Manage HTML DOM with Vanilla JavaScript

Select children of an element

Get the **children nodes** of an **element**:

```
const childNodes = ele.childNodes;
```

By looping over **chidren**, we can get the **first** or **last child**:

```
const first = childNodes[0];
const last = childNodes[childNodes.length - 1];
```

There are properties to access the **first** and **last child** directly:

```
const first = ele.firstChild;
const last = ele.lastChild;
```

Select an element or list of elements

Select an element by given ID:

```
<div id="hello" />;
```

```
document.getElementById('hello');
```

Select elements by class name.

Returns the **list of elements** that have **hello** class within a given **element**:

```
ele.getElementsByClassName('hello');
```

Select an element or list of elements

Select elements by tag name.

Returns the list of **span elements** inside a given **element**:

```
ele.getElementsByTagName('span');
```

Select elements by CSS selector.

Returns the **list of elements** that match a given **selector**:

```
ele.querySelectorAll('div.hello');
```

Returns the **first element** that match a given **selector**:

```
ele.querySelector('div.hello');
```

Set CSS style for an element

Set a CSS style.

Setting the **style** via the **style property**:

```
ele.style.backgroundColor = 'red';
ele.style['backgroundColor'] = 'red';
ele.style['background-color'] = 'red';
```

Multiple styles can be set at same time by overwriting
or updating the **cssText property**:

```
// Add new style
el.style.cssText += 'background-color: red; color: white';

// Ignore previous styles
el.style.cssText = 'background-color: red; color: white';
```

Set CSS style for an element

Remove a CSS style.

```
ele.style.removeProperty('background-color');

// Does NOT work
ele.style.removeProperty('backgroundColor');
```

Show or hide an element

Show an element.

```
ele.style.display = '';
```

Hide an element.

```
ele.style.display = 'none';
```

Wrap an element around a given element

Wrap the **wrapper** around an **element**:

```
// First, insert wrapper before  the element in its parent node
ele.parentNode.insertBefore(wrapper, ele);

// And then, turn the element into a children of  wrapper
wrapper.appendChild(ele);
```

Unwrap an element

Remove an **element** except its **children**:

```
// Get the parent node
const parent = ele.parentNode;

// Move all children node to the parent
while (ele.firstChild) {
  parent.insertBefore(ele.firstChild, ele);
}

// The element becomes an empty element
// Remove it from the parent
parent.removeChild(ele);
```

Add or remove class from an element

Add a class to an element

```
ele.classList.add('class-name');
ele.classList.add('another', 'class', 'name');
```

Remove a class from an element

```
ele.classList.remove('class-name');
ele.classList.remove('another', 'class', 'name');
```

Toggle a class

```
ele.classList.toggle('class-name');
```

Check an element against a selector

We want to find out if the **child element** is a **descendant** of the **parent element**.

1. Use the contains method

```
const isDescendant = parent.contains(child);
```

2. Go up from the child until see the parent

```
// Check if child is a descendant of parent
const isDescendant = function (parent, child) {
  let node = child.parentNode;
  while (node) {
    if (node === parent) return true;
    node = node.parentNode; // Traverse up to the parent
  }
  // Go up until the root but couldn't find the `parent`
  return false;
};
```

Check if an element has given class

```
ele.classList.contains('class-name');
```

Create an element

Create new element

```
const ele = document.createElement('div');
```

Create new text node

```
const ele = document.createTextNode('Hello World!');
```

Determine height and width of an element

```
// Get the styles
const styles = window.getComputedStyle(ele);

// The size without padding and border
const height = ele.clientHeight -
  parseFloat(styles.paddingTop) - parseFloat(styles.paddingBottom);
const width = ele.clientWidth -
  parseFloat(styles.paddingLeft) - parseFloat(styles.paddingRight);

// The size include padding
const clientHeight = ele.clientHeight;
const clientWidth = ele.clientWidth;
```

Determine height and width of an element

```
// The size include padding and border
const offsetHeight = ele.offsetHeight;
const offsetWidth = ele.offsetWidth;

// The size include padding, border and margin
const heightWithMargin = ele.offsetHeight +
  parseFloat(styles.marginTop) + parseFloat(styles.marginBottom);
const widthWithMargin = ele.offsetWidth +
  parseFloat(styles.marginLeft) + parseFloat(styles.marginRight);
```

Get CSS styles of an element

We can get all **CSS styles** via the **getComputedStyle** method:

```
const styles = window.getComputedStyle(ele, null);
```

From there, it's easy to access the value of **specific style**:

```
// Get the background color
const bgColor = styles.backgroundColor;
```

Get CSS styles of an element

The **getPropertyValue** method produces the same result:

```
const bgColor = styles.getPropertyValue('background-color');

// Or turn the parameter to camelCase format:
const bgColor = styles.getPropertyValue('backgroundColor');
```

Get siblings of an element

Get the previous sibling

```
const prev = ele.previousSibling;
```

Get the next sibling

```
const next = ele.nextSibling;
```

Get all siblings

```
// Get the parent node
const parent = ele.parentNode;

// Filter the children, exclude the element
const siblings = [].slice.call(parent.children).filter(function (child) {
  return child !== ele;
});
```

Get, set and remove data attributes

Get the data attribute's value

```
// Get the data-message attribute of the element
const message = ele.getAttribute('data-message'); // option 1
const message = ele.dataset.message; // option 2
```

Set the data attribute's value

```
ele.setAttribute('data-message', 'Hello World'); // option 1
ele.dataset.message = 'Hello World'; // option 2
```

Remove the data attribute

```
ele.removeAttribute('data-message'); // option 1
delete ele.dataset.message; // option 2
```

Note that calling delete **ele.dataset** doesn't remove all **data attributes**.

Get or set the HTML of an element

Get the HTML

```
const html = ele.innerHTML;
```

Set the HTML

```
ele.innerHTML = '<h1>Hello World!</h1>';
```

Get position of an element relative to document

```
// Get the top, left coordinates of the element
const rect = ele.getBoundingClientRect();

// Add the scroll postion to get the full distance from the element
// to the top, left sides of the document
const top = rect.top + document.body.scrollTop;
const left = rect.left + document.body.scrollLeft;
```

Get text content of an element

Returns the **raw text content** of an **element** and its **children**.
All the **HTML tags** are excluded.

```
const text = ele.textContent;
```

Get size of an image

Image is already loaded

```
const image = document.querySelector(...);

// Get the original size
const naturalWidth = image.naturalWidth;
const naturalHeight = image.naturalHeight;

// Get the scaled size
const width = image.width;
const height = image.height;
```

Get size of an image

Listen on the **load event** to calculate the **size** of **image** which can be loaded via a given **URL**:

```
const image = document.createElement('img');
image.addEventListener('load', function (e) {
  // Get the size
  const width = e.target.width;
  const height = e.target.height;
});

// Set the source
image.src = '/path/to/image.png';
```

Get size of an image

We can use a **Promise** to turn the snippet to a **reusable function**:

```
const calculateSize = function (url) {
  return new Promise(function (resolve, reject) {
    const image = document.createElement('img');
    image.addEventListener('load', function (e) {
      resolve({
        width: e.target.width,
        height: e.target.height,
      });
    });
    image.addEventListener('error', function () {
      reject();
    })
    image.src = url;
  });
};
```

Get size of an image

```
calculateSize('/path/to/image.png').then(function (data) {
  const width = data.width;
  const height = data.height;
});
```

Redirect to another page

Redirect to another page

```
location.href = '/the/new/url';
```

Go back to the previous page

```
history.back(); // option 1
history.go(-1); // option 2
```

Insert an element after or before other element

Insert after

Insert an **element** after the **refEle element**:

```
refEle.parentNode.insertBefore(ele, refEle.nextSibling); // option 1
refEle.insertAdjacentElement('afterend', ele); // option 2
```

Insert before

Insert an **element** before the **refEle element**:

```
refEle.parentNode.insertBefore(ele, refEle); // option 1
refEle.insertAdjacentElement('beforebegin', ele); // option 2
```

Insert given HTML after or before an element

Insert after

Insert **HTML** after an **element**:

```
ele.insertAdjacentHTML('afterend', html);
```

Insert before

Insert **HTML** before an **element**:

```
ele.insertAdjacentHTML('beforebegin', html);
```

Prepend to an element

Add an **element** to the beginning of the **target element**:

```
target.insertBefore(ele, target.firstChild);
```

Remove all children of a node

1. Empty the inner HTML (not recommended)

```
ele.innerHTML = '';
```

This method isn't recommended because it doesn't remove **event handlers** of **child node**. Hence, it might cause a **memory leak** if we are managing a **big list of elements**.

2. Remove child nodes

Remove its **child node** until it doesn't have any **children**.

```
while (node.firstChild) {
  node.removeChild(node.firstChild);
}
```

Replace broken images

Replace the **broken images** with an **image** telling visitors that they are not found:

```
// Assume that wewant to replace all images on the page
const images = document.querySelectorAll('img');

// Loop over them
[].forEach.call(images, function (ele) {
  ele.addEventListener('error', function (e) {
    e.target.src = '/path/to/404/image.png';
  });
});
```

Replace an element

The **element** will be removed from the **DOM tree**,
and is replaced with the **new element** :

```
ele.parentNode.replaceChild(newEle, ele);
```

Append to an element

Append an **element** to the end of the **target element**:

```
target.appendChild(ele);
```

Get parent node of an element

Returns the **parent node** of the an **element**:

```
const parent = ele.parentNode;
```

Loop over a nodelist

Assume that **elements** is a **NodeList** that matches given **selector**:

```
const elements = document.querySelectorAll(...);
```

Then we can loop over **elements** by using one of these approaches:

1. Use the ES6 spread operator

```
[...elements].forEach(function(ele) {
   ...
});
```

Loop over a nodelist

2. Use the **Array** methods

```
// option 1
Array.from(elements).forEach(function(ele) {
  ...
});

// option 2
[].forEach.call(elements, function(ele) {
  ...
});

// option 3
[].slice.call(elements, 0).forEach(function(ele) {
  ...
});
```

Loop over a nodelist

3. Use the **forEach** method

```
elements.forEach(function(ele) {
  ...
});
```

Insert an element after or before other element

Insert after:

Insert an **element** after other **element** (**refEle**).

refEle.parentNode.insertBefore(ele, refEle.nextSibling); ***// option 1***

refEle.insertAdjacentElement('afterend', ele); ***// option 2***

Insert before:

Insert an **element** before other **element** (**refEle**).

refEle.parentNode.insertBefore(ele, refEle); ***// option 1***

refEle.insertAdjacentElement('beforebegin', ele); ***// option 2***

Remove an element

1. Use the **remove** method

```
ele.remove();
```

2. Use the **removeChild** method

```
if (ele.parentNode) {
  ele.parentNode.removeChild(ele);
}
```

Clone an element

```
const cloned = ele.cloneNode(true);
```

Using **cloneNode(true)** method will **deep copy** a given **element**.

In this code, all **attributes** and **children node** of **original node** (**ele**) will be cloned in **cloned node** as well.

Passing **false** produces a **cloned node** that keeps only **attributes** and the **original node**:

```
const cloned = ele.cloneNode(false);
```

Get, set and remove attributes

Get the attribute's value

```
// Get the `title` attribute of a link element
const title = link.getAttribute('title');
```

Set the attribute's value

```
// Set the width and height of an image
image.setAttribute('width', '100px');
image.setAttribute('height', '120px');
```

Remove the attribute

```
// Remove the `title` attribute
ele.removeAttribute('title');
```

Get closest element by given selector

1. Use the native **closest()** method

```
const result = ele.closest(selector);
```

2. Traverse up until find the matching element

```
const matches = function (ele, selector) {
  return (
    ele.matches ||
    ele.matchesSelector ||
    ele.msMatchesSelector ||
    ele.mozMatchesSelector ||
    ele.webkitMatchesSelector ||
    ele.oMatchesSelector
  ).call(ele, selector);
};
...
```

Get closest element by given selector

```
...

// Find the closest element to `ele` and matches the `selector`
const closest = function (ele, selector) {
  let e = ele;
  while (e) {
    if (matches(e, selector)) {
      break;
    }
    e = e.parentNode;
  }
  return e;
};
```

Check if an element is a descendant of another

Assume that we want to find out if the **child element** is a **descendant** of the **parent element**.

1. Use the contains method

```
const isDescendant = parent.contains(child);
```

Check if an element is a descendant of another

2. Go up from the child until see the parent

```
// Check if `child` is a descendant of `parent`
const isDescendant = function (parent, child) {
  let node = child.parentNode;
  while (node) {
    if (node === parent) {
      return true;
    }

    // Traverse up to the parent
    node = node.parentNode;
  }

  // Go up until the root but couldn't find the `parent`
  return false;
};
```

Toggle password visibility

Assume that we have two **elements**: a **password element**, and a **button** for toggling the **visibility** of the **password**:

```
<input type="password" id="password" />
<button id="toggle">Toggle</button>
```

Toggle password visibility

In order to show the **password**, we turn the **password element** to an usual **textbox** whose **type attribute** is **text**:

```
// Query the elements
const passwordEle = document.getElementById('password');
const toggleEle = document.getElementById('toggle');

toggleEle.addEventListener('click', function () {
  const type = passwordEle.getAttribute('type');

  passwordEle.setAttribute(
    'type',
    // Switch it to a text field if it's a password field
    // currently, and vice versa
    type === 'password' ? 'text' : 'password'
  );
});
```

Count number of characters of a textarea

Assume that we have a **textarea element** and a **div element** for showing how many **characters** user has been entering:

```
<textarea id="message"></textarea>
<div id="counter"></div>
```

Use the maxlength attribute

The **maxlength attribute** sets maximum number of **characters** that user can put in the **textarea**.

```
<textarea maxlength="200" id="message"></textarea>
```

Count number of characters of a textarea

Count the number of characters

Handle the **input event** which is triggered if the value of **element** is changed:

```
const messageEle = document.getElementById('message');
const counterEle = document.getElementById('counter');

messageEle.addEventListener('input', function (e) {
  const target = e.target;

  // Get the `maxlength` attribute
  const maxLength = target.getAttribute('maxlength');
  // Count the current number of characters
  const currentLength = target.value.length;

  counterEle.innerHTML = `${currentLength}/${maxLength}`;
});
```

Detect if an element is focused

Assume that **ele** represents the **element** that we want to check if it has the **focus** currently:

```
const hasFocus = ele === document.activeElement;
```

Get or set document title

Get the document title

```
const title = document.title;
```

Set the document title

```
document.title = 'Hello World';
```

Get document height and width

Get the document height

```
// Full height, including the scroll part
const fullHeight = Math.max(
  document.body.scrollHeight,
  document.documentElement.scrollHeight,
  document.body.offsetHeight,
  document.documentElement.offsetHeight,
  document.body.clientHeight,
  document.documentElement.clientHeight
);
```

Get document height and width

Get the document width

```
// Full width, including the scroll part
const fullWidth = Math.max(
  document.body.scrollWidth,
  document.documentElement.scrollWidth,
  document.body.offsetWidth,
  document.documentElement.offsetWidth,
  document.body.clientWidth,
  document.documentElement.clientWidth
);
```

Go back to previous page

```
history.back(); // option 1

history.go(-1); // option 2
```

Trigger an event

Trigger event for inputs

There are some **special events** that are avaialble as **method's element**.
We can call them directly.

```
// For text box and textarea
ele.focus();
ele.blur();

// For form element
formEle.reset();
formEle.submit();

// For any element
ele.click();
```

Trigger an event

Trigger a native event

```
const trigger = function (ele, eventName) {
  const e = document.createEvent('HTMLEvents');
  e.initEvent(eventName, true, false);
  ele.dispatchEvent(e);
};
```

We can also **trigger** the **change**, **keyup**, **mousedown** and more.

```
trigger(ele, 'mousedown');
```

Trigger an event

Trigger a custom event

We can **trigger** a **custom event** named **hello** with a **data** of **{ message: 'Hello World' }**:

```
const e = document.createEvent('CustomEvent');
e.initCustomEvent('hello', true, true, { message: 'Hello World' });

// Trigger the event
ele.dispatchEvent(e);
```

Attach or detach an event handler

Use the on attribute (not recommended)

We can set an **event handler** via **on{eventName} attribute**, where **eventName** represents the **name** of **event**.

```
ele.onclick = function() {
  ...
};

// Remove the event handler
delete ele.onclick;
```

This approach isn't recommended because we can only attach one **handler** for **each event**. Setting the **onclick attribute**, for example, will override any existing **handler** for the **click event**.

Attach or detach an event handler

Use the **addEventListener** method

```
const handler = function() {
  ...
};

// Attach handler to the click event
ele.addEventListener('click', handler);

// Detach the handler from the click event
ele.removeEventListener('click', handler);
```

Note that the **event name** is passed as the first parameter in both the **addEventListener** and **removeEventListener** methods. It differs from the first approach which requires to **prefix** the **event name** with **on**.

Create one time event handler

1. Use the once option

When attach a **handler** to given **event**, we can pass **{ once: true }** to the last parameter of the **addEventListener** method:

```
const handler = function (e) {
   // The event handler
};

ele.addEventListener('event-name', handler, { once: true });
```

Create one time event handler

2. Self-remove the handler

```
const handler = function (e) {
  // The event handler
  // Do something ...

  // Remove the handler
  e.target.removeEventListener(e.type, handler);
};

ele.addEventListener('event-name', handler);
```

Prevent default action of an event

1. Return **false** for the **on<event>**

```
ele.onclick = function(e) {
   // Do some thing
   ...

   return false;
};
```

It's same if we the **inline attribute**:

```
<form>
   <button type="submit" onclick="return false">Click</button>
</form>
```

This approach isn't recommend because returning **false** just doesn't make sense and it doesn't work with the **addEventListener()** method.

Prevent default action of an event

2. Use the **preventDefault()** method

This method works with **inline attribute**

```
<button type="submit" onclick="event.preventDefault()">Click</button>
```

To **event handlers**:

```
ele.onclick = function(e) {
  e.preventDefault();
  // Do some thing
};

ele.addEventListener('click', function(e) {
  e.preventDefault();
  // Do some thing
});
```

Execute code when document is ready

```
const ready = function (cb) {
 // Check if the `document` is loaded completely
 document.readyState === "loading"
  ? document.addEventListener("DOMContentLoaded", function (e) {
    cb();
   })
  : cb();
};

// Usage
ready(function() {
  // Do something when the document is ready
  ...
});
```

Detect clicks outside of an element

Check if a **click** was outside of an **element**:

```
document.addEventListener('click', function (evt) {
  const isClickedOutside = !ele.contains(evt.target);

  // `isClickedOutside` is true if the clicked target is outside of `ele`
});
```

Submit a form with Ajax

```
const submit = function (formEle) {
  return new Promise(function (resolve, reject) {
    const params = serialize(formEle); // Serialize form data
    // Create Ajax request
    const req = new XMLHttpRequest();
    req.open('POST', formEle.action, true);
    req.setRequestHeader('Content-Type',
          'application/x-www-form-urlencoded; charset=UTF-8');
    // Handle the events
    req.onload = function () {
      if (req.status >= 200 && req.status < 400) resolve(req.responseText);
    };
    req.onerror = function () {
      reject();
    };
    req.send(params);
  });
};
```

Submit a form with Ajax

The **serialize** function serializes all the **form data** into a **query string**.

```
const formEle = document.getElementById(...);

// response is what we got from the back-end
submit(formEle).then(function(response) {
  // We can parse it if the server returns a JSON
  const data = JSON.parse(response);
  ...
});
```

Upload files with Ajax

This function sends **selected files** from a **file input element** to a **back-end**:

```
const upload = function (fileEle, backendUrl) {
  return new Promise(function (resolve, reject) {
    // Get the list of selected files
    const files = fileEle.files;
    // Create a new FormData
    const formData = new FormData();

    // Loop over the files
    [].forEach.call(files, function (file) {
      formData.append(fileEle.name, file, file.name);
    });
    ...
  });
};
```

Upload files with Ajax

```
const upload = function (fileEle, backendUrl) {
  return new Promise(function (resolve, reject) {
    ...
    // Create new Ajax request
    const req = new XMLHttpRequest();
    req.open('POST', backendUrl, true);
    // Handle the events
    req.onload = function () {
      if (req.status >= 200 && req.status < 400) {
        resolve(req.responseText);
      }
    };
    req.onerror = function () {
      reject();
    };
    req.send(formData);
  });
};
```

Upload files with Ajax

Assume that we have a **file input element** that allows user to choose **multiple files**:

```
<input type="file" id="upload" multiple />
```

We can use this code inside a **click event handler** of a **button** which performs the **uploading**:

```
const fileEle = document.getElementById('upload');

upload(fileEle, '/path/to/back-end').then(function(response) {
   // `response` is what we got from the back-end
   // We can parse it if the server returns a JSON
   const data = JSON.parse(response);
   ...
});
```

Get size of selected file

In this markup, we have two **elements** defined by different **id attributes**.

The **id="size" element** will be used to display the **size** of **selected file** from the **id="upload" element**.

```
<input type="file" id="upload" />
<div id="size"></div>
```

We listen on the **change event** of the **file input**, and get the **selected files** via **e.target.files**.

The **file size** in bytes of the **selected file** can be retrieved from the **size property** of the first (and only) file.

Get size of selected file

The **size element** is shown up or hidden based on the fact that user selects a file or not.

```
// Query the elements
const fileEle = document.getElementById('upload');
const sizeEle = document.getElementById('size');

fileEle.addEventListener('change', function (e) {
  const files = e.target.files;
  if (files.length === 0) {
    // Hide the size element if user doesn't choose any file
    sizeEle.innerHTML = '';
    sizeEle.style.display = 'none';
  } else {
    sizeEle.innerHTML = `${files[0].size} B`; // File size in bytes
    sizeEle.style.display = 'block'; // Display it
  }
});
```

Get size of selected file

Display a readable size

Instead of displaying in **bytes**, we can transform it to a **readable format** in **kB**, **MB**, **GB**, and **TB** depending on how big it is.

The **formatFileSize** helper method is created for that purpose:

```
// Convert the file size to a readable format
const formatFileSize = function (bytes) {
  const sufixes = ['B', 'kB', 'MB', 'GB', 'TB'];
  const i = Math.floor(Math.log(bytes) / Math.log(1024));
  return `${(bytes / Math.pow(1024, i)).toFixed(2)} ${sufixes[i]}`;
};

// Display the file size
sizeEle.innerHTML = formatFileSize(files[0].size);
```

Preview an image before uploading it

This is the markup for a **file input** which allows to choose an **image** using an **img element** for **previewing** the **selected file**.

```
<input type="file" id="fileInput" />

<img id="preview" />
```

Both **elements** can be taken by the **getElementById()** method:

```
const fileEle = document.getElementById('fileInput');
const previewEle = document.getElementById('preview');
```

Preview an image before uploading it

1. Use the URL.createObjectURL() method

```
fileEle.addEventListener('change', function (e) {
  // Get the selected file
  const file = e.target.files[0];

  // Create a new URL that references to the file
  const url = URL.createObjectURL(file);

  // Set the source for preview element
  previewEle.src = url;
});
```

Preview an image before uploading it

2. Use the **FileReader**'s **readAsDataURL()** method

```
fileEle.addEventListener('change', function (e) {
  // Get the selected file
  const file = e.target.files[0];

  const reader = new FileReader();
  reader.addEventListener('load', function () {
    // Set the source for preview element
    previewEle.src = reader.result;
  });

  reader.readAsDataURL(file);
});
```

Resize an image

Assume that we want to **resize** an **image** to a given **number of percentages**.

This **image** can be determined from a **file input**:

```
// A file input
<input type="file" id="upload" />;

// Get the selected file
const image = document.getElementById('upload').files[0];
```

Resize an image

The following function **scales** an **image file** to **ratio of percentages**:

```
const resize = function (image, ratio) {
  return new Promise(function (resolve, reject) {
    const reader = new FileReader();
    reader.readAsDataURL(image); // Read the file

    // Manage the `load` event
    reader.addEventListener('load', function (e) {
      const ele = new Image(); // Create new image element
      ele.addEventListener('load', function () {
        const canvas = document.createElement('canvas'); // Create new canvas
        ...
      });
      ...
    });
    ...
  });
};
```

Resize an image

```
const resize = function (image, ratio) {
  return new Promise(function (resolve, reject) {
    ...
    reader.addEventListener('load', function (e) {
      ...
      ele.addEventListener('load', function () {
        ...
        // Draw the image that is scaled to `ratio`
        const context = canvas.getContext('2d');
        const w = ele.width * ratio;
        const h = ele.height * ratio;
        canvas.width = w;
        canvas.height = h;
        context.drawImage(ele, 0, 0, w, h);
        ...
      });
      ...
};
```

Resize an image

```
    ...
    reader.addEventListener('load', function (e) {
      ...
      ele.addEventListener('load', function () {
        ...
        // Get the data of resized image
        'toBlob' in canvas
          ? canvas.toBlob(function (blob) {
              resolve(blob);
           })
          : resolve(dataUrlToBlob(canvas.toDataURL()));
      });
      ele.src = e.target.result; // Set the source
    });
    reader.addEventListener('error', function (e) {
      reject();
    });
  });
};
```