

Пишем собственный WYSIWYG редактор на основе веб-компонентов и textarea. Часть 1

Вступление



Всем привет, последние пару месяцев я активно изучаю тему веб-компонентов, собираю и нарабатываю опыт, а затем делюсь своими наработками с другими с целью обменяться опытом, получить новый опыт, фидбек и понять куда движется разработка в вебе и шагать дальше за новым опытом. Все ниже изложенное не является инструкцией как делать нужно, а является примером

того, как сделать возможно на текущий момент в 2023 году, у меня уже набрался небольшой опыт (8 публикаций и 3 веб-компонента на гитхабе) и я решился попробовать сделать что-то серьезнее чем просто очередную реактивную кнопку или лайки, в первой части моей публикации я проведу вас по MVP веб-компонента `ws-wisuwig`, немного затронем философию семантики, браузерные API и обменяемся опытом, потестим HTML5 теги в статье на хабре. Для нетерпеливых сразу вот ссылка на демо и git репозиторий. Остальных ждет технический лонгрид, прошу под кат)

Техническая основа и база редактора

В базовой функциональности редактора, важно предусмотреть фундамент для будущего развития веб-компонента, а также реализовать работу с API основных возможностей которые дают нам браузеры, но также важно знать меру и не переусердствовать, в качестве базы мы могли бы взять некий bootstrap или tailwind для стилей, а для формочек некий react/vue чтобы не морочиться с биндингом данных, а еще затащить иконочный шрифт чтобы не морочиться с иконками, но тогда весь фундаментальный смысл расширяемости просто бы пропал, зато появилась необходимость поддерживать версии библиотек в `node_modules`, сегодняшний пост совсем не об этом, мы будем писать на TypeScript используя ESNext стиль и вообще не будем использовать полифилы. Но все-таки чтобы не писать много лапши и получить код с хорошей читаемостью и оформлением, я воспользуюсь самодельной функцией `e1` которая просто будет выполнять действия над возвращаемым `Element` из функции `document.createElement`

В каком-то смысле можно сказать, что веб-компонент `ws-wisuwig` написан на функциональных компонентах основанных на браузерном DOM, в модном ныне SSR этому компоненту делать нечего, он просто добавляет возможностей к редактированию текста внутри `textarea` на клиенте.

```
/**
 * Short document.createElement
 * @param tagName element tag name
 * @param params list of object params for document.createElements
 * @returns
 */
export const e1 = (tagName: keyof HTMLElementTagNameMap | string, {classList,
```

```
styles, props, attrs, options, append}:{
  classList?: string[],
  styles?: object,
  props?: object,
  attrs?: object,
  options?: {
    is?:string
  },
  append?: Element[]
} = {}):any => {
  if(!tagName) {
    throw new Error(`Undefined tag ${tagName}`);
  }
  const element = document.createElement(tagName, options);
  // element.classList
  if(classList) {
    for (let i = 0; i < classList.length; i++) {
      const styleClass = classList[i];
      if(styleClass) {
        element.classList.add(styleClass)
      }
    }
  }
  // element.style[prop]
  if(styles) {
    const stylesKeys = Object.keys(styles);
    for (let i = 0; i < stylesKeys.length; i++) {
      const key = stylesKeys[i];
      element.style[key] = styles[key];
    }
  }
  // element[prop]
  if(props) {
    const propKeys = Object.keys(props);
    for (let i = 0; i < propKeys.length; i++) {
      const key = propKeys[i];
      element[key] = props[key];
    }
  }
  // element.setAttribute(key,val)
  if(attrs) {
    const attrsKeys = Object.keys(attrs);
    for (let i = 0; i < attrsKeys.length; i++) {
      const key = attrsKeys[i];
      if(attrs[key]) {
        element.setAttribute(key, attrs[key]);
      }
    }
  }
  if(append) {
```

```
    for (let i = 0; i < append.length; i++) {  
      const appendEl = append[i];  
      element.append(appendEl);  
    }  
  }  
  return element;  
};
```

Функция сама по себе проста насколько это возможно и от себя ничего не добавляет, создана исключительно для удобства, вы можете найти похожие функции в Vue по имени `h` или в React увидите похожий синтаксис в документации раздела `Elements`. Данная функция родилась в процессе написания этого компонента из-за острой необходимости быстро и просто и удобно что-то делать с элементами DOM дерева, я не копировал и не переделывал функции из фреймворков, так сказать вдохновился на опыте использования.

Также в базе у нас будет 2 файла со стилями в одном файле будут стили для самого редактора, а во втором файле будут базовые стили для тегов. Сами стили написаны с использованием SASS, но в репозитории также доступна и CSS версия, все цвета прописаны через переменные, цветовая палитра взята [отсюда](#).

Базовые функции редактора

Редактор в качестве основы будет поддерживать семантику HTML5 доступных нам тегов, а это значит что, бы стоило начать с тегов. Что мы знаем о HTML5 тегах в общих чертах?

- Теги могут быть одиночные и с закрывающим тегом `<hr>` или `<span>строка</span>`
- Фундаментально поведение тега в верстке определяется его `position` и `display` CSS свойствами
- Теги имеющие закрывающий тег не обязательно имеют текстовый контент внутри, например: `figure`, `audio`, `video`
- Часть тегов изначально визуально выглядит одинаково `var`, `b`, `strong` или вообще никак не выделяется на фоне текста `span`, `abbr`, `dfn`
- Часть тегов теряет смысл и семантику без своих обязательных атрибутов `a`, `abbr`, `dfn`, `time`

Из этих знаний мы можем вывести условно, что у нас существуют блочные и строчные элементы с которыми мы хотим иметь 3 базовых действия в редакторе:

- Вставлять тег и убирать его удалив или убрав форматирование у текста.
- Оборачивать существующий текст в тег, по аналогии, как мы привыкли это видеть в текстовых редакторах.
- Управлять не только текстом и тегом, но и атрибутами (иногда `properties`) тега, чтобы получить больший контроль над редактируемым текстом.

В базе, на мой взгляд, это все, что должен уметь текстовый редактор. Дополнительные функции типа: раскрашивания элементов в любые цвета, установку колонтитулов для страниц и вообще работа с текстом постранично, а также работа с таблицами, графиками, различные `drag and drop` элементы — все это не относится к идее текстового HTML5 WYSIWYG редактора, или относится косвенно в виде дополнительных возможностей, мы же начнем с азов и редактирования текста и постараемся вообще не вмешиваться в редактируемый DOM

контента, чтобы не портить пользовательский UX и дать работать с чистым HTML, что например уже нельзя в навороченном новом редакторе хабра и текст мне для статьи пришлось переносить поблочно из уже частично готово HTML5

```
▼<div class="node node_paragraph">
  ▶<div data-drag-handle contenteditable="false" class="node__drag-control">...</div>
  ▶<div contenteditable="false" class="right-menu__container">...</div>
  ▼<div class="node__inner">
    ▶<p>...</p>
    <p></p> == $0
  </div>
</div>
```

Реализуем вставку тегов

```
const allTags = [
  { tag: 'h1' },
  { tag: 'h2' },
  { tag: 'h3' },
  { tag: 'h4' },
  { tag: 'h5' },
  { tag: 'h6' },
  { tag: 'span' },
  { tag: 'mark' },
  { tag: 'small' },
  { tag: 'dfn' },
  { tag: 'a' },
  { tag: 'q' },
  { tag: 'b' },
  { tag: 'i' },
  { tag: 'u' },
  { tag: 's' },
  { tag: 'sup' },
  { tag: 'sub' },
  { tag: 'kbd' },
  { tag: 'abbr' },
  { tag: 'strong' },
  { tag: 'code' },
  { tag: 'samp' },
  { tag: 'del' },
  { tag: 'ins' },
  { tag: 'var' },
  { tag: 'ul' },
  { tag: 'ol' },
  { tag: 'hr' },
  { tag: 'pre' },
  { tag: 'time' },
  { tag: 'img' },
```

```

    { tag: 'audio'},
    { tag: 'video'},
    { tag: 'blockquote'},
    { tag: 'details'},
  ] as WCWYSIWYGTag[];

```

Если вам, как и мне хочется этот листинг превратить в простой массив, то обратите внимание на тип **WCWYSIWYGTag** в котором я заложил еще **hint, is, method** которые пригодятся позже чтобы реализовать в веб-компоненте поддержку других веб-компонентов)

Внимательный читатель, может заметить, что тут не хватает нескольких тегов, например **iframe, object, script, ruby**, отсутствует самый популярный тег **div** и с ним **section, main, footer** и еще несколько, в целом ничего не мешает их добавить в тот список, но эти теги не являются частью текстового редактора, если размышлять семантически, в редакторе мы редактируем некий **article** в котором семантически может быть **footer, header, aside**, но с точки зрения текста они роли не сыграют. Возможно в будущих версиях 1+ этого веб-компонента я добавлю какие-то стили и поддержку этих тегов в виде кнопок, а пока их можно разместить только переключившись в текстовый режим редактора.

Разобравшись со всеми тегами осталось дать пользователю выбирать их через атрибут **data-allow-tags** и на основе переданного списка атрибутов строить интерфейс:

```

//Получаем теги из атрибута если есть
const allowTags = this.getAttribute('data-allow-tags') || allTags.map(t =>
t.tag).join(',');
//...
//Собираем теги в массив
this.EditorAllowTags = allowTags.split(',');
//Формируем итоговый WCWYSIWYGTag[]
this.EditorTags = allTags.filter(tag => allowTags.includes(tag.tag));

```

И осталось описать функцию, которая соберет нам кнопки, тк собирать кнопки нам придется еще не 1 раз, сделаем два аргумента для функции, 1 элемент в который собираем кнопки и 2 набор кнопок (тегов), благодаря функции `el` код выглядит очень просто:

```

#makeActionButtons(toEl:HTMLElement, actions:WCWYSIWYGTag[]) {
  for (let i = 0; i < actions.length; i++) {
    const action = actions[i];
    const button = el('button', {
      classList: ['wc-wysiwyg_btn', `-${action.tag}`],
      props: {
        tabIndex: -1,
        type: 'button',
        textContent: action.is ? `${action.tag} is=${action.is}` :
action.tag,
        onpointerup: (event) => this.#tag(action.tag, event,
action.is),
      },
      attrs: {
        'data-hint': action.hint ? action.hint : this.#t(action.tag)
|| '-',

```

```
    }
  });
  toEl.appendChild(button);
}
}
```

Функция достаточно проста, в цикле создаем кнопки и привязываем с помощью стрелочных функций и `onpointerup` действия к ним. Абстрактно, мы всегда будем вызывать действие `#tag` а уже внутри этого метода разбираться, что будем делать с этим тегом. Рассмотрим функцию `#tag`

```
#tag = (tag:WCWYSIWYGTag) => {
  switch (tag.tag) {
    case 'audio':
      this.#Media('audio');
      break;
    case 'video':
      this.#Media('video');
      break;
    case 'details':
      this.#Details();
    case 'img':
      this.#Image();
      break;
    default:
      if(typeof tag.method === 'function') {
        tag.method.apply(this, tag);
      } else {
        this.#wrapTag(tag, tag.is);
      }
      break;
  }
}
```

Тоже все очень просто, мы перебираем доступные варианты действия над тегом, мы можем его или обернуть с поправкой на тег или вставить тег самостоятельно с поправкой на особенности тега (или `custom-element`), на весь набор тегов выходит 4 метода для **Audio\Video, img и details**, в остальном мы можем просто создать тег и обернуть текст в него или если доступен собственный метод у тега, выполнить его. Рассмотрим обработку блочного элемента на примере Audio/Video.

```
#Media = (tagName:string) => {
  const mediaSrc = prompt('src', '');
  if(mediaSrc === '') {
    return false;
  }
  const mediaEl = el(tagName, { attrs: { controls: true }, props: { src: mediaSrc } });
  this.EditorNode.append(mediaEl);
  this.updateContent();
}
```

```
}
```

Т.к. минимализм наше все, в место модальных окон я буду использовать prompt чтобы не раздувать редактор очередным изобретением модального окна с одним полем ввода, хотя с el функцией это выглядело бы не так сложно.

А вот с методом #wrapTag все немного сложнее, но концептуально он похож на метод #Media, с несколькими исключениями:

```
#wrapTag = (tag, is:boolean|string = false) => {  
  //Обрабатываем случай, когда оборачивают в список, то текст будет в li а сверху добавим ol/ul  
  const listTag = ['ul', 'ol'].includes(tag) ? tag : false;  
  tag = listTag !== false ? 'li' : tag;  
  const Selection = window.getSelection();  
  let className = null;  
  //подготовим параметры по умолчанию для создания el  
  let defaultOptions = {  
    classList: className ? className : undefined,  
  } as any;  
  if(is) {  
    defaultOptions.options = {is};  
  }  
  let tagNode = el(tag, defaultOptions);  
  
  if (Selection !== null && Selection.rangeCount) {  
    if(listTag !== false) {  
      const list = el(listTag);  
      tagNode.replaceWith(list);  
      list.append(tagNode)  
    }  
    const range = Selection.getRangeAt(0).cloneRange();  
    range.surroundContents(tagNode);  
    Selection.removeAllRanges();  
    Selection.addRange(range);  
    //Если выделенного текста на странице нет, добавим имя тега  
    //чтобы пользователь не мучался с поданием урсором в пустой тег  
    if(Selection.toString().length === 0) {  
      tagNode.innerText = tag;  
    }  
    this.updateContent();  
  }  
}
```

Чтобы не добавлять отдельный метод для списков и поддерживать возможность обернуть тест в список и получить список из элемента который был выделен в тексте, обрабатываем это исключение прямо в этом методе.

Многие пользователи сначала нажимают на тег, а потом собираются туда что-то писать, но попасть курсором в пустой тег затруднительно по этому мы обрабатываем случай `Selection.toString().length === 0` и если текст не был выделен, добавим в новый тег

имя этого тега, чтобы было проще потом отредактировать содержимое тега.

Оборачивать в текст можно не только в простой тег, но и в custom-element так что добавим и поддержку is для автономных веб-компонентов, а для custom-elements просто обернем текст в этот тег, под оборачиванием в текст я имею в виду конструкцию `range.surroundContents(tagNode);`

Отлично! на этом этапе, мы уже имеем базовый функционал и можем вставлять теги в наш EditorNode и оборачивать в теги существующий текст, давайте сразу проработаем кнопку отмены вставки, тот случай, когда мы хотим снять с части текста обрамление каким-то тегом. Создадим наш ClearFormatButton

```
this.EditorClearFormatBtn = el('button', {
  classList: ['wc-wysiwyg_btn', '-clear'],
  attrs: {
    'data-hint': this.#t('clearFormat'),
  },
  props: {
    innerHTML: 'X',
  },
});
```

По умолчанию кнопка очистки формата не имеет собственного слушателя событий, ее работа будет зависеть от текущего выделенного тега в редакторе, добавим в нашу область редактирования EditorNode слушатель onpointerup, обработку события очистки формата, а также проверку возможности редактировать по выбранному элементу, в целом весь NodeEditor редактора в базовой версии будет выглядеть так:

```
//.... в connectedCallback()
this.EditorNode = el('article', {
  classList: ['wc-wysiwyg_content', this.getAttribute('data-content-class') || ''],
  props: {
    contentEditable: true,
    //Поведение при клике в области редактирования
    onpointerup: event => {
      this.checkCanClearElement(event);
      if(this.#EditProps) {
        this.checkEditProps(event);
      }
    },
    //Обновляем контент по input событию
    oninput: event => {
      this.updateContent();
      if(this.#Autocomplete) {
        this.#checkAutoComplete();
      }
    },
    //Проверяем сочетания клавиш нажатых в редакторе
    onkeydown: event => {
```

```
        this.#checkKeyBindings(event)
      },
    },
  });
```

Вернемся к нашей функции форматирования текста, мое повествование идет в порядке наращивания функционала, по этому мы рассматриваем код не в той очередности, в которой вы его видите в git репозитории.

```
#checkCanClearElement(event:Event) {
  const eventTarget = event.target as HTMLElement;
  if(eventTarget !== this.EditorNode) {
    if(eventTarget.nodeName !== 'P'
    && eventTarget.nodeName !== 'SPAN') {
      this.EditorClearFormatBtn.style.display = 'inline-block';
      this.EditorClearFormatBtn.innerHTML = `✕
${eventTarget.nodeName}`;
      this.EditorClearFormatBtn.onpointerup = (event) => {
eventTarget.replaceWith(document.createTextNode(eventTarget.textContent));
      }
      this.showEditorInlineDialog();
    } else {
      this.EditorClearFormatBtn.style.display = 'none';
      this.EditorClearFormatBtn.onpointerup = null;
    }
  }
}
```

В момент нажатия на элемент, мы проверяем что нажатие произошло не в P или SPAN это единственные два тега, которые мы не будем очищать, для остальных мы в кнопку очистки формата подставим текущий тег и добавим уже здесь слушатель события нажатия, сама очистка тега выглядит очень просто, мы меняем тег на `TextNode` и получаем просто текст `document.createTextNode(eventTarget.textContent)`. Из минусов такого решения можно выделить, что очистка формата происходит только над 1 тегом и пользователь не может очистить формат сразу нескольких тегов в глубину (`parentElements`). На этом этапе мы получили CRUD действия над тегами, их можно вставлять\оборачивать в тег и можно удалять, осталось проработать U — Update а именно, редактирование свойств тегов, ведь некоторые теги без атрибутов не имеют семантического смысла и ли теряют функциональность.

Редактирование атрибутов тегов

О том, в какой момент мы проверяем нажатие на тег мы уже проговорили, в этот же момент мы также проверяем можем ли мы редактировать атрибуты у тега. Для начала пробросим JSON строку вида `{a: [«href», «class», «target»]}` которая содержит объект, где ключом является имя тега, а значением массив строк в виде имен атрибутов, которые мы допускаем к редактированию в редакторе.

```
#checkEditProps(event) {
  const eventTarget = event.target as HTMLElement;
```

```
//Проверяем eventTarget доступен ли такой тег для редактирования
if(this.#EditProps[eventTarget.nodeName]) {
    const props = this.#EditProps[eventTarget.nodeName];
    event.stopPropagation();
    //Показываем форму редактирования пропсов и наш инлайн диалог
    this.EditorPropertyForm.style.display = '';
    this.showEditorInlineDialog();
    //создаем в цикле набор инпутов каждый из которых биндим на свой атрибут, не
забываем очистить форму перед этим
    this.EditorPropertyForm.setAttribute('data-tag',
eventTarget.nodeName);
    this.EditorPropertyForm.innerHTML = '';
    for (let i = 0; i < props.length; i++) {
        const tagProp = props[i];
        const isAttr = tagProp.indexOf('data-') > -1 || tagProp ===
'class';
        this.EditorPropertyForm.append(el('label', {
            props: { innerText: `${tagProp}=`,
            append: [
                //Сразу же добавим инпут с редактированием свойств
                el('input', {
                    attrs: { placeholder: tagProp },
                    classList: ['wc-wysiwyg_inp'],
                    props: {
                        value: isAttr ?
eventTarget.getAttribute(tagProp) : eventTarget[tagProp] || '',
                        oninput: (eventInput) => {
                            const eventInputTarget = eventInput.target
as HTMLInputElement;
                            //Чтобы пользователь мог вводить несколько классов одной строкой, будем
подставлять класс через className
                            if(tagProp === 'class') {
                                eventTarget.className =
eventInputTarget.value;
                            }
                            //Тут же обработаем исключение для datetime
                            if((isAttr || tagProp === 'datetime') &&
eventInputTarget !== null) {
                                eventTarget.setAttribute(tagProp,
eventInputTarget.value)
                            } else {
                                eventTarget[tagProp] =
eventInputTarget.value;
                            }
                            this.updateContent();
                        }
                    }
                })
            ]
        })
    }
}
```

```
        }));  
    }  
    //Добавляем кнопку отправки нашей формы для поддержания привычного UX  
    this.EditorPropertyForm.append(el('button', {  
        classList: ['wc-wysiwyg_btn'],  
        props: {  
            type: 'submit',  
            innerHTML: '&#8627;',  
        },  
    }));  
}  
}
```

Не спешите пролистывать код, только в статье я оставляю русские комментарии к коду, на github все на английском и комментариев меньше. К этому моменту мы получили полноценный MVP, осталось разрешить всем элементам редактировать class и можно дальше просто обвешать текст классами из вашего CSS и будет вам счастье:) шучу конечно, больше фишек и возможностей на текущий момент читайте в Readme.md

Это была первая часть публикации, во второй части я рассмотрю реализацию фишек и удобств для редактора, чтобы сделать его по настоящему функциональным, удобным и легковесным веб-компонентом, расскажу про фидбек от сообществ из телеграм каналов, упомяну опыт интеграции в настоящие сайты большие и маленькие и даже в гости к \$m0l узнать как дела у них с веб-компонентами я заглянул, т.к. там тоже про opensource вродебы;)

Заключение

Хочу в конце статьи еще раз напомнить, что версия компонента 0.9.33 что как бы намекает, что для версии 1 еще сыроват компонент, но практическое применение и первых пользователей, а также пару сотен установок в npm и пару звезд на гитхабе он уже нашел, что дает мне силы и мотивацию продолжать развивать это дело на некоммерческой основе. Никаких донатов как некоторые опенсус разработчики под обещания я не собираю и не буду, просто так на чай тоже не нужно, у меня есть любимые галеры с комфортной з.п. а это просто часть развития кругозора)

P. S. все что буду находить и считать полезным и нужным я буду складывать вот тут, не стесняйтесь предлагать свои ссылки в цикл для расширения кругозора и обмена опытом

p.p.s как и обещал попытка вставить HTML5 простые теги в хабр статью — Демонстрация и обзор возможностей веб-компонента wc-wysiwyg — сравните с демкой) за раз всего не рассказать, постараюсь ответить на все вопросы в комментариях) have fun!

- Многие пользователи сначала нажимают на тег, а потом собираются туда что-то писать, но попасть курсором в пустой тег затруднительно по этому мы обрабатываем случай `Selection.toString().length === 0` и если текст не был выделен, добавим в новый тег имя этого тега, чтобы было проще потом отредактировать содержимое тега
- Оборачивать в текст можно не только в простой тег, но и в custom-element так что добавим и поддержку is для автономных веб-компонентов, а для custom-elements просто обернем текст в этот тег, под оборачиванием в текст я имею в виду конструкцию `range.surroundContents(tagNode);`

Дополнения и Файлы

- [Ссылка на оригинальную статью](#)
-  [wc-wysiwyg](#)

From:
<https://synoinstall-gqctx9n8ug2b3eq1.direct.quickconnect.to/> - **worldwide open-source software**

Permanent link:
https://synoinstall-gqctx9n8ug2b3eq1.direct.quickconnect.to/doku.php?id=software:development:web:docs:writing_our_own_wysiwyg_editor

Last update: **2025/07/20 09:18**

