Javascript

Modules, GUI components

Bjarte Wang-Kileng

HVL

August 22, 2025

Western Norway University of Applied Sciences

Export and import

- Modules let JavaScript handle its dependencies.
- The import statement lets a JavaScript file import functions, objects or primitives from other JavaScript files.
- The import function let us do dynamic import, e.g. import a module on a condition.
- A JavaScript module must use the export statement to make content available to the importer.
- A JavaScript module will be imported only once.
  - Import of an already imported module is ignored.
- Always import if needed by module.
  - Do not assume that another module already have done the import.

Example with export and import

- HTML file:

```
<head>
    <script src='js/main.js' type='module'></script>
</head>
```

- JavaScript file **js/main.js**:

```
import utils from './modules/utils.js';
```

- JavaScript file **./modules/utils.js**:

```
export default {
    ...
}
```

Observations

- Strict mode is enforced when working with *import*, *export* or *Class*.
- Import URL is relative to the JavaScript file, **not** the HTML file.
  - Path must be a full path, i.e. "./" if current directory.
  - Path can only start with ".", "/" or "http(s)".
- HTML tag *SCRIPT* with *type='module'* implies attribute *defer*.

Types of exports

- Default exports.
  - Zero or one default export per module.
- Named exports.
  - Zero or more named exports per module.
- A true module should define a single class, single object or single custom tag only.
  - Should have no exports if defining a custom tag, or otherwise
  - should have no named exports, and one default export.
- We will work with default exports and custom tags only.

Global exports and naming

- Exported module must be assigned a name on import.
  - Default export:

```
export default ...
```

  - Import of default export must assign a name to the imported module:

```
import ImportName from ...
```

- The import does not need to know the name used by the export.
  - Exported default can be anonymous, i.e. do not need a name.
- Measure against code collisions.
- An import does not need to import any data.
  - Will still run the code of the imported file.
  - Useful if e.g. module creates a custom tag.

Dynamic imports

- The *import* statement will import module whether used or not.
- The *import* function let us do dynamic import.
  - Can import module depending on a condition.
  - Does not require *type='module'* on *SCRIPT* tag.
- The *import* function returns a **Promise**.
  - Dynamic import function is run asynchronously.
  - Details on **Promise** later in the course.

GUI components

- Reusable, custom made elements.
- Must be isolated from the rest of the code.
- Should never cause code collisions.
- Main document and other components should have no access to the elements, properties or structure of the component.
- Component should never access elements of the main document, or elements of other components.
- Communication between components, and component and main document only through component API.

Component isolation aims

- Component CSS should not affect main document, or other components.
- CSS of main document should have no effect on component.
- JavaScript in main document, or other components should not see elements inside component.
  - Main document code below should not include elements of component.

```
const pElements  = document.querySelectorAll('P');
```

- JavaScript of component should not access main document.
  - Not enforced by the technology.

Component solutions

- Client side framework, e.g. React, Vue, Angular, Ember.
  - Can require a build process to assemble the code.
  - Avoids code collisions.
  - Typically no true isolation. Details later.
- Web components through standard JavaScript.
  - Require a modern web browser.
  - Avoids code collisions.
  - True isolation.
  - Used for components in some frameworks, e.g. Lit and Stencil.

Components through standard JavaScript

- Create a new web element by extending the class **HTMLElement**.
  - Class **HTMLElement** is the base class of all HTML elements.
- Isolate the component DOM tree through shadow DOM.
  - Creates a separate DOM that can be made unaccessible from outside.
- Assign a custom tag to the new element.
  - Custom tag names must include a hyphen character.

```
<MY-COMPONENT data-number="2"></MY-COMPONENT>
```

- HTML elements *SLOT* and *TEMPLATE* can be useful for injecting DOM into component.
- HTML tags, and also HTML custom tags are global entities.

Shadow DOM

- A shadow DOM internals are hidden from, and invisible from main document.
- A shadow DOM can have its own *STYLE* and *LINK* HTML elements.
  - Shadow DOM CSS does not affect the main document.
  - Main document CSS does not see the shadow DOM.
- JavaScript in main document, or other components do not have access to shadow DOM in component.
- HTML IDs and styles in shadow DOM do not clash with main document.

Component demo

- Create component:

```
class CourseInfo extends HTMLElement {
  constructor() {
    // Always call super first in constructor
    super();

    // Create a shadow DOM structure
    const shadow = this.attachShadow({ mode: 'closed' });

    // Add some structure to the shadow DOM
    const pElement = document.createElement('p');
    pElement.textContent = 'Welcome';
    shadow.appendChild(pElement);
  }
}
```

- Assign tag *COURSE-INFO* to component:

```
customElements.define('course-info', CourseInfo);
```

Component lifecycle

constructor():
: Run once for each occurence of the custom element in the DOM.

connectedCallback():
: Run when the element has been connected to the DOM.

disconnectedCallback():
: Run when the element has been disconnected from the DOM.

attributeChangedCallback():
: Run if an attribute of the element is added, removed, or changed.

adoptedCallback():
: Run when moved to new document, i.e. owning document change.

  Useful e.g. with iframes.

Use of constructor and *connectedCallback*

- The constructor is run only once for each occurence of the custom element in the DOM.
- The *connectedCallback* can be run more than once, e.g. when component is removed and re-added to the DOM.
- Metods and properties may not be applicable to the component before connected to the DOM.
  - E.g. if adding DOM structures from browser memory to component.

Deciding on the use of constructor or *connectedCallback*

Extract from MDN

The specification recommends that, as far as possible, developers should implement custom element setup in this callback rather than the constructor.

See MDN, section Using custom elements.

Extract from the HTML Living standard

In general, the constructor should be used to set up initial state and default values, and to set up event listeners and possibly a shadow root.

See section 4.13.2 of the HTML Living standard.

Advices on the use of constructor and *connectedCallback*

- Create your shadow root in the constructor.
- Run one-time initialization work in the constructor.
- Add event listeners of component in the constructor.
- Listneres living outside of component should be added with *connectedCallback* and removed with *disconnectedCallback*.
  - Otherwise, e.g. a *window.setTimeout* will continue after component is removed from the DOM.
- Use *connectedCallback* for most other component setup.

Data through content of custom tag

- Data can be given to custom tag as DOM content of custom tag.
- Custom tag with course name set as DOM content:

```
<course-info>
    <ul>
        <li>DAT152</li>
        <li>JavaScript</li>
    </ul>
</course-info>
```

- Accessing attributes *data-course* and *data-topic* from tag class:

```
const liElms = this.querySelectorAll("li");
const course = liElms[0].textContent;
const topic = liElms[1].textContent;
```

Data as attributes of custom tag

- HTML element custom data attributes are accessible in component class, e.g. using element property dataset.

- Custom tag with attributes *data-course* and *data-topic*:

```
<course-info
    data-course='DAT152'
    data-topic='JavaScript'
>
</course-info>
```

- Accessing attributes *data-course* and *data-topic* from tag class:

```
const course = this.dataset.course;
const topic = this.dataset.topic;
```

Using a component API

- Component can define an API for modifying component content.

```
class CourseInfo extends HTMLElement {
    constructor() { ... }

    setCourse(course) { ... }

    setTopic(topic) { ... }

    setLecturer(lecturer) { ... }
}
```

- API can be used by application controller to modify component:

```
const courseelement = document.querySelector('course-info');

courseelement.setCourse("DAT151");
courseelement.setTopic("Database management");
```

HTML tag *SLOT*

- Tag *SLOT* in component can be replaced with DOM:

```
class CourseInfo extends HTMLElement {
    constructor() {
        super();

        this.attachShadow({ mode: 'closed' }).innerHTML=`
            <P>
                Course is <SLOT name="course"><SPAN>DAT152</SPAN></SLOT>.
            </P>
        `;
  }
}
customElements.define('course-info', CourseInfo);
```

- Using the component and inject DOM through the *SLOT*:

```
<BODY>
    <COURSE-INFO>
        <SPAN slot="course">dat151</SPAN>
    </COURSE-INFO>
</BODY>
```

More on *SLOT*

- Several elements can be injected into the same slot.

```
<BODY>
    <COURSE-INFO>
        <UL SLOT='details'>
            <LI>Course name: DAT152</LI>
        </UL>
        <UL SLOT='details'>
            <LI>Course name: DAT151</LI>
        </UL>
    </COURSE-INFO>
</BODY>
```

- The HTMLSlotElement method *assignedElements()* returns list of injected elements.
  - Will be a list of two *UL* elements in the example above.

Document fragment for component

- A DocumentFragment is a segment of a document with no parent.
  - Lightweight container that can hold DOM nodes, i.e. web content.
- Can be used as template for the DOM of a component.
- If using a document fragment for component, always make a clone.
  - The same DOM structure can only occur once in a document.
- Sources for a document fragment for component:
  - HTML element *TEMPLATE*,
  - Document with HTML, fetched from server using Ajax,
  - DOM methods.
- Ajax will be covered later in the course.

HTML element *TEMPLATE*

- HTML tag *TEMPLATE* creates a document fragment.

```
<TEMPLATE id="course-template">
    <P>
        Course name: <slot name="course"><span>DAT152</span></slot>.
    </P>
</TEMPLATE>
```

- The DOM structure of *TEMPLATE* is not displayed anywhere.
  - Exists in browser memory only.
- Cloning a template is faster than creating DOM with JavaScript.
  - Cloning is good if more instances of a DOM structure.
- Template should therefore be a global entity in module, or a static class field.
  - Template as a non-static class field will always be counter productive.

TEMPLATE element for component

```
const template = document.createElement("template");
template.innerHTML = `
    <p>
        Welcome to <slot name="course"><span>DAT152</span></slot>.
    </p>`;

class CourseInfo extends HTMLElement {
    constructor() {
        super();
        const shadow = this.attachShadow({ mode: 'closed' });
        const content = template.content.cloneNode(true);

        // Move DOM of "content" from "content" to "shadow"
        shadow.append(content);
    }

    ...
}

customElements.define('course-info', CourseInfo);
```

On *TEMPLATE* for component

- Use static fields of component class, or constants of JS module.
  - *TEMPLATE* element must be shared by all instances of component.
- When appending DOM of a cloned template, the DOM is moved into the component.
  - Clone will become empty when the content is moved into component.
- Adding event listerens, and also other DOM features may only work on a proper DOM structure.
  - Add event listeneres to the shadow document, not the template, nor a clone.

HTML modules

Read-only, not subject for the exam

- Load HTML document fragment as module from JavaScript code:

```
import courseDocument from "./template.html";
const template = courseDocument.querySelector("template");
const content = template.content.cloneNode(true);
```

  - Not to be confused with HTML imports.

- No browser support yet.
  - Status in Chrome.
  - Possible to use Ajax to import an HTML document into module.

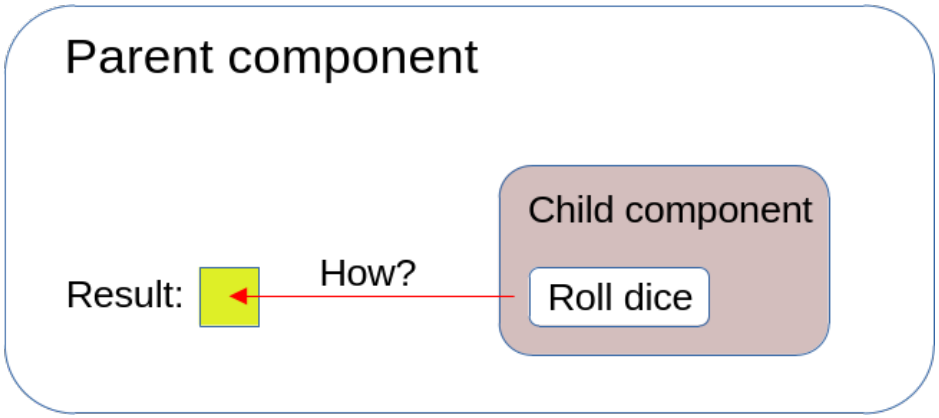
- Proposal for a standard.

Component communication

- A parent component can access a child component.
  - Using child component API.
- Parent-child relation only allow access from parent to child.
  - Child should only access structures within itself, or throug the API of any of its children.
- Components that need mutual access must communicate through a common parent.
  - Lift communication to closest common parent.
- Parent must use child API to register callbacks to be run by child.
  - The child communicates with callback owner by running the callback.

Communication example – Isolated Dice component

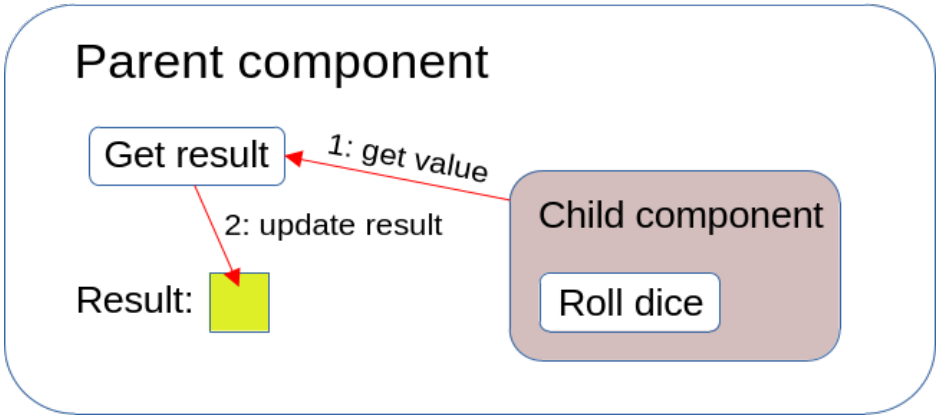
- At click on a button in child, a parent propety must be updated.



- Challenges:
  - The updated value is private to the child.
  - The parent does not know that *Roll dice* has been clicked, nor that a new value has been generated.
- See Eclipse Isolated dice component.

Communication example – Using the Dice component API

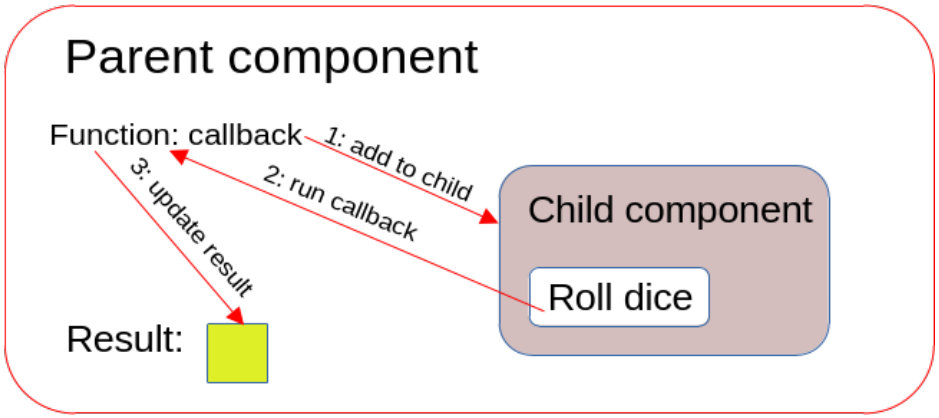
- Parent can use child API to retrieve value, then update the property.



- Requires two buttons and two clicks to update the value:
  - First a click on the child button *Roll dice*, and then
  - click on the parent button *Get result*.
- Solution is not good as one button and one click should be sufficient.
- See Eclipse Dice component with calls from parent.

Communication example – Callback to parent

- Components can communiate using callback of the other.



- Child can call *callback* of parent when user clicks *Roll dice*.
  1. Parent creates child and adds *callback* using child API.
     - Parent method *callback* becomes a private property of child.
  2. Child *click* event handler for *Roll dice* runs parent *callback*.
     - Parent *callback* is run with the new dice result as parmeter.
     - Parent *callback* is run by the child *click* event handler for *Roll dice*.
  3. Parent method *callback* updates the parent property for result.

- See Eclipse Dice component with calls from parent.

HTML tags are global entities

- Each JS class can only be used once to define a custom HTML tag.
- Each custom HTML tag can only be defined once.
  - Applies even if created and used by independent JS files or modules

JavaScript modules for custom tags

Always use JavaScript modules for custom HTML tags.

Define the HTML class and its custom tag in the same module.

Examples of illegal code for defining custom tags

- Using the same class more than once:

```
customElements.define('my-tag', MyTag);

// Below will fail, as class MyTag already defines a tag
customElements.define('other-tag', MyTag);
```

- Defining the same tag more than once:

```
customElements.define('my-tag', FirstClass);

// Below will fail, as tag my-tag is already defined
// Code will fail even if done in a different module as
// HTML tags are global entities
customElements.define('my-tag', SecondClass);
```

Custom Element Best Practices

- See e.g. the article Custom Element Best Practices.