onSaveName(this.name);
  };
}
```

- `set-username.hbs`: the form itself, which is using `eq` to disable the button if there is no name set[^3]

```
---
import { on } from '@glimmer/modifier';
import { eq } from 'ember-truth-helpers';
---

<form {{on "submit" this.saveName}}>
  <label for='name'>Set username:</label>
  <input
    id='name'
    value={{this.nameValue}}
    {{on "input" this.updateName}}
  />

  <button type='submit' disabled={{eq this.value.length 0}}>
    Generate
  </button>
</form>
```

Looking forward

Hopefully this gives you a good sense of the overall *feel* of the moves currently under consideration in design the space. You may have some opinions already about which of these you like best—certainly I did when I first started thinking about this. Even so, I hope you'll also consider the tradeoffs here with an open mind as I present them in the parts ahead!

Part 2 – Teaching and Understanding

Which template imports design has the biggest set of wins for teaching and understanding components?

In this second part of my series on Ember Template Imports, I am tackling the subject of **Teaching and Understanding**. The first part introduced the series and the options on the table. In future posts, I will look at each option in terms of **Tooling** and in terms of **Testing**, before wrapping up with a conclusion.

As a reminder, the four formats under discussion are:

- `<template>` tags with a custom file extension (currently `.gjs` and `.gts`)
- template literals using an `hbs` literal
- something like Svelte's and Vue's SFC format
- an imports-only extension of the current format

I will also be assuming the Definitions. However, unlike in the first post, where I simply aimed to get all the options on the table clearly, I will *not* be repeating the same examples with each different format. Instead, this post is structured as an *argument* in favor of my preferred format: `<template>` tags.

Prefatory comments

One of the most important aspects of a decision about the design of a language or an API is how it impacts developers' ability to learn it and to develop a correct mental model for it. The design of template imports sits right at the boundary between programming language and API design, because it is a way of expressing the relationship between two programming languages: JavaScript and the Glimmer templating language.

It's worth remembering, too, that the relationship between host language and some sort of templating language is a fundamental decision in the design space for application programming of all sorts which render HTML. This is not a concern only of client-side-rendered applications or SPAs: it applies equally to Rails and ERB or to PHP or to C# apps with Razor templates.

On motivation

Historically, Glimmer templates have been *almost* completely separated from JavaScript, with very specific and explicit bridges: component backing classes, helpers, and modifiers. That narrow boundary has helped Ember keep a strong "separation of concerns" between HTML and JS, and this has been a real win—both for being able to optimize the rendering layer and for being able to statically analyze and therefore *lint* the rendering layer.

At the same time, every Glimmer template have had implicit access to *every single* component, helper, or modifier throughout an app and its addons, which has made it difficult to perform many *other* kinds of optimizations and analyses. Full

dead code elimination, for example, has been effectively intractable; and features like go-to-definition or refactoring have been *much* harder to support than they would be with imports. Supporting developer discoverability—via docs, autocomplete, etc.—has likewise been challenging.

Adopting strict mode inherently solves this second problem: it requires that components, helpers, and modifiers[5] be imported and available in lexical scope—specifically, with *JavaScript*'s lexical scoping rules.

lexical scoping Access to any given binding is defined by rules defined in terms of the *definition* of a given block (including functions, class methods, if statements, etc.). Things in the same block or a parent scope are available; things in child or sibling blocks are not.

This is in contrast to dynamic scoping, where the *invocation* of a given function defines what variables it has access to.

This is great! It enables tools like Glint or ELS to work much more easily, and to take advantage of existing tooling which understands JavaScript's semantics. But precisely *because* this works by requiring *some* values used in templates to be available in lexical scope, it raises the question: *Why shouldn't templates have access to* ***other*** *values in JavaScript lexical scope? Why should this be limited to imports? Wouldn't it be useful for other things, too?*

Each of the `<template>` tags, `hbs` tagged literals, and Svelte/Vue-style SFC designs answers that templates *should* have access to other kinds of values in scope in some kind of JavaScript context. The imports-only/"front matter" does what the name says: it sticks to imports only.

On design principles

I also take it as a given for this design that we should embrace a key idea in both teaching specifically and API design more generally: progressive disclosure of complexity.

progressive disclosure of complexity a design principle for UIs, including APIs, which says we should only require a user of the API to do or even understand the *minimum* amount required to accomplish the task at hand.

As suggested by the definition, there are two similar but not identical ways that that progressive disclosure of complexity shows up here.

1. How many concepts does accomplishing a task require the user to *understand*?
2. How many concepts does accomplishing a task require the user to *use*?

In both cases, it's generally preferable to minimize the number of concepts in play, and in particular to design our APIs to avoid introducing concepts which

[5] and possible future features like Resources and Effects

aren't directly required for the task at hand—both because that introduces learning overhead and because it introduces more places a user can get confused or make mistakes when using the API.

At the same time, we have to hold that principle in tension with another constraint: trying to uphold the principle of least surprise.

principle of least surprise So far as possible, a design should behave in the way that users will generally *expect* it to behave. An interface which matches other interfaces should not behave wildly differently from them.

That is: if there are existing (especially if there are well-established) reasons for a user to expect a given API to have certain semantics or meaning, our design should usually follow that. This makes it easier to learn and to remember, and it also helps prevent mistakes even for experienced developers: because they don't have to remember "Oh, right, it's different for this case!"

So: which of the template import designs is the best move for us in terms of teaching and learning, especially when keeping in mind the idea of progressive disclosure of complexity and the principle of least astonishment?

In the discussion which follows, I'm using `.js` in every import, and I'll be going back to update part 1 in the post to do this as well. This might be surprising, but there's a reason for it: this matches the ES Modules spec as implemented in Node 12+, and is therefore what ecosystem tooling (including the TypeScript Language Server) expects. We may choose as a community to layer on custom tooling to reinterpret other extensions to resolve and therefore be less surprising! However, that carries some risks as well as additional implementation effort, and this also serves to highlight some of the tradeoffs in this space nicely.

Analysis

Imports-only

Let's start with a quick evaluation of the imports-only/"front matter" design, as the odd one out. Out of the box, this has incredibly low overhead. Our simplest possible `<Greeting>` component is "just HTML":

```
<p>Hello, {{@name}}!</p>
```

Introducing imports, say into a `UserOverview` component, only extends this a little bit:

```
---
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';
---

<div>
```

```
  <Greeting @name="Chris" />
  <WeatherSummary />
</div>
```

Per my comments above, this uses `.js` for the import, because even standalone files are ultimately compiled to `.js`. We could certainly build tooling which enables us to use `./greeting.hbs` here. However, notice that in that case, as soon as we introduce a class-backed component, users would also need to update all of their `import` statements which reference it, changing them to `.js` at that time, because importing just the `.hbs` file would become invalid. This in turn would produce the very awkward situation where importing a `.hbs` file is valid *sometimes*—requiring extra explanation. Fundamentally, Glimmer templates aren't pure HTML: they always compile to JS.

You can see how this would look if we assume that the `WeatherSummary` component is class-backed (because it has some internal state from checking the weather regularly), and `Greeting` is still template-only:

```
import Component from '@glimmer/component';

export default class WeatherSummary extends Component {
  @tracked currentTemp;

  interval;

  getWeather = () => {
    this.currentTemp = // something
  }

  constructor(owner, args) {
    super(owner, args);
    this.interval = setInterval(this.getWeather, 10000);
  }

  willDestroy() {
    super.willDestroy();
    clearInterval(this.interval);
  }
}
```

```
---
import { gt, lt } from '@glimmer/helper';
---

<p>
  The current temperature is {{this.currentTemp}}!
  {{#if (lt 50 this.currentTemp)}}
    Brr! 
```

```
  {{else if (gt 80 this.currentTemp)}}
    Yikes! 
  {{/if}}
</p>
```

Then we would end up with this surprising usage:

```
---
import Greeting from './greeting.hbs';
import WeatherSummary from './weather-summary.js';
---

<div>
  <Greeting @name="Chris" />
  <WeatherSummary />
</div>
```

This is just weird!

Additionally, this design indicates to existing JavaScript developers that the space between the `---` has JavaScript semantics… because it *does* have JavaScript semantics, but only for imports! It will naturally lead people to ask whether they can do *other* JavaScript things in that space, like defining a helper with a function:

```
---
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';
import Celebration from './celebration.js'

function isBirthday(dateOfBirth) {
  const now = new Date();
  return (
    dateOfBirth.getDate() === now.getDate() &&
    dateOfBirth.getMonth() === now.getMonth()
  );
}
---

<div>
  <Greeting @name={{@user.name}} />
  {{#if (isBirthday @user.dateOfBirth)}}
    <Celebration type='birthday' />
  {{/if}}
  <WeatherSummary />
</div>
```

This seems quite natural and indeed desirable: it keeps the separation between templates and JavaScript which many developers highly value, while still making it easy to provide local functionality. This is exactly the intuition which leads to

Svelte/Vue-style SFCs, which are just a strict superset of the design, and which furthermore are much easier to get at least nice syntax highlighting for (as we'll see below). ***But this isn't allowed in this design.*** Even if there's no reason for the function to exist *other* than this particular component, we still have to put it in its own file and import it:

```
export default function isBirthday(dateOfBirth) {
  const now = new Date();
  return (
    dateOfBirth.getDate() === now.getDate() &&
    dateOfBirth.getMonth() === now.getMonth()
  );
}
```

```
---
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';
import isBirthday from './is-birthday.js'
---

<div>
  <Greeting @name={{@user.name}} />
  {{#if (isBirthday @user.dateOfBirth)}}
    <Celebration type='birthday' />
  {{/if}}
  <WeatherSummary />
</div>
```

We *are* using JavaScript semantics, but only for `import` statements. This design doesn't allow developers to lean on their intuitions about JavaScript, and indeed it requires us to explicitly teach *more* special semantics for templates.[6] The primary upside here is for *existing* Ember developers, for whom this is the smallest change compared to the existing design. However, even there, it has a quirky wrinkle with class-backed components: the set of things in scope for a component template becomes:

- all values available on the backing class
- whatever values are explicitly imported in the template

That first point means that there *is* a way to get a standalone helper or modifier in scope without exposing it to other modules. You just have to write it in the JavaScript file for a backing class and then attach it to the class somehow:

```
import Component from '@glimmer/component';
```

[6] This was actually my objection when Sam Selikoff first proposed using JS import semantics to deal with the problems of the design. Sam was absolutely right about use of JS imports, but I still think I was right about the problems of the --- design!

```
function isBirthday(dateOfBirth) {
  const now = new Date();
  return (
    dateOfBirth.getDate() === now.getDate() &&
    dateOfBirth.getMonth() === now.getMonth()
  );
}

export default class Summary extends Component {
  isBirthday = isBirthday;
}
```

Then we can use the helper in the template, as before:

```
---
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';
import Celebration from './celebration.js'
---

<div>
  <Greeting @name={{@user.name}} />
  {{#if (this.isBirthday @user.dateOfBirth)}}
    <Celebration type='birthday' />
  {{/if}}
  <WeatherSummary />
</div>
```

In this case, it would also be fine for `isBirthday` to be a getter, but even then it's introducing a backing class when there's no need for a class at all *except* to get around the fact that the design here doesn't give any other way to provide a simple helper.

Now, this does maintain the advantages of a very strict separation of concerns: JavaScript is only ever defined in dedicated JavaScript files, and HTML is only ever defined in dedicated `.hbs` files. However, the *point* of that separation of concerns is not separate files, but the ability to *reason about* and *work with* those concerns discretely. As we'll see below, we can maintain that without this specific limitation on what can and cannot be defined adjacent to a template.

In sum, the imports-only/front matter design appears simple, and is the smallest change relative to today's baseline, but it actually has some pretty significant quirks for teaching developers new to Ember around file extensions and around the semantics of the "scope" available to templates—and at least some of those scope concerns are weird for *existing* Ember users as well.

In my view, this approach simply doesn't pay for itself in terms of teaching and ability to develop a robust mental model. For all that it initially appears to be the simplest, it requires us to teach a fair bit about JavaScript module semantics *and*

to explain that it's really still just compiling to JS. For another, if we adopted it, we would immediately have people clamoring for the Svelte/Vue-style SFC superset of its functionality—and rightly so! Given which: let's dig into the other options on the table.

The other options

Template-only Let's start by looking at `Greeting`, our simple component which just says "Hello" to the user. The simplest of the remaining options is a Svelte/Vue-style SFC:

```
<p>Hello, {{@name}}!</p>
```

By contrast, there's a bit more overhead with the `<template>` tags design: it requires us to immediately introduce a wrapping `<template>` tag. (Recall from the first post that a single, top-level `<template>` tag is equivalent to writing `export default <template>...</template>`.)

```
<template>
  <p>Hello, {{@name}}!</p>
</template>
```

That's not a *lot* of extra teaching, but it is a real difference. The SFC design has *no* overhead here. In both cases, though, we have something that at first blush looks roughly like "just HTML". By contrast, though, the template literals format requires us to introduce a *lot* more ideas right out of the box:

```
import { hbs } from '@glimmer/component';

export default hbs`
  <p>Hello, {{@name}}!</p>
`;
```

Notice all the additional concepts that presents:

- module imports
- default module exports
- template string syntax

The overhead of these additional concepts isn't a deal-breaker by itself, but it's worth recognizing the jump in complexity for this form. While we shouldn't over-optimize for the simplest case—after all, very few components in our apps and libraries are this simple!—we also shouldn't disregard the simple cases. (There's also another important point here, on the *semantics* of template literals in JavaScript; I cover that in Scope semantics below.)

Right out of the gate, we can see that SFCs do best, template literals worst, and `<template>` tags right in the middle on the scale for progressive disclosure on the *simplest* case.

Introducing imports Once we introduce imports, the dynamics start to change. We'll start once again with SFCs, since they were the "winner" of the first round. To add imports, we need to introduce a `<script>` tag, and add the imports within its body. Here, again, with our `UserOverview` component:

```
<script>
  import Greeting from './greeting.js';
  import WeatherSummary from './weather-summary.js';
</script>

<div>
  <Greeting @name="Chris" />
  <WeatherSummary />
</div>
```

So far, this seems quite reasonable. The scoping rules aren't *quite* the same as in normal JavaScript and HTML, but that's really only because regular HTML doesn't have any notion of components. The import rules *are* the same (except that we would technically need to write `<script type="module">` rather than simply `<script>`). Once again, SFCs look pretty good[^imports-tooling]

Turning next to `<template>` tags and tagged template literals, we see that they share tradeoffs with each other. This makes sense: they're both basically a special kind of JavaScript. First up, with `<template>`:

```
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';

<template>
  <div>
    <Greeting @name="Chris" />
    <WeatherSummary />
  </div>
</template>
```

And now the same with the template strings:

```
import { hbs } from '@glimmer/component';
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';

export default hbs`
  <div>
    <Greeting @name="Chris" />
    <WeatherSummary />
  </div>
`;
```

There is one significant point in favor of `hbs` here: since this file is "just"

JavaScript, the imports all match what you would see on disk (or, in the case of TypeScript, the same thing you would see for any other TS file). (This actually poses its own issues for tooling, but I will take up that issue in the next post.) With the SFC and `<template>` tag formats, we would presumably have different on-disk authoring extensions (perhaps `.gbs` for SFCs and `.gjs` for `<template>`).

Dynamic functionality Next up, we can introduce dynamic behavior into this component, with our `isBirthday` helper. I noted above that the SFC format is basically just the natural extension of the imports-only proposal, and that's most obvious here:

```
<script>
  import Greeting from './greeting.js';
  import WeatherSummary from './weather-summary.js';
  import Celebration from './celebration.js'

  function isBirthday(dateOfBirth) {
    const now = new Date();
    return (
      dateOfBirth.getDate() === now.getDate() &&
      dateOfBirth.getMonth() === now.getMonth()
    );
  }
</script>

<div>
  <Greeting @name={{@user.name}} />
  {{#if (isBirthday @user.dateOfBirth)}}
    <Celebration type='birthday' />
  {{/if}}
  <WeatherSummary />
</div>
```

This Just Works™, exactly the way we would expect—very much *unlike* in our imports-only flow. And again, it has the basic semantics we would expect from a `<script>` tag. The same is true for both our JS formats: we can just introduce a function in local scope, and it's available to use. So, with `<template>`:

```
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';

function isBirthday(dateOfBirth) {
  const now = new Date();
  return (
    dateOfBirth.getDate() === now.getDate() &&
    dateOfBirth.getMonth() === now.getMonth()
  );
```

```
}

<template>
  <div>
    <Greeting @name="Chris" />
    {{#if (isBirthday @user.dateOfBirth)}}
      <Celebration type='birthday' />
    {{/if}}
    <WeatherSummary />
  </div>
</template>
```

With `hbs`:

```
import { hbs } from '@glimmer/component';
import Greeting from './greeting.js';
import WeatherSummary from './weather-summary.js';

function isBirthday(dateOfBirth) {
  const now = new Date();
  return (
    dateOfBirth.getDate() === now.getDate() &&
    dateOfBirth.getMonth() === now.getMonth()
  );
}

export default hbs`
  <div>
    <Greeting @name="Chris" />
    {{#if (isBirthday @user.dateOfBirth)}}
      <Celebration type='birthday' />
    {{/if}}
    <WeatherSummary />
  </div>
`;
```

In each of these cases, things basically "just work" exactly the way we would expect: a function declared in a way appropriate for the format is available to the template to invoke, with no caveats about only supporting imports! Spaces which *look* like JavaScript *are* JavaScript, and therefore spaces which seem like they *should* have JavaScript semantics *do* have JavaScript semantics.

Class-backed components When we turn to class-backed components, the dynamics shift dramatically. The `<template>` and `hbs` designs come off pretty well here, so I'll start by showing the `WeatherSummary` component in each. With `hbs`:

```
import Component, { hbs } from '@glimmer/component';
import { gt, lt } from '@glimmer/helper';

export default class WeatherSummary extends Component {
  @tracked currentTemp;

  interval;

  getWeather = () => {
    this.currentTemp = // something
  }

  constructor(owner, args) {
    super(owner, args);
    this.interval = setInterval(this.getWeather, 10000);
  }

  willDestroy() {
    super.willDestroy();
    clearInterval(this.interval);
  }

  static template = hbs`
    <p>
      The current temperature is {{this.currentTemp}}!
      {{#if (lt 50 this.currentTemp)}}
        Brr! 
      {{else if (gt 80 this.currentTemp)}}
        Yikes! 
      {{/if}}
    </p>
  `;
}
```

And with `<template>`:

```
import Component from '@glimmer/component';
import { gt, lt } from '@glimmer/helper';

export default class WeatherSummary extends Component {
  @tracked currentTemp;

  interval;

  getWeather = () => {
    this.currentTemp = // something
  }
```

```
  constructor(owner, args) {
    super(owner, args);
    this.interval = setInterval(this.getWeather, 10000);
  }

  willDestroy() {
    super.willDestroy();
    clearInterval(this.interval);
  }

  <template>
    <p>
      The current temperature is {{this.currentTemp}}!
      {{#if (lt 50 this.currentTemp)}}
        Brr! 
      {{else if (gt 80 this.currentTemp)}}
        Yikes! 
      {{/if}}
    </p>
  </template>
}
```

Here, there is a clear and consistent connection between the class and the template for the class. (There are problems with the static field definition and `hbs` here, but I will return to those below.) And as I'll demonstrate in the next section, this works consistently even if we have multiple components in the same file. Unfortunately, things aren't quite a nice for SFCs. We end up with something like this:

```
<script>
  import Component from '@glimmer/component';
  import { gt, lt } from '@glimmer/helper';

  export default class WeatherSummary extends Component {
    @tracked currentTemp;

    interval;

    getWeather = () => {
      this.currentTemp = // something
    }

    constructor(owner, args) {
      super(owner, args);
      this.interval = setInterval(this.getWeather, 10000);
    }
```

```
    willDestroy() {
      super.willDestroy();
      clearInterval(this.interval);
    }
  }
</script>

<p>
  The current temperature is {{this.currentTemp}}!
  {{#if (lt 50 this.currentTemp)}}
    Brr! 
  {{else if (gt 80 this.currentTemp)}}
    Yikes! 
  {{/if}}
</p>
```

Up to this point, the scope semantics for the SFC style all more or less matched those from normal HTML and JS. Here, though, there's special-casing for this `export default class`: it magically becomes the `this` of the component. And unlike the `import` statements, where the module semantics more or less match normal HTML, the *export* statements *don't* match. There is nothing at all to indicate that a default export from a given `<script>` tag should have anything to do with the context—we just have to teach it as a bare fact.

We could make other named exports work, but (as I will cover in more detail in the next section) *not as components*. This isn't without precedent in JavaScript frameworks; both Svelte and Vue have special, non-native-HTML semantics for their scoping too, with Svelte in particular doing very unusual things with the semantics of `export`. Given that we're designing this from scratch, though, and that the other options *don't* have this issue, I count this a significant mark against SFCs.

Pulling components into a single file As we come to the final part of our worked example, this problem gets *much worse*. When we go to build up the `UserOverview` component, both `<template>` and `hbs` allow us to make our choices about where each component should live, while the SFC design does not. For example, if the `Greeting` component isn't used anywhere else, and we really only want to extract it for simplicity of working with as a concrete thing of its own (just like we do all the time with functions and classes in JS), we can do that with `<template>`—

```
import WeatherSummary from './weather-summary.js';

const Greeting = <template>
  <p>Hello, {{@name}}!</p>
</template>;
```

```
function isBirthday(dateOfBirth) {
  const now = new Date();
  return (
    dateOfBirth.getDate() === now.getDate() &&
    dateOfBirth.getMonth() === now.getMonth()
  );
}

<template>
  <div>
    <Greeting @name="Chris" />
    {{#if (isBirthday @user.dateOfBirth)}}
      <Celebration type='birthday' />
    {{/if}}
    <WeatherSummary />
  </div>
</template>
```

—or with `hbs`—

```
import { hbs } from '@glimmer/component';
import WeatherSummary from './weather-summary.js';

const Greeting = hbs`
  <p>Hello, {{@name}}!</p>
`;

function isBirthday(dateOfBirth) {
  const now = new Date();
  return (
    dateOfBirth.getDate() === now.getDate() &&
    dateOfBirth.getMonth() === now.getMonth()
  );
}

export default hbs`
  <div>
    <Greeting @name="Chris" />
    {{#if (isBirthday @user.dateOfBirth)}}
      <Celebration type='birthday' />
    {{/if}}
    <WeatherSummary />
  </div>
`;
```

—but *not* with an SFC. This is the fallout of two design constraints:

- having the template be the root primitive, with JavaScript added in via

`<script>` tag
- having, as a corollary, special-cased the default export from the `<script>` tag to become the `this` for class-backed components

Both of these mean that a given SFC can always and only define exactly one component—even if there are perfectly good reasons to define multiple components in a single file. As a result, SFCs are somewhat arbitrarily hobbled here, not unlike JavaScript functionality more generally in the imports-only design.

Scope semantics

This leads directly into one of the key tradeoffs for the design: how each of these deals with scoping, and particularly around the mental model we have of scoping.

I suggested this above but it's important to emphasize again: scope semantics is *the* biggest weak point of SFCs for teaching and mental model.[7] This is also where the downsides of the `hbs` design really show up: While it uses template literal *syntax*, it does not have template literal *semantics*. Any normal template literal string in JavaScript can use expression interpolation, including tagged template literals. Accordingly, this is perfectly legal syntax for any `.js` (or `.ts`) file:

```
import { hbs } from '@glimmer/component';

const BREAKFAST = 'Waffles are yummy';

const Breakfast = hbs`<p>${BREAKFAST}</p>`;
```

Unfortunately, this example doesn't work in Glimmer or Ember apps; in fact, it is a build error! You will something like this:

```
path/to/app/app/components/breakfast.js: placeholders inside a tagged template string are not suppo
  4 |
  5 | const Breakfast = hbs`&lt;div&gt;${somethingInScope}&lt;/div&gt;`;
  6 |                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This is because `hbs` is *not* a template literal, but a compile-time macro, which repurposes template literal string syntax for something with totally different semantics. While we could explain that Glimmer and Ember special-case this particular template string and compile it out, it will surprise developers coming from other frameworks (or from no framework at all).

It doesn't stop there, either. Recall our definition of a class-backed `WeatherSummary` component from above:

```
import Component, { hbs } from '@glimmer/component';
import { gt, lt } from '@glimmer/helper';
```

[7] The other big weak point is around testing, and it's actually closely related to this dynamic, as I will cover in detail in Part 4.

```
export default class WeatherSummary extends Component {
  // the rest of the class...

  static template = hbs`
    <p>
      The current temperature is {{this.currentTemp}}!
      {{#if (lt 50 this.currentTemp)}}
        Brr! 
      {{else if (gt 80 this.currentTemp)}}
        Yikes! 
      {{/if}}
    </p>
  `;
}
```

This definition provides the template as a static class field syntactically, but that's a mismatch: The `this` value of a static field is *not* an instance, but rather the class itself, so `this.currentTemp` is *wrong* for a static field. You can see this with a very simple example:

```
class Example {
  anInstanceProp = 123;

  static demoStatic =
    `this.anInstanceProp (static) = ${this.anInstanceProp}`;

  demoInstance =
    `this.anInstanceProp (instance) = ${this.anInstanceProp}`;
}

console.log(Example.demoStatic);
console.log(new Example().demoInstance);
```

This will print:

```
this.anInstanceProp (static) = undefined
this.anInstanceProp (instance) = 123
```

It only works in Glimmer components because the build rewrites the declaration into a relationship in a `WeakMap`.[8] This means, again, that developers coming to Ember cannot use their existing JavaScript knowledge to understand these semantics—indeed, their existing JS knowledge will *actively mislead them* here, as it does with `hbs`'s altered semantics.

What's more, as I'll cover in Part 4 of the series, this is a significant (though solv-

[8] This is a key point we will also return to in our discussion of tooling impact in the next post, and it's one reason why I'm unpersuaded that the template literal string syntax makes the story better for running without compilation.

able) challenge for some ecosystem tooling for the same reason: `static` doesn't actually have the right semantics.

The net of these is that `<template>` actually has a substantial advantage over both SFCs and `hbs` here. Things in lexical scope are available in the body of the `<template>`. However, unlike with template strings, there is no possibility of confusion with the existing semantics of JavaScript—neither for how to include values from the surrounding scope nor for how the template is connected to backing class. The use of the `<template>` tag (as well as a dedicated file extension) makes obvious that there is a language shift: this isn't JavaScript anymore.

On `<template>` semantics

As a counter to that last point, though, it's important to note that `<template>` *also* has established semantics from HTML. Quoting MDN:

> The `<template>` HTML element is a mechanism for holding HTML that is not to be rendered immediately when a page is loaded but may be instantiated subsequently during runtime using JavaScript.
>
> Think of a template as a content fragment that is being stored for subsequent use in the document. While the parser does process the contents of the `<template>` element while loading the page, it does so only to ensure that those contents are valid; the element's contents are not rendered, however.

As with `hbs`, this is a kind of "uncanny valley" problem. Our proposed use of `<template>` looks *similar* to the platform use, but is different enough that any existing knowledge is inapplicable and indeed misleading. I suspect most working front-end developers are unfamiliar with `<template>`,[9] which *reduces* the impact from the overlap… but the overlap really does exist, and it could result in very odd code in the rare cases where a Glimmer developer needs to use the *actual* `<template>` tag:

```
const ComponentWithActualTemplate = <template>
  <template id="actual-template">
    {{! some other content to fill in... }}
  </template>
</template>
```

One option here is to consider a name or design for this which *isn't* `template`—or even to capitalize it: `<Template>` is, for parsing purposes, distinct from `<template>`, which is one of the reasons that most front-end libraries now use it for component invocation. There are other options here, as well, like `<Glimmer>`. None of those are without their own downsides, including that existing apps

[9] I didn't even know `<template>` existed until it came up in early discussions with pzuraq and the Typed Ember team as we played with different designs and experimented with it in the Glint alpha period!

might already be using whatever name chosen, and would need to refactor to switch.

Net, though, I don't think the overlap with `<template>` is intractable. This is in contrast to the use of tagged template literals, which are impossible to make compatible with JS developers' existing knowledge and expectations.

Summary

This post has covered a *lot* of ground, so I'm going to wrap up with an overview table which summarizes my take on it. (I'm trying very hard to be fair to each of the options available here, so feel free to let me know if you think I'm unfairly categorizing the tradeoffs!)

Consideration

<template>

Template literals

SFCs

Imports-only

Progressive Disclosure

Good

Bad

Very good

Good

JavaScript semantics

Good

Good

Good

Very bad

Scope semantics

Very good

Bad

Okay

Good

Semantic mismatch

Yes/HTML/tractable

Yes/JS/intractable

No

No

Looking at the whole picture like this, I would go so far as to say that for a single component with no backing class (and leaving aside considerations about testing we'll get to later in the series, especially around testing), the SFC approach is the best design choice. It starts with plain HTML, and then adds dynamicism via a `<script>` tag—just like the code we would write if we were targeting the browser with no compile step, even including the scoping rules.

Notably, `<template>` is close behind here, though. The only additional factor in the base case is the wrapping `<template>` tag—whereas `hbs` *immediately* introduces full JavaScript semantics. What's more, when we add in the constraint of trying to minimize *surprise*, the SFC design ends up falling down in a couple key areas, particularly around the relationship with the scope and export rules; and the `hbs` design has even worse semantic mismatches with JavaScript. I think it's fair to call `<template>` the winner here. It's not that it's perfect: it's only the absolute winner in one category, and has its own quirk with existing HTML semantics. But it averages out much better across the board than any of the others.

I'll add in conclusion here: this summary isn't just my justification for my preferred design. Rather, it was thinking through exactly these tradeoffs which *made* this my preferred design.

Next up: the impact on **Tooling**, both for individual codebases and for ecosystem tooling!

Part 3 – Tooling

Evaluating the tradeoffs of template language designs for tooling.

In this, the third of a planned five-part series on Ember's *template imports*, I am digging into the implications of each of the designs for ecosystem tooling. Previously:

- Part 1: Introducing the series and walking through the formats.
- Part 2: Which template imports design has the biggest set of wins for teaching and understanding components?

Recall from those posts that there are four basic formats under discussion:

- `<template>` tags with a custom file extension (currently `.gjs` and `.gts`)
- template literals using an `hbs` literal
- something like Svelte's and Vue's SFC format
- an imports-only extension of the current format

In those previous posts, I said Part 3 was going to be about **Scaling**. However, I think it's more useful to talk about **Tooling** here. As I have kept working on this series, I'm not actually persuaded that there are particularly meaningful differences between these approaches for scaling codebases which aren't subsumed in the other topics—especially teaching and testing. So: tooling it is!

Overview

There are (at least) five broad categories to consider in evaluating the impact of these formats in terms of tooling:[10]

- basic editor integration, e.g. syntax highlighting, code folding, etc.
- lint tooling, e.g. ESLint and ember-template-lint support
- formatter support, e.g. Prettier integration
- language server tooling, including ELS and Glint, and interactions with existing TypeScript support
- running the components in a server-side context like Fastboot

(Notice that TypeScript support cuts across several of these in various ways, but is most pronounced in the final point.)

Spoilers for this post: there aren't actually any *great* outcomes here, my preferred `<template>` included. All of them have pretty significant downsides. The "good news" is that the same thing is true for the formats chosen by Vue and Svelte, and that hasn't been a serious hindrance to either of them. The problem is tractable for us, too—but it's probably *not* tractable unless we pick one format and commit

[10] I provide *examples* here in terms of things like ESLint and Prettier, but it's important to recognize that these are *categorical* costs. If we choose at some point to switch our linting over to something like RSLint for the sake of its speed, we would have to pay any costs associated with a given format there as well.

to it, simply because there *is* a lot of work to be done and we are a fairly small community.

React also has a custom syntax extension, but has had support for it built natively into the single two highest impact tools for web developers over the last half decade: the TypeScript Language Service and Visual Studio Code. I strongly suspect *all* the non-React tool maintainers wish that system were pluggable! For the sake of this particular post, I take the *status quo* as a given, though. It hasn't changed in the last half decade, and I don't expect it to any time soon.

Syntax

There already exists a *basic* degree of support for all of these formats in terms of syntax highlighting and and code folding—albeit with some important caveats.

- The imports-only format "works" in all editors I have tried, but without any JS-specific formatting for the imports section. It is simply presented as plain text.
- SFCs get syntax highlighting "out of the box" by VS Code if you use a `.hbs` extension, which makes sense: the Handlebars syntax highlighting is an extension of basic HTML highlighting, and VS Code's HTML highlighter has built-in support for embedded langauges in `<script>` and `<style>` tags. Other editors—including Vim, Sublime Text, and IntelliJ—generally work similarly here, and for the same reason.
- Both`hbs` and `<template>` have at least some degree of syntax highlighting support via various editor extensions, e.g. vscode-glimmer for VS Code, which also adds support for treating `.gjs` and `.gts` as aliases for the JS and TS syntaxes respectively.

  (Notably, however, for reasons I will cover below, just aliasing to JS and TS is actually *not* a great move for `.gjs` and `.gts` files, for reason I discuss below under Lint tooling. If you try this today, you will see red squiggles *everywhere* in VS Code and possibly other editors.)

In sum, as far as the most basic editor integration goes, these are all basically a wash.

The same rough mix of support for existing syntax highlighting tooling exists on GitHub, GitLab, and Bitbucket. You can see this in practice by taking a look at how GitHub renders Part 2 of this series. It works surprisingly well already across the board. The `hbs` and imports-only modes coming out *worst* in that they just parse as strings and get no highlighting. Both SFCs and `<template>` highlight reasonably well.[11]

[11] That `<template>` more or less works surprised me; it appears to be a function of the overloading of `<template>` discussed as a downside in the last post. Supporting something like `<Template>` or `<Glimmer>` would require more work: it highlights more or less reasonably (though not necessarily "correctly") until the closing `</Glimmer>`, which does *not* highlight correctly—but content after it

Lint tooling

When we come to lint tooling—specifically, ESLint and ember-template-lint—the long and short of it is that *nothing* works particularly well, but SFCs and template literals come out slightly better in one specific way, imports-only basically neutral, and `<template>` slightly worse.

Out of the box, the existing linting tools simply do not understand template imports. In this regard, it's a level playing field: all of the approaches on offer will have to implement custom handling. In particular, all of them incorrectly flag anything used only in the template as an unused value. That includes the imports-only mode! For example, in this lightly-modified version of an example from Part 2, the `Greet` and `WeatherSummary` components here will both be marked as unused imports, and `isBirthday` will be flagged as defined but never used.

```
import { hbs } from '@glimmer/component';
import Greet from './greet.js';
import WeatherSummary from './weather-summary.js';

function isBirthday(dateOfBirth) {
  // ...
}

export default hbs`
  <Greet @name={{@user.name}} />
  {{#if (isBirthday @user.dob)}}
    <p>Happy birthday!</p>
  {{/if}}
  <WeatherSummary />
`;
```

Likewise, in the backing class for this variation on the `WeatherSummary` component from Part 2, the `getCurrentTemp` method and the `isSet` helper function will both be flagged as unused:

```
import Component, { hbs } from '@glimmer/component';
import { tracked } from '@glimmer/tracking';

const isSet = (val) => val != null;

export default class WeatherSummary extends Component {
  @tracked currentTemp;

  getCurrentTemp = () => {
    this.currentTemp = Math.floor(Math.random() * 100));
  };
```

highlights correctly again.

```
  static template = hbs`
    <button
      type='button'
      {{on "click" this.getCurrentTemp}}
    >
      Check the weather
    </button>

    {{#if (isSet this.currentTemp)}}
      <p>The current temperature is {{this.currentTemp}}</p>
    {{/if}}
  `;
}
```

We'd see exactly the same warnings about unused code in an SFC format, and for the same basic reasons: we have to inform the JavaScript and template linters about values in the *other* language.

Now, all of the options other than the imports-only format work more or less correctly for all of the parts of any given module which *aren't* related to templates. (The imports-only format simply doesn't connect the two layers at all at present, so there are no false positives… but I don't think we can call that a win.) Perhaps surprisingly, `<template>` actually *appears* to do slightly better than the others in terms of recognizing that we are actually using values in module scope—but this is because it's attempting to parse `<template>` and the contents of it as JSX. This means that editors which use the TypeScript Language Service (including for their JavaScript support) get *very* confused and report syntax errors everywhere, because Glimmer templates and JSX aren't compatible.

The net here is that we have to implement a custom parsing layer for *any* of these formats to have usable linting integration.

Formatter tooling

The story is similar for existing formatter tooling with Prettier: *none* of the formats work well across the board.

- Out of the box, Prettier basically works for the *template* side of SFCs, but doesn't work at all for the JS side.
- Exactly the inverse is true for template literals: Prettier works for the JavaScript side but not at all for the content between `hbs`, which it treats as a string (no surprise there: remember from Part 2 that that's exactly what it is semantically!).
- For the `<template>` proposal, Prettier simply fails to parse, and so formatting does not work at all.

- With the imports-only proposal, like SFCs, Prettier works for the template side, but doesn't format the imports section at all.

In sum, as with the lint tooling, we actually need to implement custom language support to make *any* of these work correctly. However, it's worth acknowledging that the template literals and SFC proposals are halfway there, whereas (in very different ways) the `<template>` and imports-only proposals are much worse off.

Language server tooling

Finally, we come to language server tooling and integration. Most of the JavaScript ecosystem uses the TypeScript Language Service to support features like documentation-on-hover, go-to-definition, and refactoring. That includes React, since TS has built-in support for JSX; Vue, Angular, and Svelte via custom language server integrations; and Ember/Glimmer, via the various experimental ELS implementations and Glint (which it itself used by some of the other language servers). With any of the proposed formats, we would need to create a language server which understood the format and could connect it to the TS LS.

Per Dan Freeman and James Davis, who built and maintain Glint, there is very little difference in effort in supporting `hbs` vs. `<template>`, and because these all compile to the same primitives, even SFCs are tractable. The main challenge there is handling the same custom scoping semantics with the default export as I described as odd in Part 2. However, that is the same basic issue as supporting Glimmer components in Ember apps *today*: something Glint and the experimental ELSs already do.

Notably, Glint also supports GlimmerX, which uses the same syntax as the template literals proposal. This means that we get the integration "for free" (really, for Dan's and James' hard work in 2019–2020). To get the same support for `<template>`, we would need to update the implementation of the Babel transform for `<template>` to provide some data about the original string, so that we can map invocations, error messages, and so on. We would have to build something similar for SFCs (albeit from scratch, since no implementation exists whatsoever yet for them).

However, there's a problem here that's easy to miss: because we're giving new semantics to template literal strings, we have to override existing TypeScript's existing understanding of what JS and TS files mean. In all cases, this is *work*.

- For the `<template>` proposal, this is somewhat tractable and there are a variety of ways to approach it: the custom language integration means we can potentially leave "normal" TS files alone and just *add* information to TypeScript via something like Glint. Doing it that way requires doing a build pass to provide the info, though. The alternative is to disable the TS LS in favor of something like Glint.
- For SFCs, the story is very similar to that with `<template>`, though with a qualification: we could do similar to what Vue's Vetur language server does

and provide a blanket type definition for TypeScript, roughly like this:

```
import Component from '@glimmer/component';
declare module '*.glimmer' {
    export default class extends Component {}
}
```

That would make the TypeScript side type-check—though not particularly *helpfully*—so a tool like Glint would then do its own pass over those as well. You will end up with multiple layers of feedback in your editor—one from TSServer and one from Glint—but that may be fine (and if we went that direction, we could provide a language server plugin to make that experience nicer). All of these proposals require Glint running over top of `tsc`/TSServer; the difference here is taht it means that you don't have to disable the original TSServer to make your editor work.

- In the case of imports-only, we *have* to disable the TS LS, because we have to stitch the script and template files together to create the correct context. Otherwise, the backing classes will always and unavoidably report that there is no usage of anything which is only used in templates.
- For the template literals proposal, it might initially seem like we could just integrate with the TS LS plugin tooling. After all, the docs say that one of the intended uses for plugins is:

  > Enable new errors or completions in string literals for a custom templating language

  Unfortunately, they also specify that one of the things language plugins cannot do is:

  > Customize the type system to change what is or isn't an error when running `tsc`

  The net of this is that we can *add* errors in a standard TS LS plugin, but we cannot *remove* them. We're stuck with all those warnings about unused values! What's more, "plugins aren't loaded during normal commandline typechecking or emitting." This is why Glint works the way it does today: as a custom language server and CLI. To get consistent behavior between our build/CI environments and our editors, we still need to run everything through a custom pipeline. That would leave us with the downside of having two separate paths for type-checking vs. editor support… or with the alternative of disabling existing TypeScript support for those files.

Net, the `hbs` implementation initially appears to have a very small edge on language server implementation, because it already exists in support of GlimmerX—but we should not take this as a particularly important constraint: it's basically just slight variations on the same underlying sets of tradeoffs. (Once again, everyone who isn't using JSX *really* wishes that the syntax extensions part of TS were pluggable. Alas.)

Server-side

One common reason a few people have suggested we ought to prefer `hbs` is that they think it makes it easier to support running Glimmer templates in server-side environments—that is, that it would make it viable to use `hbs` as an actual import which works without needing any compilation step.

Unfortunately, while that sounds appealing, it isn't actually true, at least today! The problem is the mismatch in semantics discussed in Part 2:

1. The scoping semantics are wrong, because `hbs` literals aren't actually string literals.[12] Remember: if you write this code…

   ```
   const Greeting = hbs`<p>Hello, {{@name}}!</p>`

   const Summary = hbs`
     <Greeting @name={{user.name}}>
     {{! ... }}
   `;
   ```

   …it *does not work without a transform*. `Greeting` isn't "in scope" for the `Summary` component! While we could work around this by introducing some other invocation form for `hbs` where it also takes the scope as an argument, that makes the ergonomics *much* worse—close, in fact, to the original `precompileTemplate` invocation that *is* the compile target.

2. Additionally, remember that the scoping semantics are *also* wrong when you switch to a component with a backing class, because of the mismatch between a `static` class field and the semantics of component templates. If you want the `this` value to work correctly, you *have* to introduce some degree of processing. At a minimum, we would need to rewrite the internals of `getComponentTemplate` to go look up that static field—not an impossible hurdle, by any means, but a real and significant change to the current design. (As to whether it's otherwise well-motivated, I refer you to the rest of the series!)

Moreover, I'll go further here and say that I don't think there's any particular value to being able to run a Glimmer component without any build step. Having a build pipeline is *extremely* normal for both client- and server-side code—and it can even be done fairly transparently and on demand for server-side code, e.g. with `@babel/node` or `ts-node`. If someone wants to run Glimmer component code natively in a Node runtime, they can precompile it using our standard build tools *or* they can simply use `@babel/node` to integrate the transform automatically.

Net, I take this to be something of a non-issue for the design choice here, as it requires *some* non-zero degree of extra work compared to today's baseline

[12] Dan Freeman pointed this out on the Ember Discord in response to the update where I added this section. I knew this, and in my COVID-recovery-induced mental haze, I totally forgot it. Thanks, Dan!

regardless and there are straightforward options for this regardless of the design chosen.

Summary

In this particular comparison, the template literals proposal clearly comes out with a *small* edge. In most categories, it's the same or slightly better than the other options, as we can see in this table:

Consideration

<template>

Template literals

SFCs

Imports-only

Syntax

JS

working

working

working

not working

templates

working

working

working

working

Linting

JS

partial

partial

partial

no

templates

no

no

no

yes

needs custom parser

yes

yes

yes

yes

Formatting

JS

no

partial

partial

no

templates

no

partial

partial

no

needs custom parser

yes

yes

yes

yes

LS effort

small

none

medium

small

Server-side requires compilation

yes

yes

yes

yes

These differences are very small, though. Accordingly, I still believe `<template>` is the best choice—because the small deltas here are fairly straightforward to tackle, and because I think the issues around **Teaching** described in Part 2 and around **Testing** as I will describe in Part 4 *profoundly* outweigh these small tooling differences.

Part 4 – Testing

Keeping, and improving on, one of Ember's fundamental commitments—and biggest strengths: its integrated testing.

Previously I've introduced the alternatives, discussed each approach's implications for **teaching and understanding**, and looked a bit at how the different approaches might play out in terms of **tooling**. In this post, I turn to *testing*: something Ember has long focused on, and an area that it's important to avoid regressing!

With the Glimmer Component 2 API, there is potentially an even broader range of testing options than Ember's current support. For example, it would be straightforward to implement `@testing-library/glimmer` to match the other testing-library approaches. However, *all of the concerns and constraints I lay out below hold in that case*: it would be an *expansion* of our current story, if we make the right choices. (The fact that other libraries and frameworks may not have a great story here doesn't mean *we* shouldn't.)

Today, Ember allows you to author local snippets of template in tests, which makes certain kinds of component render testing *very* straightforward. This is the familiar inline `hbs` invocation:

```
import { module, test } from 'qunit';
import { setupRenderingTest } from 'ember-qunit';
import { hbs } from 'ember-cli-htmlbars';
import { render } from '@ember/test-helpers';

module('Rendering | Component | MyComponent', function (hooks) {
  test('it renders and yields stuff!', async function (assert) {
    await render(hbs`
      <MyComponent as |mc|>
        <div data-test-hammer>{{mc.hammer}}</div>
      </MyComponent>
    `);

    assert.dom('[data-test-hammer]').hasText("Let's Get it Started");
  });
});
```

Notably, this means you can write a test at the level of an individual component, running it in the browser *without* having to spin up the whole application at the level of an end-to-end test. This is very useful for testing components in isolation and exercising *just* their own API, without testing everything else any given component may be integrated in. It lets us test the interface for our components.

Both the template literals form and the `<template>` tag form would work well

with this. In the case of template literals, the only thing we would change for tests would be the import path for `hbs`, and in principle we wouldn't even have to change that: the export from `ember-cli-htmlbars` could simply become a re-export from `@glimmer/component`:

```
import { module, test } from 'qunit';
import { setupRenderingTest } from 'ember-qunit';
import { hbs } from '@glimmer/component';
import { render } from '@ember/test-helpers';
import MyComponent from '../app/components/my-component';

module('Rendering | Component | MyComponent', function (hooks) {
  test('it renders and yields stuff!', async function (assert) {
    await render(hbs`
      <MyComponent as |mc|>
        <div data-test-hammer>{{mc.hammer}}</div>
      </MyComponent>
    `);

    assert.dom('[data-test-hammer]').hasText("Let's Get it Started");
  });
});
```

For `<template>` tags, we would simply drop the `hbs` import and replace its usage with `<template>`:

```
import { module, test } from 'qunit';
import { setupRenderingTest } from 'ember-qunit';
import { render } from '@ember/test-helpers';
import MyComponent from '../app/components/my-component';

module('Rendering | Component | MyComponent', function (hooks) {
  test('it renders and yields stuff!', async function (assert) {
    await render(<template>
      <MyComponent as |mc|>
        <div data-test-hammer>{{mc.hammer}}</div>
      </MyComponent>
    </template>);

    assert.dom('[data-test-hammer]').hasText("Let's Get it Started");
  });
});
```

What's more, because of the rest of the benefits available via strict mode, it becomes that much easier to introduce a test-only component. Historically, to do that we have had to use the registry to do this, with patterns like this:

```
// imports and setup
```

```
this.owner.register(
  'component:test-dummy',
  setComponentTemplate(
    hbs`
      <button
        type='button'
        data-test-cool-button
        {{on "click" this.handleClick}}
      >
        Click!
      </button>
    `,
    class CustomButton extends Component {
      handleClick = () => {
        assert.ok(true, 'got clicked!');
      };
    }
  )
);
```

Once we have either the template literals or `<template>` proposals, this becomes much easier. It's exactly the same as authoring a component in the first place! This is actually a bonus for our teaching story, which I intentionally skipped over in Part 2 because it's such a key point of this part. Instead of needing to teach the "special sauce" for testing, authoring a local-only component is identical in app code and test code.

With `<template>`, for example:

```
import { module, test } from 'qunit';
import { setupRenderingTest } from 'ember-qunit';
import { click, render } from '@ember/test-helpers';
import Component from '@glimmer/component';
import MyModal from '../app/components/my-modal';

module('Rendering | Component | MyComponent', function (hooks) {
  test('it renders and yields stuff!', async function (assert) {
    // one for rendering, one for interacting
    assert.expect(2);

    class CustomButton extends Component {
      handleClick = () => {
        assert.ok(true, 'got clicked!')
      }

      <template>
        <button
```

```
          type='button'
          data-test-cool-button
          {{on "click" this.handleClick}}
        >
          Click!
        </button>
      </template>
    }

    await render(<template>
      <MyModal @closeButton={{CustomButton}} />
    </template>);

    assert
      .dom('[data-test-cool-button]')
      .exists('the button gets rendered');

    // Trigger the button click, which will trigger the `handleClick` assertion
    // if the component is wired up correctly!
    await click('[data-test-cool-button]')
  });
});
```

This would be effectively identical with `hbs`, with the relevant substitution of `static template = hbs...`.

see it with hbs

```
import { module, test } from 'qunit';
import { setupRenderingTest } from 'ember-qunit';
import { click, render } from '@ember/test-helpers';
import Component, { hbs } from '@glimmer/component';
import MyModal from '../app/components/my-modal';

module('Rendering | Component | MyComponent', function (hooks) {
  test('it renders and yields stuff!', async function (assert) {
    // one for rendering, one for interacting
    assert.expect(2);

    class CustomButton extends Component {
      handleClick = () => {
        assert.ok(true, 'got clicked!')
      }

      static template = hbs`
        <button
          type='button'
```

```
          data-test-cool-button
          {{on "click" this.handleClick}}
        >
          Click!
        </button>
      `;
    }

    await render(hbs`
      <MyModal @closeButton={{CustomButton}} />
    `);

    assert
      .dom('[data-test-cool-button]')
      .exists('the button gets rendered');

    // Trigger the button click, which will trigger the `handleClick` assertion
    // if the component is wired up correctly!
    await click('[data-test-cool-button]')
  });
});
```

This would be impossible with the Vue/Svelte-style SFC or with the imports-only proposal. While you can absolutely test *existing* components with those formats by importing them and rendering them, you cannot author new ones locally. That means that to avoid regressing our existing test infrastructure—a non-negotiable—we would need to maintain a separate syntax used only in testing to support these kinds of tests. It would, most likely, need to be something roughly the shape of the template literals proposal, in fact—that's the lowest lift from where we are today.

The fallout here is pretty significant in my view:

- To make that a good experience, we would need to make the full investment in tooling described in Part 3… for *both* formats. That's double the effort up front, and double the ongoing maintenance costs. The only alternative would be leaving testing a second-class experience, and I think that's a non-starter.
- It would leave us in the position we are in today, where we have different behavior for tests than for runtime code—and have to teach accordingly. But one of the big potential upsides of template strict mode is that we can significantly reduce the number of concepts Ember and Glimmer developers need to learn to be productive. These kinds of opportunities don't come along often; we shouldn't squander this one.
- Experientially, we would be dangling in front of people the ability to author local components, and then telling them they can't do it in app code. Per-

> sonally, I would find that incredibly frustrating. More and worse, I cannot imagine trying to explain it to the hundreds of developers I support on the flagship app at LinkedIn!

From where I stand, this is *the* fundamental reason to use either `hbs` or `<template>` over a Vue- or Svelte-style SFC approach. Testing is a fundamental concern of building applications, and Ember has long done well to prioritize it. The design for template imports should take that as a fundamental constraint as well.

As for the differences between `hbs` and `<template>` here: there really aren't any from a testing point of view. There *is* a migration cost if we use `<template>`, since all existing integration tests would have to be rewritten in terms of the new syntax. However, this cost is very low: it is straightforward to write a thorough and robust codemod for it. Any existing `render()` call with an `hbs` body can be replaced with one a `<template>` body instead. The semantics are identical, and both are equally flexible for defining local components. The tradeoffs besides migration reduce to those discussed in previous posts.

Thus, as with the **Tooling** discussion in Part 3, `hbs` has a slight edge over any other proposal simply in terms of existing usage. However, beyond that, both `<template>` and `hbs` both are *far* better than imports-only or SFCs in this case. For those keeping score, that makes my current evaluation:

- **Teaching and understanding:** `<template>` the winner across the board, followed by SFCs, then imports-only, then `hbs`
- **Tooling:** `hbs` only *slightly* better than the other options
- **Testing:** `hbs` slightly better than `<template>` because it has effectively no migration cost; but both far better than either the SFC or imports-only proposals

The net, as I'll cover in a bit more detail in the conclusion, is that even though `hbs` comes out slightly ahead in the tooling and testing categories, I think `<template>` remains the clear winner overall.

Part 5 – Styling

Styling and presentation are fundamental aspects of authoring user interfaces. So how does CSS work with the various Ember template imports proposals?

Previously in this series, I have covered **Teaching and Understanding**, **Tooling**, and **Testing**. I had planned for this fifth post in the series to be the *final* post in the series, summarizing the tradeoffs and concluding my case for `<template>`. However, as a few folks have mentioned styles over the course of the last many months, I thought it would be helpful to add one extra post discussing CSS in particular.

Fundamentally, there are two basic approaches available for styling in the modern front-end ecosystem—everything out there is a variation on one of these themes:

1. **Standalone style definitions.** This is the classic approach: defining styles in a completely separate context from the JavaScript and HTML. This can be accomplished any number of ways, from a single monolithic CSS file to a bunch of files integrated together via a build step with Sass or PostCSS to atomic styles like Tailwind or Tachyons.

2. **Integrated style definitions.** This is the approach which became much more popular in the late 2010s: style definitions which are directly related in some way to UI components. This includes CSS Modules, styled components, Emotion, vanilla-extract, and no doubt more. Notably, many of these can *also* work with some of the preprocessing tools from the classic build approach. The big upsides to these are usually tooling integration and built-in per-component scoping.

All of these work perfectly well with any of the template imports proposals. This is the reason I originally left this aside, and it's also the reason this is the shortest of the posts. The templating layer is a fundamental part of a front-end framework: it is *the* thing a front-end framework *must* do. By contrast, a framework can be styling-method-agnostic. Frameworks certainly can provide first-class primitives for it—as Vue and Svelte do—but it is not a necessary constraint for the integration of *state change* with *DOM updates*, unlike templating.

This is by no means to diminish the importance of styling, which I take to be an under-appreciated element of truly great web applications. It is rather to make clear and explicit what constraints it does and doesn't face compared to templates.

There are certainly interesting moves we could make with styles in any of the proposals. It's easy to imagine introducing a `<style>` block alongside the template and `<script>` tags in an SFC design, much as Svelte has done:

```
<script>
  const isChristmas = date =>
    date.getMonth() === 11 &&
    date.getDate() === 25;
</script>
```

```
<style>
  .christmas {
    color: green;
    background: red;
  }
</style>

{{#if (isChristmas @today)}}
  <p class='christmas'>Merry Christmas!</p>
{{/if}}
```

Notice, however, that even here with an SFC design we do not *need* to use a `<style>` tag, because the semantics of template imports make JavaScript values available to the template. If we wanted to use vanilla-extract instead, we could do that:

```
<script>
  import { style } from '@vanilla-extract/css';

  const christmas = style({
    color: 'green',
    background: 'red',
  });

  const isChristmas = date =>
    date.getMonth() === 11 &&
    date.getDate() === 25;
</script>

{{#if (isChristmas @today)}}
  <p class={{christmas}}>Merry Christmas!</p>
{{/if}}
```

Exactly the same holds for the template literals and `<template>` designs: as long as we provide a value in a way that can be integrated into the `precompileTemplate` invocation which *all* of these formats compile to, it will "just work."[13] And

[13] If you'd like to prove this to yourself, create a new app on Ember 3.28.4+, and make a component JavaScript file with this body and render it:

```
import { precompileTemplate } from '@ember/template-compilation';
import { setComponentTemplate } from '@ember/component';
import templateOnly from '@ember/component/template-only';
import { helper } from '@ember/component/helper';

const Style = {
  christmas: 'christmas_1234abcd',
  normal: 'normal_9876beef',
};
```

indeed, that means that these even work with imports-only, though with the usual constraints:

```
import { style } from '@vanilla-extract/css';

export const christmas = style({
  color: 'green',
  background: 'red',
});

export const isChristmas = date =>
  date.getMonth() === 11 &&
  date.getDate() === 25;
```

```
---
import { isChristmas, christmas } from './christmas.js';
---

{{#if (isChristmas @today)}}
  <p class={{christmas}}>Merry Christmas!</p>
{{/if}}
```

In the case of the `<template>` design in particular, I can imagine that community members might want to implement a `<style>` element which compiles a CSS language declaration to a scoped value:

```
const Style = <style>
  .christmas {
```

```
const isChristmas = helper(
  ([date]) =>
    date.getMonth() === 11 &&
    date.getDate() === 25
);

const now = helper(() => new Date());

const template = precompileTemplate(
  `
  {{#if (isChristmas (now))}}
    <p class={{Style.christmas}}>Happy Christmas!</p>
  {{else}}
    <p class={{Style.normal}}>Have a nice day!</p>
  {{/if}}
  `,
  {
    scope: { Style, isChristmas, now },
  }
);

export default setComponentTemplate(template, templateOnly());
```

Then you can check that it applies the correct class in the inspector.

```
    color: 'green';
    background: 'red';
  }
</style>

const isChristmas = date =>
  date.getMonth() === 11 &&
  date.getDate() === 25;

<template>
  {{#if (isChristmas @today)}}
    <p class={{Style.christmas}}>Merry Christmas!</p>
  {{/if}}
</template>
```

This is quite nice! But again, the key point is that *this does not need to be built into the framework*: there is no fundamental coupling between the way we generate our styles and the rendering or state management layers. The upside to the template imports design is that any and all such solutions "just work." It may even be that in the future, we as a community find one that we particularly prefer and therefore write an RFC to ship out-of-the-box support for it. But it simply isn't a constraint in any way for the design of template imports.

Conclusion

Given the tradeoffs in the space, what is the best set of compromises we can make?

Over the course of this project, I have systematically examined the tradeoffs of the proposed options for Ember and Glimmer's template imports design:

1. Introducing the series and walking through the formats.
2. Which template imports design has the biggest set of wins for ***teaching and understanding*** components?
3. Evaluating the tradeoffs of template language designs for ***tooling***.
4. Keeping, and improving on, one of Ember's fundamental commitments — and biggest strengths: its integrated ***testing***.
5. What about **styles**? (A bonus post!)

In this final part, I am going to pull all of those threads together and synthesize them into my conclusion: that we should use `<template>`. Not because it's perfect (none of the options are! This is software engineering!) but because it strikes the best balance across all these axes.

When I first started discussing these options with people a few years ago, after the strict mode and initial template imports RFCs were opened, I had a strong bias toward the `hbs` template literal strings design. I have since then come to think that `hbs` is the *second-worst* option, with only imports-only being worse—and that's iffy, because the problems with `hbs` are so bad in my view! I also originally very much disliked SFC designs, but have now come to see them as better than the `hbs` design in many areas, and if it weren't for a couple key limitations they might be my favorite. And the `<template>` proposal moved from being my second-least-favored proposal to the one I think we should adopt.

This was the result of discussing and thinking on all the tradeoffs I've laid out over the past few weeks in this series: what works best and has the fewest problems pedagogically? implementation-wise? for integrating with styles? for migrating existing users? for language server support?

I sympathize with people who still *like* other proposals better. Those feelings, in many cases, mirror my own initial responses in various ways, even if the details differ. I also expect that no matter what we choose, some people will be unhappy. That could include me! I continue to believe it is *most* important that we choose one of these formats and execute on it—even if that ends up being a decision in a direction I think is misguided.

That gets at something important, though: in order to make a decision, we simply have to evaluate the tradeoffs around these designs and do the best we can with the constraints we have. That's why "just use JSX" isn't one of the options I've discussed: JSX doesn't work with the constraints of the Glimmer VM today. It's also why I haven't spoken at all about the aesthetics or questions of taste: we

will likely all differ on which approach we think looks and feels nicest, and those differences are likely intractable. If you think the `hbs` proposal looks nicest, and I think SFCs look nicest,[14] how could we possibly persuade each other? There is, as the saying goes, no accounting for taste.

With that in mind, let's review the conclusions I drew in each of the earlier parts of the series—once again in tabular form:

Consideration

<template>

Template literals

SFCs

Imports-only

Progressive Disclosure

Good

Bad

Very good

Good

JavaScript semantics

Good

Good

Good

Very bad

Scope semantics

Very good

Bad

Okay

Good

Semantic mismatch

Yes/HTML/tractable

Yes/JS/intractable

No

[14] This is in fact what I think: SFCs *look* nicest. But that doesn't make them the best decision.

No
Syntax
JS
working
working
working
not working
templates
working
working
working
working
Linting
JS
partial
partial
partial
no
templates
no
no
no
yes
needs custom parser
yes
yes
yes
yes
Formatting
JS
no

partial

partial

no

templates

no

partial

partial

no

needs custom parser

yes

yes

yes

yes

LS effort

small

none

medium

small

Server-side requires compilation

yes for all formats, including hbs

shared syntax for tests

yes

yes

no

no

Styling

Everything “just works” for all formats

When you put all the pieces together like this, I think it’s fairly obvious why I ultimately concluded that `<template>` is the best choice. In every overarching category except for formatting, it is either comparable to the other choices or

substantially better. It's not perfect. It's more work in a few spots than the alternatives, and in some other cases it's just kind of *the same* as the alternatives. But it does come out the best overall.

As a final note, I think it's worth considering the relationship between the different categories. Tooling costs are real, but they're something we address straightforwardly by building some software. We're software developers; we're very good at building software! While there are some ongoing costs to maintaining software, they're small and tractable, and they only fall on the people who maintain those tools.

By contrast, the problems of teaching and understanding are ongoing, and distributed. Every new developer who enters the Ember/Glimmer ecosystem will have to pay them—for as long as we use whatever format we choose here. In my view, that makes the case for `<template>` even more compelling: it's not just that it comes out the winner in a general analysis (though it does), but that it comes out *by far* the winner in the most important and least-"solvable" category.

As ever, I'm happy to discuss this—and the series as a whole!—on the Ember forums or in Discord. And keep your eyes open for the RFC I'll be opening in the next few weeks advocating that we ship `<template>` as the officially-supported syntax and as part of Ember's upcoming Polaris edition!