JavaScript - модули

модули;
export/import;
динамический import;
Сборщик модулей

Модули - история

По мере роста нашего приложения, мы обычно хотим разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями.

Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули. Появились библиотеки для динамической подгрузки модулей.

Например:

- AMD – одна из самых старых модульных систем, изначально реализована библиотекой require.js.
- CommonJS – модульная система, созданная для сервера Node.js.
- UMD – ещё одна модульная система, предлагается как универсальная, совместима с AMD и CommonJS.

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и Node.js.

Модули зачем нужны

Модули в JavaScript — это способы организации и изоляции кода. Они позволяют разбивать код на более мелкие, управляемые части, которые можно использовать повторно.

Особенности модулей в JavaScript

- Изоляция: Модули изолируют код, ограничивая его доступность и предотвращая случайные конфликты.
- Повторное использование: Модули позволяют легко использовать одну и ту же часть кода в разных местах приложения.
- Инкапсуляция: Модули скрывают внутренние детали реализации, предоставляя только необходимые интерфейсы.

Плюсы

- Управляемость: Код становится легче управлять и поддерживать, так как он разбит на более мелкие, независимые части.
- Повторное использование: Однажды написанный модуль можно использовать в других проектах, что экономит время и усилия.
- Изоляция и безопасность: Модули помогают избежать конфликтов имен и других проблем, связанных с глобальной областью видимости.
- Загрузка на запрос: Современные системы модулей поддерживают динамическую загрузку, что может улучшить производительность.

Минусы

- Сложность настройки: Использование модулей может потребовать дополнительных настроек, таких как конфигурация загрузчиков модулей.
- Поддержка старых браузеров: Не все старые браузеры поддерживают современные стандарты модулей, что может потребовать использования транспайлеров или полифилов.
- Повышенные требования к структурированию: Необходимость тщательно планировать структуру модулей и их зависимости может усложнить разработку.

Модули

Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

- **export** отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
- **import** позволяет импортировать функциональность из других модулей.

Директива **import** загружает модуль по пути относительно текущего файла и записывает экспортированную функцию в соответствующую переменную.

Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута **<script type="module">**.

Модули не работают локально. Только через HTTP(s)

Если вы попытаетесь открыть веб-страницу локально, через протокол file://, вы обнаружите, что директивы import/export не работают. Для тестирования модулей используйте локальный веб-сервер, например, static-server или используйте возможности «живого сервера» вашего редактора, например, расширение Live Server для VS Code.

Всегда «use strict»

В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

```
<script type="module">
  a = 5; // ошибка
</script>
```

Своя область видимости переменных

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

Если нам нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту window, тогда получить значение переменной можно обратившись к window.user. Но это должно быть исключением, требующим веской причины.

Код в модуле выполняется только один раз при импорте

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортерам.

Такое поведение позволяет конфигурировать модули при первом импорте. Мы можем установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

```
// 📁 alert.js
alert("Модуль выполнен!");
```

```
// Импорт одного и того же модуля в разных файлах

// 📁 1.js
import `./alert.js`; // Модуль выполнен!

// 📁 2.js
import `./alert.js`; // (ничего не покажет)
```

```
// 📁 admin.js
export let admin = { };

export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}
```

```
// 📁 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 📁 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Оба файла, 1.js и 2.js, импортируют один и тот же объект
// Изменения, сделанные в 1.js, будут видны в 2.js
```

import.meta

Объект import.meta содержит информацию о текущем модуле.

Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

```
<script type="module">
  alert(import.meta.url); // ссылка на html страницу для встроенного скрипта
</script>
```

В модуле «this» не определён

Это незначительная особенность, но для полноты картины нам нужно упомянуть об этом.

В модуле на верхнем уровне this не определён (undefined).

Сравним с не-модульными скриптами, там this – глобальный объект:

```
<script>
  alert(this); // window
</script>
<script type="module">
  alert(this); // undefined
</script>
```

Модули являются отложенными (deferred)

Модули всегда выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом *defer*. Это верно и для внешних и встроенных скриптов-модулей.

- загрузка внешних модулей, таких как <script type="module" src="...">, не блокирует обработку HTML.
- модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
- сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

При использовании модулей нам стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать.

Атрибут async работает во встроенных скриптах

Для не-модульных скриптов атрибут async работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут async работает на любых скриптах.

Например, в скрипте ниже есть async, поэтому он выполнится сразу после загрузки, не ожидая других скриптов.

```
<!-- загружаются зависимости (analytics.js) и скрипт запускается -->
<!-- модуль не ожидает загрузки документа или других тэгов <script> -->
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

Не допускаются «голые» модули

В браузере import должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (bare). Они не разрешены в import.

Другие окружения, например Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

```
import {sayHi} from 'sayHi'; // Ошибка, "голый" модуль
// путь должен быть, например './sayHi.js' или абсолютный
```

export/import

В JavaScript, export и import — это ключевые слова, используемые для работы с модулями, что позволяет разработчикам разделять код на независимые, повторно используемые части.

export: Экспорт из модуля - Ключевое слово export используется для предоставления функций, объектов, или значений другим модулям

import: Импорт из модуля - Ключевое слово import используется для загрузки экспортированных значений из других модулей.

Особенности и правила

- Импорт и экспорт должны быть на верхнем уровне: Их нельзя использовать внутри функций, условий или циклов.
- Статический анализ: Браузеры и инструменты сборки могут анализировать зависимости на этапе компиляции, что улучшает производительность и упрощает обнаружение ошибок.
- Модули имеют свою область видимости: Переменные и функции, определенные в модуле, не попадают в глобальную область видимости.
- Неизменность экспортируемых значений: Хотя экспортируемые значения могут быть изменены импортирующим модулем, это изменение не влияет на экспортируемый модуль.

Экспорт именованный (Named Export)

Позволяет экспортировать несколько переменных или функций. Каждый элемент экспортируется под своим именем.

```
// module.js
export const name = 'Alice';
export function greet() {
    console.log(`Hello, ${name}!`);
}
export class Person {
    constructor(name) {
        this.name = name;
    }
    sayHello() {
        console.log(`Hi, I'm ${this.name}`);
    }
}
```

Экспорт отдельно от объявления

Этот метод используется для экспорта ранее объявленных переменных, функций или классов. Это удобно, когда нужно экспортировать сразу несколько элементов, не указывая export при каждом объявлении.

```
// module.js

// Объявление переменных, функций, классов
const name = 'Alice';

function greet() {
    console.log(`Hello, ${name}!`);
}

class Person {
    constructor(name) {
        this.name = name;
    }
    sayHello() {
        console.log(`Hi, I'm ${this.name}`);
    }
}

// Экспорт объявленных элементов
export { name, greet, Person };
```

Экспорт по умолчанию (Default Export)

Позволяет экспортировать одно значение по умолчанию из модуля. Обычно используется для экспорта основного элемента из модуля.

```
// module.js
export default function greet() {
    console.log('Hello, world!');
}
```

Экспорт с переименованием

Можно экспортировать элементы с другими именами.

```
// module.js
const name = 'Alice';
function greet() {
    console.log(`Hello, ${name}!`);
}
export { name as userName, greet as sayHello };
```

Импорт именованный

Позволяет импортировать конкретные элементы из модуля, указав их имена.

```
// main.js
import { name, greet } from './module.js';

console.log(name); // Alice
greet(); // Hello, Alice!
```

Импорт по умолчанию

Используется для импорта значения, экспортированного по умолчанию.

```
// main.js
import greet from './module.js';

greet(); // Hello, world!
```

Импорт всех элементов

Можно импортировать весь модуль как объект и обращаться к его элементам через этот объект.

```
// main.js
import * as myModule from './module.js';

console.log(myModule.name); // Alice
myModule.greet(); // Hello, Alice!
```

Импорт с переименованием

Позволяет импортировать элементы с изменением их имен.

```
// main.js
import { name as userName, greet as sayHello } from './module.js';

console.log(userName); // Alice
sayHello(); // Hello, Alice!
```

Реэкспорт в JavaScript

Реэкспорт (или повторный экспорт) — это важная часть работы с модулями в JavaScript. Он позволяет экспортировать значения из одного модуля через другой модуль, что удобно для создания центральных точек доступа к функциональности, распределенной по нескольким модулям.

Применение и преимущества реэкспорта

- Централизация экспорта: Создание единого модуля для экспорта элементов из различных источников упрощает управление зависимостями и их использование в проекте.
- Удобство в больших проектах: В больших проектах реэкспорт позволяет организовать код так, чтобы было легче находить и использовать необходимые модули.
- Обеспечение изоляции: Реэкспорт помогает изолировать изменения в структуре модулей, не затрагивая код, который их использует.

```
// module1.js
export const name = 'Alice';
export function greet() {
    console.log(`Hello, ${name}!`);
}

// module2.js
export const age = 25;
export function sayAge() {
    console.log(`Age: ${age}`);
}

// centralModule.js
export * from './module1.js';
export * from './module2.js';
```

Динамические импорты

Динамические импорты в JavaScript позволяют загружать модули на лету, во время выполнения программы, а не на этапе загрузки. Это может улучшить производительность приложения, особенно если модули большие или не всегда нужны сразу. Динамические импорты поддерживаются с помощью функции import().

Синтаксис и особенности

- Функция import(): Всегда возвращает промис, даже если модуль уже загружен.
- Использование с async/await: Динамические импорты можно использовать с async/await для более читаемого кода.

Ограничения

- Необходимость поддержки в браузерах: Динамические импорты поддерживаются в современных браузерах, но могут потребовать полифилы или транспиляцию для старых браузеров.
- Требования к структуре проекта: Для использования динамических импортов, особенно в связке с инструментами сборки, нужно правильно организовать проект и конфигурировать сборку.

```
// main.js
const button = document.getElementById('loadButton');

button.addEventListener('click', () => {
    import('./module.js')
        .then(module => {
            module.greet(); // Вызов функции из загруженного модуля
        })
        .catch(error => {
            console.error('Ошибка при загрузке модуля:', error);
        });
});
```

```
// main.js
async function loadModule() {
    try {
        const module = await import('./module.js');
        module.greet(); // Использование импортированного модуля
    } catch (error) {
        console.error('Ошибка при загрузке модуля:', error);
    }
}

loadModule();
```

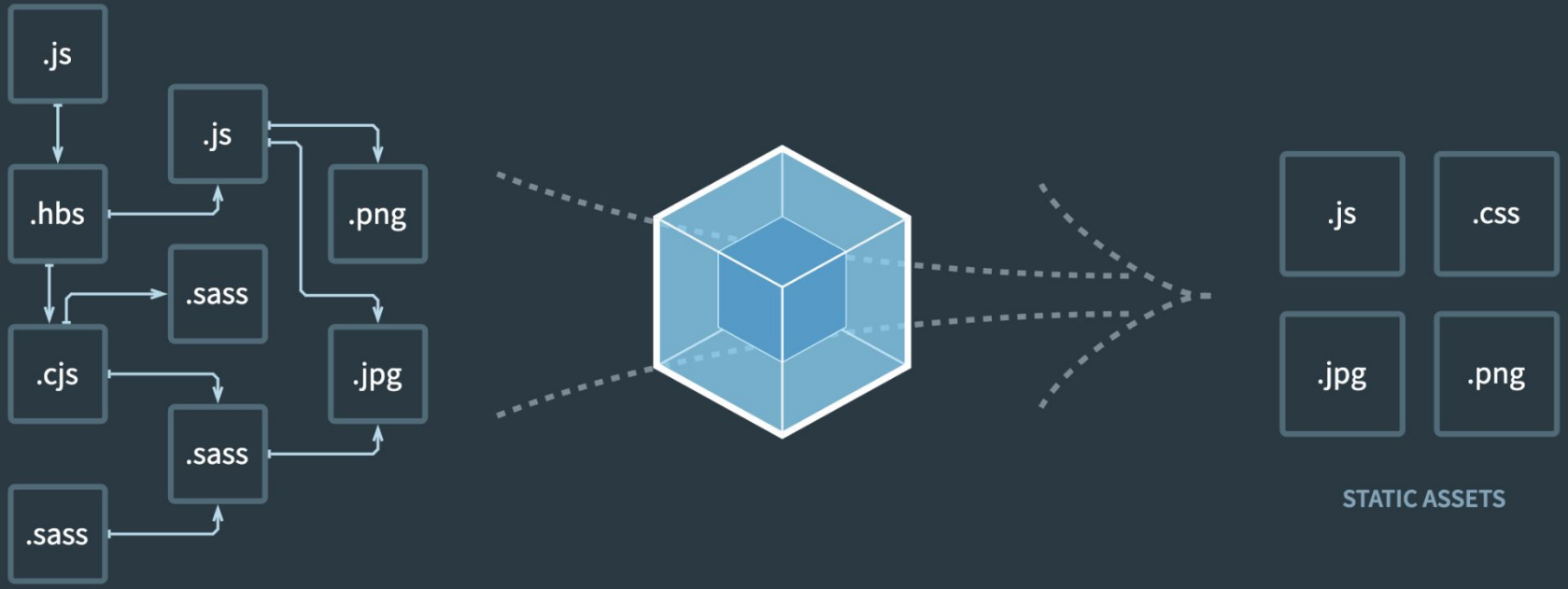
Сборщик модулей

Сборщик модулей (или модульный бандлер) во front-end разработке — это инструмент, который объединяет различные файлы и модули JavaScript, CSS и другие ресурсы в один или несколько файлов для удобства доставки в веб-браузер. Он позволяет разработчикам работать с модулями, организовывая код в удобной для работы структуре, и решает проблемы, связанные с зависимостями и производительностью.

Популярные сборщики модулей:

- WebPack
- Vite
- TurboPack

bundle your styles



Webpack

Webpack — это популярный и мощный сборщик модулей для JavaScript приложений. Он позволяет разработчикам организовывать, оптимизировать и собирать модули в веб-приложениях, что упрощает управление зависимостями и улучшает производительность.

Основные особенности Webpack:

- **Объединение модулей**: Webpack берет модули (JavaScript, CSS, изображения и другие) и объединяет их в один или несколько файлов, называемых бандлами. Это позволяет уменьшить количество запросов на сервер и ускоряет загрузку приложения.
- **Конфигурация**: Webpack чрезвычайно настраиваемый. Он использует файл конфигурации (webpack.config.js), в котором можно определить входные и выходные точки, лоадеры и плагины.
- **Лоадеры**: Это специальные модули, которые позволяют Webpack обрабатывать не только JavaScript, но и другие типы файлов, такие как CSS, изображения, шрифты и т.д. Лоадеры могут трансформировать эти файлы перед включением их в бандл.
- **Плагины**: Плагины позволяют расширить функциональность Webpack. Например, они могут использоваться для оптимизации бандлов, генерации HTML файлов, минификации кода и многого другого.
- **Код сплиттинг (Code Splitting)**: Это техника, которая позволяет разделять код на отдельные части или "чанки". Это помогает уменьшить начальный размер загрузки приложения, загружая дополнительные части кода только по мере необходимости.
- **Хотрелоадинг** (Hot Module Replacement): Эта функция позволяет обновлять модули в браузере без перезагрузки всей страницы, что значительно ускоряет процесс разработки и тестирования.
- **Оптимизация производительности**: Webpack предоставляет различные методы оптимизации, включая минификацию кода, удаление дубликатов и управление кешированием, что помогает улучшить производительность веб-приложений.

Vite

Vite является сборщиком модулей. Он представляет собой современный инструмент для сборки JavaScript приложений, который особенно популярен среди разработчиков, работающих с фреймворками, такими как Vue.js и React.

Vite отличается следующими особенностями:

- **Быстрая разработка:** Vite использует нативные модули ES (ESM) и работает как сервер разработки с быстрой горячей перезагрузкой (hot module replacement, HMR). Это позволяет мгновенно обновлять модули в браузере без перезагрузки всей страницы.
- **Оптимизация производительности**: Для финальной сборки Vite использует Rollup, что позволяет получить оптимизированные и минимизированные бандлы.
- **Поддержка современных фич**: Vite поддерживает многие современные возможности экосистемы JavaScript, включая модульные CSS, TypeScript, и другие языки и инструменты, используемые в современном разработке.

Таким образом, Vite можно рассматривать как гибрид сборщика модулей и сервера для разработки, который упрощает и ускоряет процесс разработки и сборки фронтенд-приложений.

Turbopack

Turbopack — это современный инструмент для сборки фронтенд-приложений, который позиционируется как высокопроизводительная альтернатива традиционным сборщикам модулей, таким как Webpack. Он разработан с акцентом на скорость и эффективность и предназначен для работы с большими и сложными проектами.

Основные особенности Turbopack:

- **Высокая производительность**: Turbopack обещает значительно более быструю сборку и обновление модулей по сравнению с традиционными инструментами. Это достигается за счет оптимизации алгоритмов сборки и эффективного использования многопоточности.
- **Совместимость с Webpack**: Turbopack разработан как модернизированный инструмент, совместимый с Webpack. Это позволяет разработчикам легко перейти на него с минимальными изменениями в конфигурации.
- **Современная архитектура**: Использует современные технологии и парадигмы, такие как модульная система ES и улучшенные механизмы кэширования, что способствует быстрому обновлению кода во время разработки.
- **Акцент на разработчиков**: Turbopack обеспечивает улучшенное взаимодействие с разработчиком за счет более быстрой обратной связи и минимизации времени ожидания при изменениях в коде.

Что такое bundle?

bundle (или бандл) в контексте веб-разработки — это скомпилированный файл, который объединяет в себе несколько модулей или файлов (например, JavaScript, CSS, изображения и т.д.) в один или несколько выходных файлов. Цель создания бандлов — оптимизировать загрузку и использование ресурсов в веб-приложении, уменьшив количество HTTP-запросов и улучшив производительность.

Что такое минимизация кода?

Минимизация кода (или минификация) — это процесс удаления всех ненужных символов из исходного кода, таких как пробелы, переводы строк, комментарии и прочие, без изменения функциональности кода. Минимизация помогает уменьшить размер файлов (например, JavaScript, CSS), что, в свою очередь, ускоряет их загрузку и выполнение в браузере.

Минимизация кода включает следующие основные действия:

- **Удаление пробелов и переносов строк:** Это уменьшает размер файла без изменения логики программы.
- **Сокращение имен переменных:** Переменные, функции и другие идентификаторы переименовываются в более короткие символы.
- **Удаление комментариев:** Комментарии не влияют на выполнение кода, поэтому они удаляются для уменьшения размера файла.
- **Удаление ненужного кода:** Если код определён, но не используется, он также может быть удалён.

Работа WebPack-а “под капотом”

Webpack — это мощный инструмент для сборки модулей, который под капотом выполняет несколько важных функций для преобразования и оптимизации кода. Вот основные этапы и процессы, происходящие "под капотом" Webpack:

- Создание графа зависимостей
- Преобразование модулей
- Объединение и упаковка
- Резолвинг и компоновка (Bundling)
- Плагины
- Вывод и создание файлов

Ресурсы

Модули, введение - ТЫК

Экспорт и импорт - ТЫК

Динамические импорты - ТЫК

WebPack официальный сайт - ТЫК

Vite официальный сайт - ТЫК

TurboPack официальный сайт - ТЫК

Пример TODO приложения с использованием сборщика WebPack - ТЫК