GeekBrains

Инструменты сборки веб-проектов

Модули в JavaScript

На этом уроке

1. Поговорим о проблемах разработчиков в сложных веб-приложениях.
2. Разберём каждую из проблем на примере вымышленного веб-приложения «Таймер, секундомер, будильник, калькулятор дат».
3. Рассмотрим модули в JavaScript и их реализации.
4. Познакомимся с такими модульными системами, как CommonJS, AMD (RequireJS), UMD, ECMAScript Modules (ES2015 Modules).
5. Применим ECMAScript Module на практике, написав первую часть приложения «Калькулятор дат».

Оглавление

Возрастающая сложность веб-сайтов

Исторически так сложилось, что сначала сайты были очень простыми. Их задачи сводились к выводу форматированной информации. Именно поэтому HTML и получил название HyperText Markup Language, язык **разметки** гипертекста.

Со временем сайты усложнялись, обрастали новым инструментарием, и вывода простого текста стало уже недостаточно. Появилась потребность сначала просто запрашивать информацию у пользователя, а в дальнейшем взаимодействовать с ним. Эта эволюция прослеживается и в рамках нашего обучения.

На первых курсах мы верстали простые сайты и этого было достаточно. Но постепенно, изучая JavaScript, приобретались навыки создания сайтов совершенно другого уровня. Появилась возможность взаимодействовать с пользователем.

Но всё имеет свою цену. Например, возрастающая сложность сайтов заставляет разработчика писать более сложные и объёмные по количеству кода скрипты. При этом требуется логика работы сайта с пользователем, а от браузера, ожидаемо (для разработчика), интерпретировать весь этот код и показывать пользователю задуманное.

В рамках нашего курса мы поймём, какие проблемы и сложности вытекают из этого роста сложности веб-сайтов, и узнаем об инструментах, призванных их решать.

Новые вызовы перед разработчиком

Информации о фронтенд-фреймворках здесь нет. Это тема для следующего курса. Наш курс о **процессе** разработки сайта и отправке этого кода в браузер. Мы коснёмся вопроса использования инструментов для непосредственной разработки **продукта**.

Итак, к каким именно проблемам приводит возросшая сложность веб-сайта.

Проблема 1. Рост объёма кода

Подумаем, как появляется эта проблема, и к каким последствиям она приводит.

Представим, что нам поступила задача — разработать простое веб-приложение — таймер. Задача действительно очень простая. Нам нужно одно поле, одна кнопка и простой скрипт из нескольких функций.

В поле, где нужно установить таймер, пользователь вводит время. Нажимаем кнопку «Старт», а скрипт, в свою очередь, обрабатывает нажатие на кнопку, считывает значение поля и отрисовывает оставшееся время. Значение обновляется каждую секунду или миллисекунду.

Сложной ли получилась кодовая база нашего проекта? Очевидно, что нет. В зависимости от оформления, у этого веб-приложения появляются варианты вёрстки или стилей. Но мы сейчас говорим в первую очередь о логике, то есть о JavaScript, куда также входят CSS и HTML.

Где хранить JavaScript:

- в самом HTML файле;
- в отдельном файле, чтобы подключать его как внешний скрипт, используя тег <script src>.

Действительно, пока у нас мало логики, это не критично.

Но к нам приходит заказчик и говорит: «Просто таймер — это хорошо, но наши пользователи хотят больше опций. Требуется добавить функцию сброса времени, сохранять отрезки времени, считать интервалы, и вообще, хорошо бы добавить ещё и сигнал при окончании времени с возможностью отключать звук и выбирать мелодию».

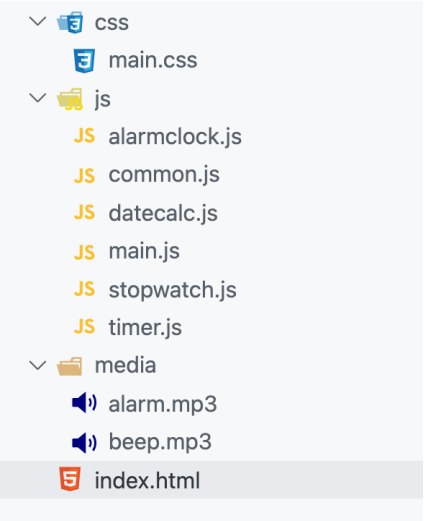
И вот объём нашего кода значительно растёт. Нам уже недостаточно некоторых функций на несколько строчек кода. Очевидно, что потребуется один внешний файл с кодом. В первую очередь это нужно нам, как разработчикам, для удобной разработки и поддержки кодовой базы проекта таймера.

В дальнейшем заказчик придёт за новым инструментарием ещё не раз. Например, он попросит добавить в наше приложение секундомер и калькулятор дат, чтобы считать промежуток между двумя датами. Для корректной работы этому калькулятору требуется учитывать таймзону и её смещение, а также високосные года.

Это увеличит нашу кодовую базу ещё сильнее. Нет, конечно, возможно добавить больше кода в наш один файл, но это приведёт к огромному и сложному файлу на много тысяч строк кода. Это сделает его сложным для разбора и доработок.

Хорошо, поделим наш код на несколько файлов.

Возьмём приблизительную структуру проекта:



1. В css/main.css хранятся стили нашего веб-приложения.
2. В папке js для каждой функциональной части лежит свой файл с кодом, содержащий логику этой части нашего веб-приложения.
3. Файлы common.js и main.js содержат общий код нашего приложения.
4. В папке media находятся звуки для нашего приложения.
5. index.html содержит вёрстку нашего приложения и соединяет все файлы.

Бесспорно, есть и другие варианты организации структуры нашего приложения, а для каждой части инструментария (таймер, секундомер) можно завести свой .html файл. Но представим, что все файлы у нас подключаются в один общий html-файл и меняются динамически (без перезагрузки страницы), когда пользователь нажимает кнопки соответствующей функции.

Итак, в наш index.html требуется подключить все js-файлы:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <link rel="stylesheet" href="css/main.css">
             <script src="js/main.js"></script>

    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

    <script src="js/common.js"></script>
    <script src="js/alarmclock.js"></script>
    <script src="js/stopwatch.js"></script>
    <script src="js/datecalc.js"></script>
    <script src="js/timer.js"></script>
</body>

</html>
```

Получилось много файлов, и появился риск забыть подключить один из них.

Мы разместили наши файлы в самом конце body. Это позволит быстрее загрузить и отрисовать вёрстку приложения (построить DOM-дерево). Наши скрипты без него всё равно бесполезны, поэтому нужно постоянно обрабатывать событие DOMContentLoaded.

Браузер последовательно ожидает загрузку и исполнение каждого указанного в теге script-файла. Поэтому, если они находятся в секции head, есть риск «блокировки» отрисовки всей страницы, пока каждый из файлов не загрузится и выполнится. Например, если у пользователя слабое интернет-соединение или медленное устройство (смартфоны).

Хорошее решение — указать атрибут defer. Он гарантирует, что:

- скрипт выполнится только после окончания парсинга DOM, но до события DOMContentLoaded;
- и в том порядке относительно других defer-скриптов, в каком те указаны в исходном коде.

Помните наш калькулятор дат? Тот самый, что должен учитывать таймзоны и високосные года? Что-то не хочется писать всю логику расчёта самостоятельно — сложная логика и условия. Очень легко ошибиться.

Ничего страшного, воспользуемся трудами сторонних программистов и применим стороннюю библиотеку для работы с датами Luxon. Для этого подключим её в наш проект.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <link rel="stylesheet" href="css/main.css">

    <script src="js/main.js"> </script>

    <script defer
src="<https://cdn.jsdelivr.net/npm/luxon@1.25.0/build/global/luxon.min.js>"></script>
    <script defer src="js/common.js"> </script>
    <script defer src="js/alarmclock.js"> </script>
    <script defer src="js/stopwatch.js"> </script>
    <script defer src="js/datecalc.js"> </script>
    <script defer src="js/timer.js"> </script>
    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

</body>

</html>
```

Отлично, мы сделали это! Теперь, воспользовавшись атрибутом defer, наши скрипты не заблокируют поток отрисовки вёрстки.

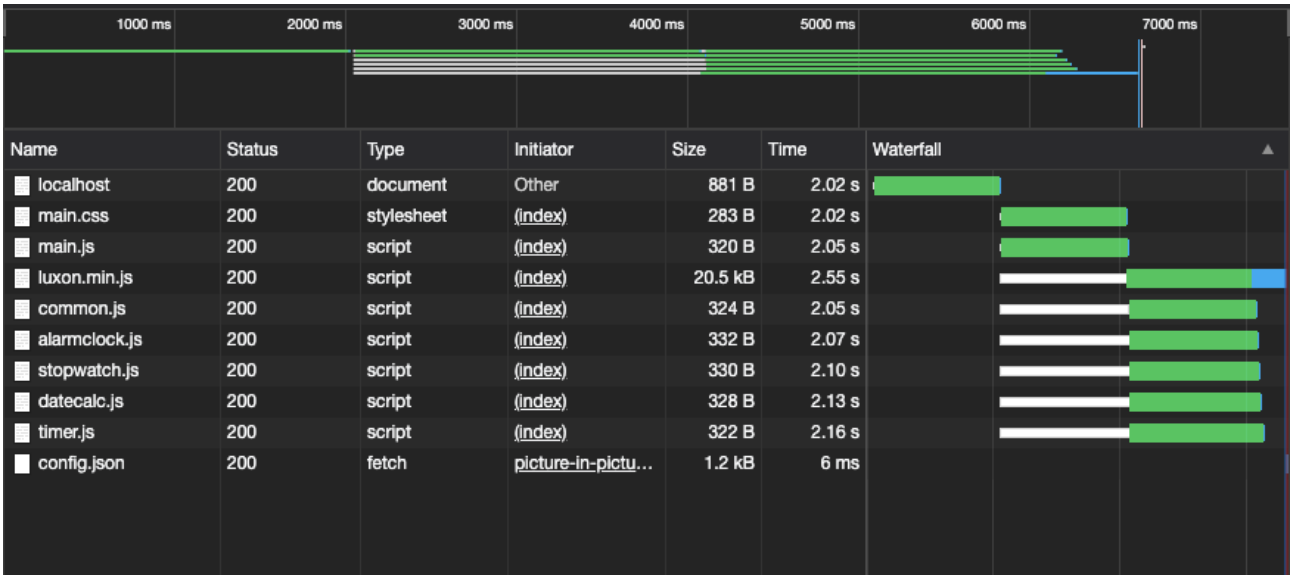
Обратите внимание на файл js/main.js. Предполагается, что он содержит какой-то основной код нашего приложения. Его задача — выполняться сразу, а не ждать окончательной загрузки DOM-дерева.

Итак, решили ли мы проблему большого объёма кода в одном файла? Да, но из этого решения вытекают новые проблемы.

Проблема 2. Растущая нагрузка на сеть

Каждый подключаемый внешний файл — это отдельная сущность файловой системы сервера нашего сайта. Чтобы попасть в браузер, этот файл нужно загрузить. И так с каждым файлом. Чтобы

убедиться в этом, заглянем во вкладку Network инструментов разработчика нашего браузера (Developer Tools).



Да, пока у нас всего 6 файлов со скриптами и один сторонний, это не так страшно. Но что, если мы захотим дальше декомпозировать наш проект, вынося разные сущности по отдельным файлам? Сохраним настройки в файлах config/*.js, пойдём по стопам ООП и под каждый класс создадим свой файл?

Быстро количество файлов перевалит за сотни, а то и тысячи. Это значит, что мы сильно нагрузим наш сервер и устройство клиента, да и подключать все эти файлы в один index.html утомительно.

Проблема 3. Неиспользуемый код

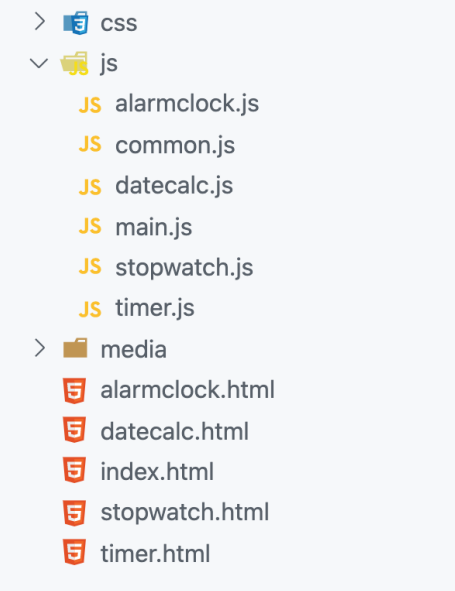
Ещё одна проблема — загрузка неиспользуемого кода, его также называют мёртвым.

Наше веб-приложение состоит из разных частей: таймер, секундомер, будильник, калькулятор дат. Но не все пользователи воспользуются всеми опциями. Один воспользуется только секундомером, а другой захочет лишь узнать количество дней между двумя датами. Но вот беда, каждому такому пользователю на компьютер загрузятся все файлы нашего исходного кода. Однако реально используется только один, максимум два раздела.

Да, с маленьким приложением это небольшая проблема. Но что, если мы создаём большое приложение социальной сети? Загрузим код, отвечающий за отрисовку настроек сообществ пользователю, просто зашедшему посмотреть свою страницу?

Самое простое решение этой проблемы — выделить по html-файлу для каждого раздела. Пользователь будет переключаться между ними с полноценной загрузкой каждой страницы. Для этого нам придётся создать несколько html-файлов и разнести по ним html-код.

Теперь структура нашего приложения выглядит так:



И в каждом html-файле подключим только нужные для работы этой части сайта файлы с JavaScript-кодом.

Файл alarmclock.html:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <link rel="stylesheet" href="css/main.css">

    <script src="js/main.js"> </script>

    <script defer
src="<https://cdn.jsdelivr.net/npm/luxon@1.25.0/build/global/luxon.min.js>"></script>
    <script defer src="js/common.js"> </script>
    <script defer src="js/alarmclock.js"> </script>
    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

</body>

</html>
```

Файл datecalc.html:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <link rel="stylesheet" href="css/main.css">

    <script src="js/main.js"> </script>

    <script defer
src="<https://cdn.jsdelivr.net/npm/luxon@1.25.0/build/global/luxon.min.js>"></script>
    <script defer src="js/common.js"> </script>
    <script defer src="js/datecalc.js"> </script>
    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

</body>

</html>
```

Файл stopwatch.html:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <link rel="stylesheet" href="css/main.css">

    <script src="js/main.js"> </script>

    <script defer
src="<https://cdn.jsdelivr.net/npm/luxon@1.25.0/build/global/luxon.min.js>"></script>
    <script defer src="js/common.js"> </script>
    <script defer src="js/stopwatch.js"> </script>
    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

</body>

</html>
```

Файл timer.html:

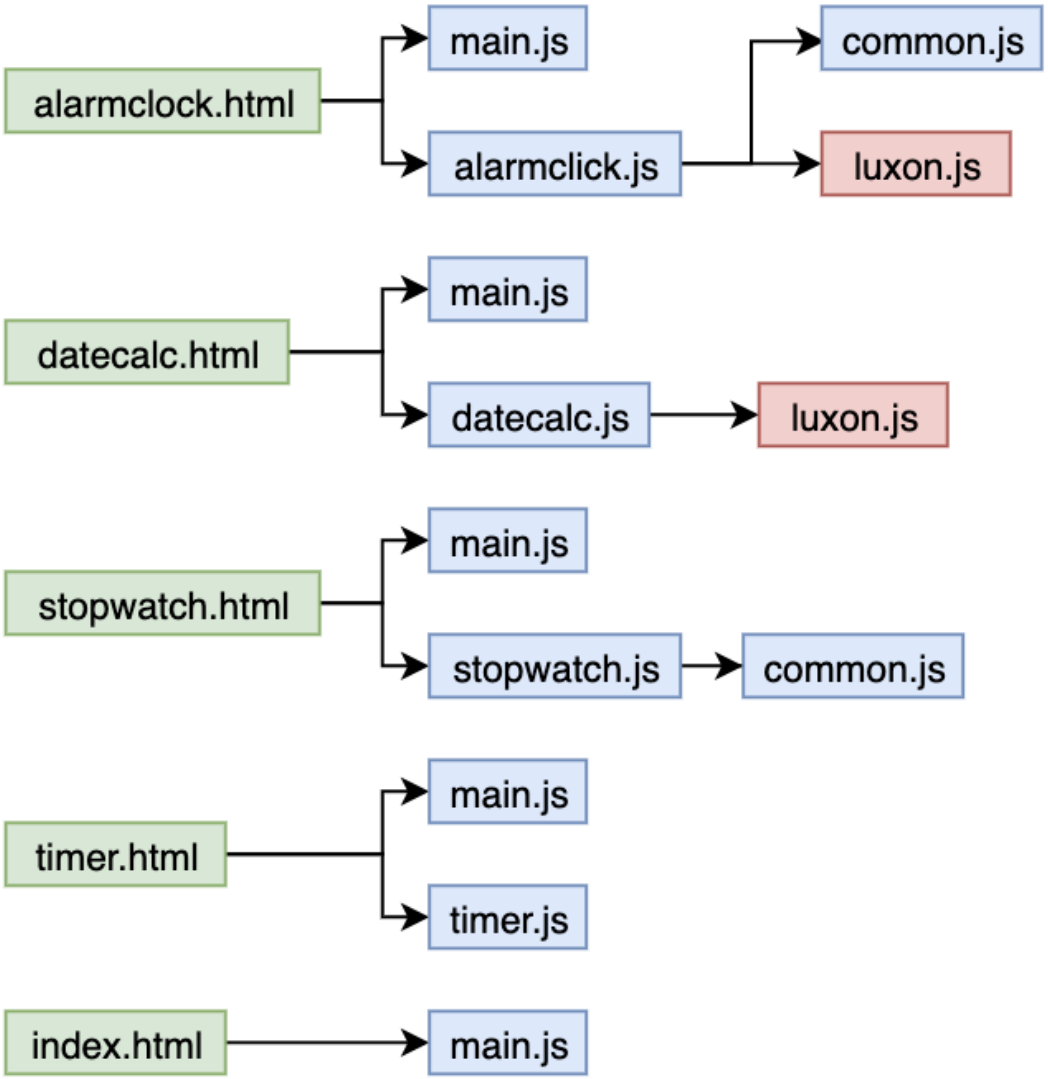
```
<!DOCTYPE html>
<html lang="en">

<head>
    <link rel="stylesheet" href="css/main.css">

    <script src="js/main.js"> </script>

    <script defer
src="<https://cdn.jsdelivr.net/npm/luxon@1.25.0/build/global/luxon.min.js>"></script>
    <script defer src="js/common.js"> </script>
    <script defer src="js/timer.js"> </script>
    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

</body>

</html>
```

Отлично, мы решили ещё одну проблему! Но использует ли код в js/timer.js стороннюю библиотеку Luxon? Нужны ли ему функции, определённые в common.js? Требуется ли Luxon для функционирования stopwatch?

Сразу ответить на эти вопросы не получится, нужно смотреть исходный код каждого файла. Конечно, если добавить тот файл, что не используется приложением, ничего страшного не произойдёт. При этом не забываем о дополнительной нагрузке на сеть. А вот если забыть и не добавить файл с используемыми функциями, часть опций перестанет работать.

Проблема 4. Связи между файлами исходного кода

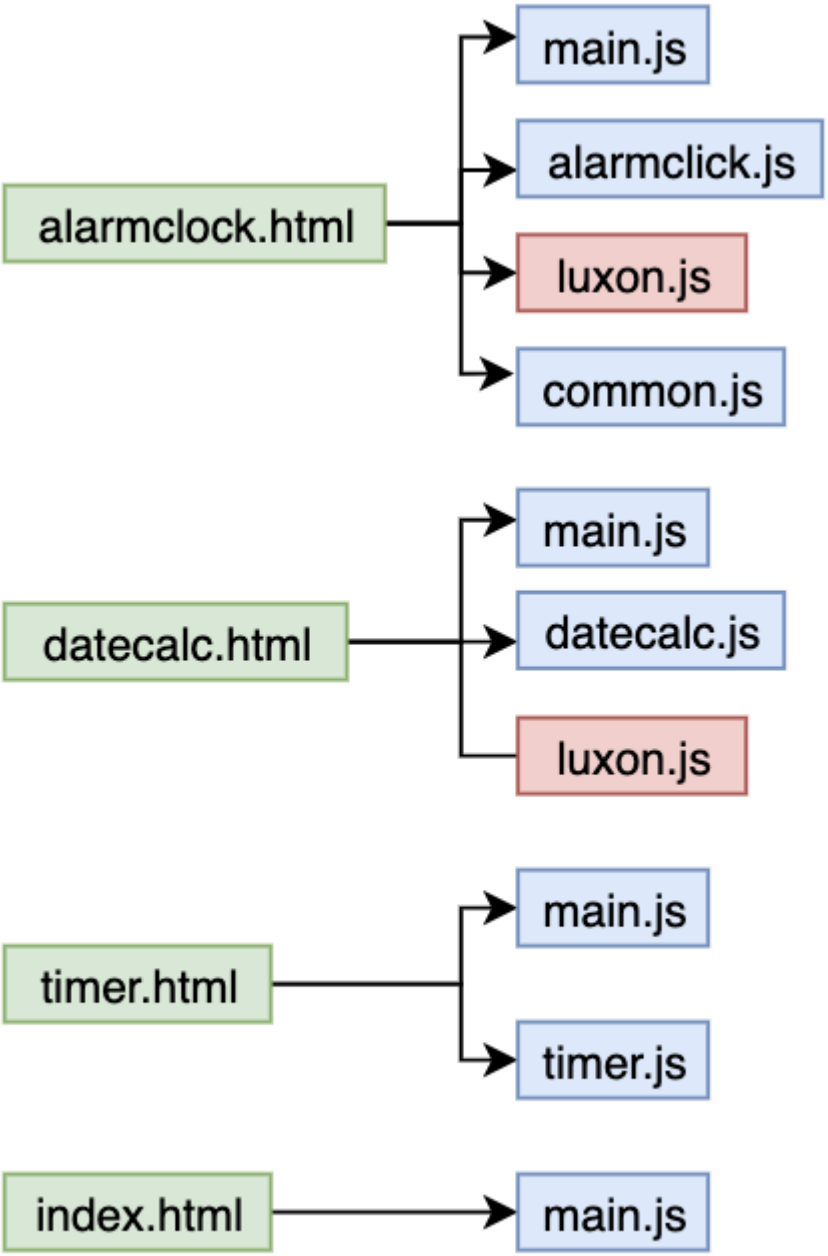
Мы подошли к проблеме связанности исходных файлов нашего приложения. Если внимательно к ним присмотреться, окажется, что они представляют определённую иерархию связей. В ней одни исходные файлы зависят от вторых, а третьи — полностью независимые и в дополнительных внешних файлах не нуждаются:



Это примерная схема и её цель — показать возможные варианты связей файлов исходного кода в абстрактном веб-приложении

Но если присмотреться к нашему приложению, пример исходного приводится в предыдущем разделе, окажется, что каждый из файлов загружается из html-файла. Это значит, что структура приложения плоская, а сами файлы с кодом ничего не знают об остальных нужных для его работы файлах. Это ещё принято называть зависимостями, от английского **dependencies**.

Наш html-файл, наоборот, владеет слишком большой частью приложения. Ему приходится знать о зависимостях кода, хотя он никак напрямую их и не использует:



На этой схеме нет «мёртвых» зависимостей, то есть тех файлов, подключение которых мы указали в примерах исходного кода в предыдущей главе, но которые в действительности не используются

Чтобы решить эту проблему, нужно добавить новые теги <script> динамически, с использованием методов манипулирования DOM-деревом. И браузер, увидев новый тег, тут же подгрузит внешний файл, на который он указывает.

Напишем простую функцию для добавления в наш документ новые скрипты:

Файл load.js:

```
function loadScript(url, callback) {
    const element = document.createElement("script");
    element.type = "text/javascript";
    element.src = url;
    element.onload = callback;

    document.body.appendChild(element);
}
```

Обратим внимание на функцию обратного вызова (callback) и повесим её на событие onload нашего нового элемента. Нужно, чтобы наш файл, запрашивающий загрузку другого файла, продолжил исполняться, только когда код успешно загрузится. При этом не забываем, что все операции, связанные с сетью, асинхронные.

Файл timer.html:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <link rel="stylesheet" href="css/main.css">

    <script src="js/main.js"> </script>
    <script src="js/load.js"> </script>

    <script defer src="js/timer.js"> </script>
    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

</body>

</html>
```

Файл timer.js:

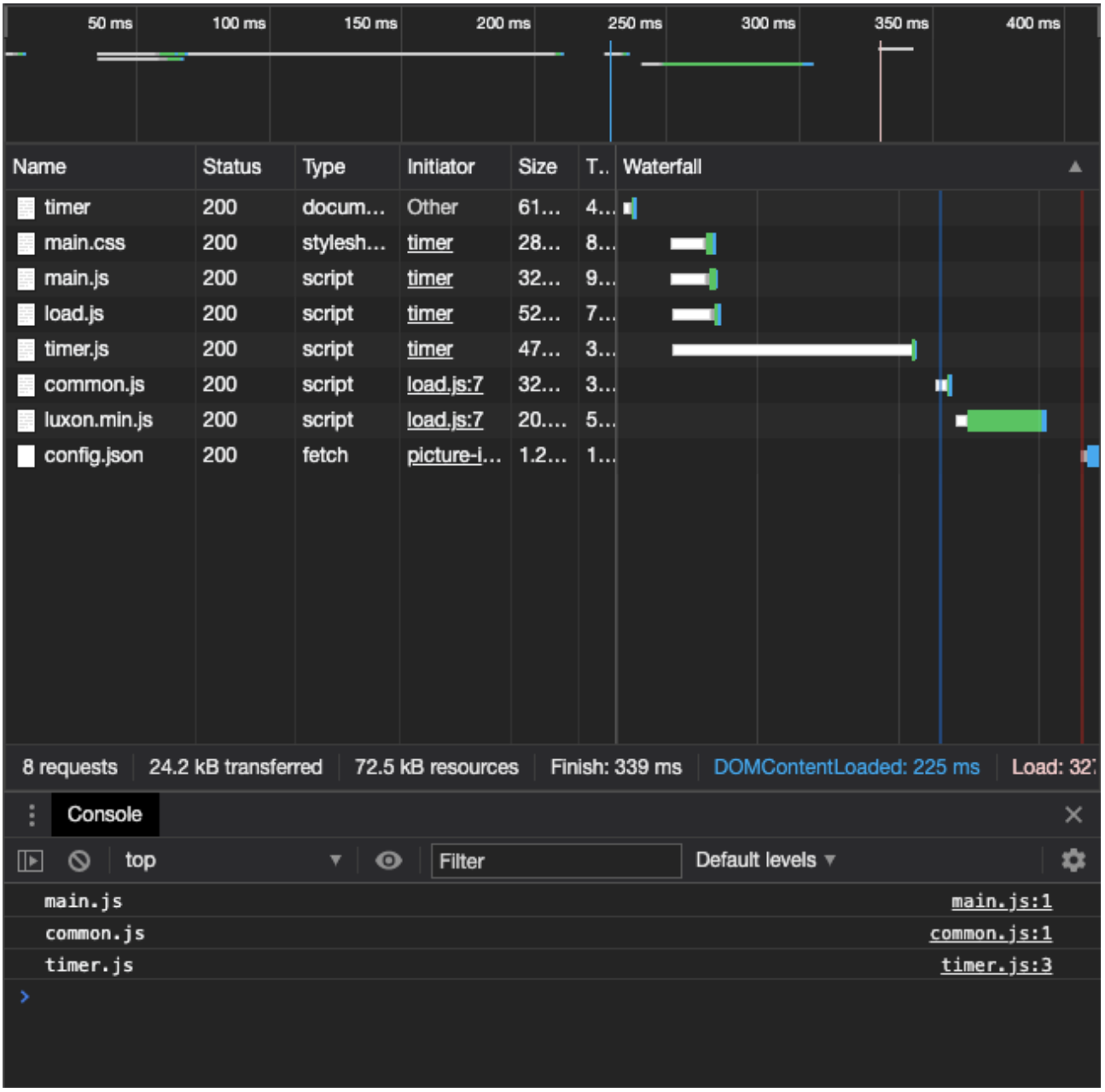
```
loadScript("js/common.js", () => {

loadScript("<https://cdn.jsdelivr.net/npm/luxon@1.25.0/build/global/luxon.min.js>",
() => {
        console.log("timer.js ")
```

```
    })
})
```

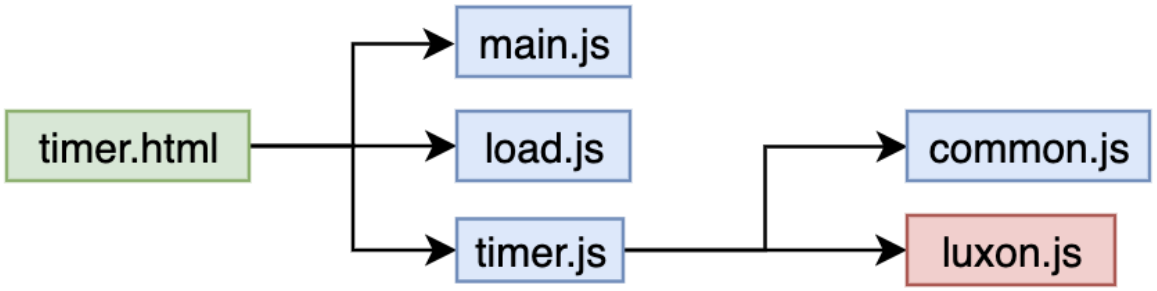
Файл common.js:

```
console.log("common.js");
```

Если запустим этот код в браузере, то увидим, что последовательность загрузки файлов и вывода текста в консоль примет вид:



Сначала из timer.html загружаются main.js и load.js (с функцией loadScript), потом timer.js, требуя загрузить файлы common.js и luxon, и лишь после этого выполняется код внутри функции обратного вызова.



Отлично, мы решили часть проблем. Теперь наш html-файл знает только о тех файлах, с которыми взаимодействует напрямую, а js-файлы могут загружать нужный им для работы код.

Такая структура имела бы и более сложный вид, если, например, common.js мог (чисто теоретически) загрузить какой-то дополнительный код, требуемый для его работы.

Но наша простая реализация функции loadScript далека от идеала. Например, она принимает всего один файл, из-за чего, при загрузке нескольких внешних файлов, приходится делать вложенные функции. Чтобы это исправить, нужно передать в неё массив путей к файлам.

Нерешёнными остаются вопросы, касающиеся:

- рекурсивной загрузки файлов, когда файл A зависит от файла B, а файл B — от файла A;
- запроса уже загруженных файлов, когда файл A зависит от B, и C зависит от B.

Проблема 5. Отсутствие изоляции кода разных файлов

Теперь мы можем подключать свои и даже сторонние файлы, написанные другими разработчиками. Но всегда есть риск дать имя функции или переменной, находящейся в нашей кодовой базе. Особенно, когда это большое приложение.

Лучше, если бы наши файлы прямо указывали на нужную им функцию из конкретного файла. На самом деле, такая замкнутая в себе сущность уже придумана до нас и называется **модулем**.

Каждый модуль прямо определяет:

- что он хочет экспортировать наружу для других модулей;
- а что хочет импортировать себе, из других модулей.

Часть приложения, отвечающая за загрузку и управление модулями называется **системой модулей**. Мы обязательно рассмотрим такие системы чуть дальше в уроке.

Улучшать код функции loadScript и прописывать собственную систему модулей не требуется. Всё уже придумали за нас. Рассмотрим имеющиеся на текущий момент системы модульности. А в качестве практического задания попробуем улучшить функцию loadScript.

Системы модулей

Итак, что же такое модуль? Эдди Османи, автор книги «Паттерны для масштабируемых JavaScript-приложений», даёт следующее определение модуля:

«Модуль — это популярная реализация паттерна, инкапсулирующего приватную информацию, состояние и структуру. Паттерн «модуль» возвращает только публичную часть API, оставляя всё остальное доступным только внутри замыканий».

Это хорошее решение, чтобы скрыть внутреннюю логику от посторонних глаз и производить всю тяжёлую работу исключительно через интерфейс. Скрытие внутренней логики достигается через код, доступный нам, как разработчикам, чтобы использовать его и в других частях приложения.

Как мы выяснили выше по тексту, код записывается и без использования модулей. Но модуль позволяет решить ряд проблем:

1. **Управлять зависимостями**, легко изменяя зависимости без переписывания кода.
2. **Абстрагировать код**, передавая функциональные возможности сторонним библиотекам, так что не придётся разбираться во всех сложностях их реализации.
3. **Инкапсулировать код**, скрывая его внутри модуля без изменений.
4. **Переиспользовать код**, избавляясь от постоянного написания одного и того же.

При этом модули никак не решают вопрос дополнительной нагрузки на сеть. Конечно, мы решим этот вопрос, но чуть дальше по курсу. А система модулей в этом только поможет.

Так как модуль — это паттерн, то в мире JavaScript есть несколько его реализаций. Одни из самых популярных считаются CommonJS, AMD (RequireJS), UMD, ECMAScript Modules (ES2015 Modules). На самом деле, систем модулей намного больше. Мы рассмотрим только самые популярные, чтобы понять общее направление развития модулей в языке. Подробно остановимся лишь на ECMAScript Modules, как на системе модулей, появившейся в стандарте языка и поставившей точку на всех остальных.

Для рассмотрения и сравнения разных систем модулей поставим перед собой следующую задачу: написать простой код из функции, что возвращает текст приветствия на указанном языке. Без модулей решение выглядит следующим образом:

```
const translates = {
      en: "Hello, ",
      ru: "Привет, ",
}

function sayHello(name, lang = "en") {
      return translates[lang] + name;
}

console.log(sayHello("Ivan"));
```

Константа translates находится в глобальной области видимости, что не желательно. Конечно, можно перенести её в саму функцию sayHello, но в этом случае она будет инициализироваться при каждом вызове функции, что тоже нехорошо.

Модули CommonJS

Модульная система CommonJS появилась в 2009 году. Она предназначается для использования в серверном окружении JavaScript. В первую очередь, в Node.js.

Для загрузки модуля вызывается функция require(), а экспортируемые данные присваиваются как свойства объекту module.exports. Простой пример выглядит так:

Файл greeting.js:

```
const translates = {
      en: "Hello, ",
      ru: "Привет, ",
}
function sayHello(name, lang = "en") {
      return translates[lang] + name;
}

module.exports = sayHello;
```

Файл main.js:

```
const { sayHello } = require('./greeting');

console.log(sayHello("Ivan"));
```

Теперь мы экспортируем только функцию sayHello. Это значит, что константа translates больше недоступна извне.

Так как module и require не считаются встроенными конструкциями JavaScript, Node.js оборачивает код каждого файла в такую конструкцию:

```
(function (exports, require, module, __filename, __dirname) {
    // Your code is injected here!
});
```

Эта система модулей используется и в браузере. Но для этого нужно использовать специальные инструменты сборки. О них мы поговорим в следующем уроке.

Модули AMD (RequireJS)

Модульная система AMD появилась в 2009 году и расшифровывается как Asynchronous Module Definition. Не стоит путать с производителем процессоров для компьютеров. AMD возникла как ответ на полностью синхронную систему модулей CommonJS. Она позволяет загружать модули в асинхронном режиме.

Для определения модуля требуется вызвать функцию define. Если первым аргументом передать в неё массив, он расценится как список зависимостей для этого модуля. В любом случае код модуля нужно указывать в функции, которая будет выполнена. Только когда все модули, указанные как зависимости, загрузятся.

Файл greeting.js:

```
define(function() {
	const translates = {
		en: "Hello, ",
		ru: "Привет, ",
	}
	function sayHello(name, lang = "en") {
		return translates[lang] + name;
	}

	return { sayHello };
});
```

Файл main.js:

```
define(["greeting"], function({ sayHello }) {
	console.log(sayHello("Ivan"));
});
```

Для использования AMD в браузере нужно использовать стороннюю библиотеку RequireJS. Она предоставляет API для работы с функцией define и асинхронную загрузку файлов.

Модули UMD

CommonJS и AMD активно конкурировали между собой, но AMD эту схватку проиграл. Однако разный способ объявления модулей в этих двух системах приносил много хлопот разработчикам библиотек. Они не знали, в какой именно системе модульности будет использоваться их код. Поэтому в 2011 году появился способ написания модулей для работы в обеих системах — UMD (Universal Module Definition).

Файл greeting.js:

```
(function(define) {

    define(function () {
        const translates = {
                                en: "Hello, ",
                                ru: "Привет, ",
                            }
                            function sayHello(name, lang = "en") {
                                return translates[lang] + name;
                            }

                            return { sayHello };
    });

}(
    typeof module === 'object' && module.exports && typeof define !== 'function' ?
    function (factory) { module.exports = factory(); } :
    define
));
```

Теперь у нас есть самовызывающаяся функция. Она проверяет, определился ли у нас в глобальной области module или define. Писать такую большую обёртку самостоятельно не придётся, если воспользоваться инструментами сборки, о которых мы поговорим на следующем уроке.

ECMAScript Modules (ES2015 Modules)

Всем понимали, что долго такое разнообразие систем модулей уживаться не сможет. Поэтому в 2010 году комитет ECMAScript начал работу над стандартизацией системы модулей. Её отличительной чертой стала поддержка на уровне интерпретатора и, как следствие, наличие собственного

синтаксиса. Это было невозможно в предыдущих системах модулей, добавляющих модули поверх существующих конструкций языка.

Новая система модулей включалась в спецификацию ES2015, вышедшую в 2015 году. Она добавила две новые синтаксические конструкции, import и export. Наш пример с функцией sayHello теперь выглядит следующим образом:

Файл greeting.js:

```
const translates = {
      en: "Hello, ",
      ru: "Привет, ",
}

export function sayHello(name, lang = "en") {
      return translates[lang] + name;
}
```

Файл main.js:

```
import { sayHello } from "./greeting.js";

console.log(sayHello("Ivan"));
```

Мы указываем расширение файла greeting.js и путь до файла как «./». Эта конструкция означает, что файл модуля находится в той же папке. Без этих важных указаний браузер не разрешит зависимость, и всё закончится ошибкой. На втором уроке познакомимся с инструментами сборки, указывая только имя файла.

Благодаря полноценному синтаксису, наш код получился изящным и выразительным. К сожалению, в старых браузерах он работать не будет.

Браузер и ECMAScript Modules

Благодаря официальной спецификации, эта система модулей вытесняет все остальные. Она уже имеет нативную поддержку в последних версиях браузеров и node.js. Разберём её подробнее.

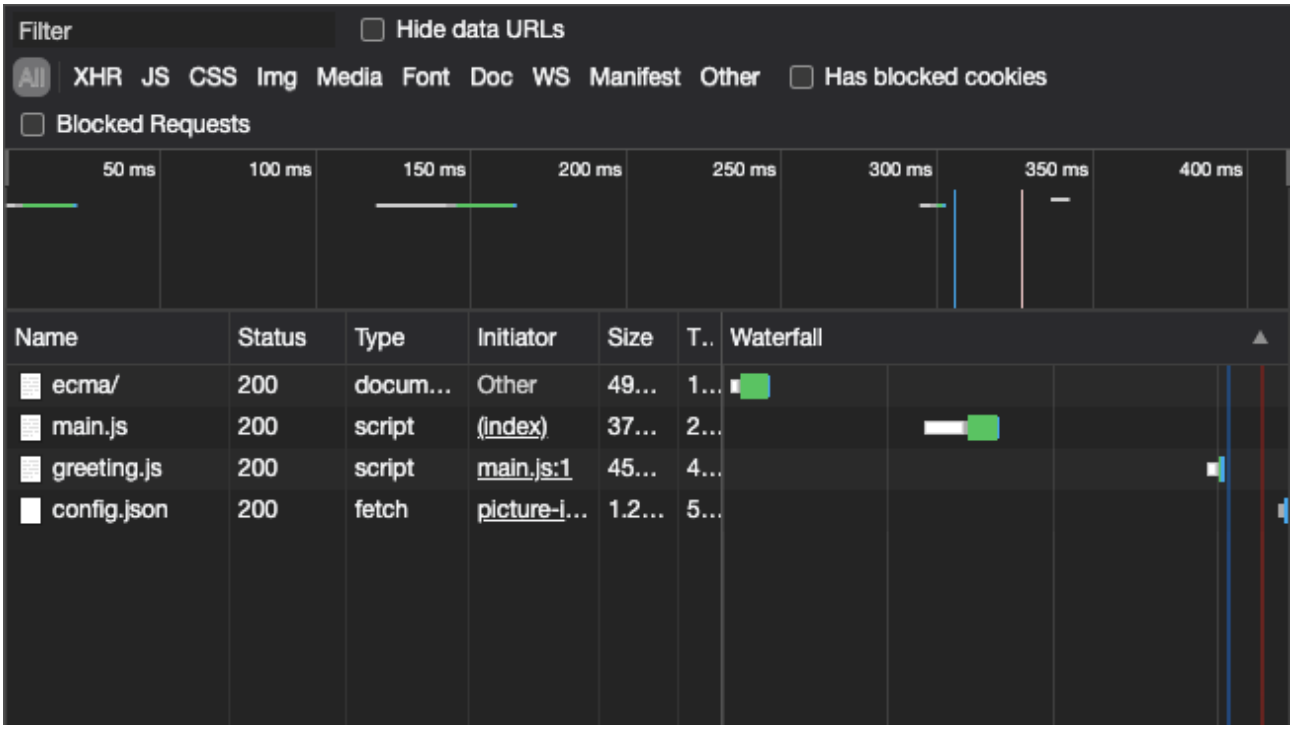
Для выполнения следующих заданий, убедитесь, что браузер поддерживает модули. Это можно сделать на caniuse.com.

Нужно явно указать браузеру, что файл с JavaScript содержит модуль. Для этого требуется прописать тег <script type="module">. Допускается как написание кода во внешнем подключаемом файле, так и напрямую в html файле. Посмотрим, как подключить файл main из примера с функцией sayHello в html.

Файл index.html:

```
<!DOCTYPE html>
<html lang="en">

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

    <!-- остальной код -->
</head>

<body>

    <!-- остальной код -->

</body>

</html>
```

Обратите внимание, теперь в файл html подключается только один js-файл. При этом новые версии браузера понимают, что у модуля (файла) main.js есть зависимость greeting.js, и автоматически её подгружают.



А что делать со старыми версиями браузеров? Есть два варианта.

1. Указать специальный тег <script nomodule>. Скрипты, указанные в таком модуле, отработают только в старых браузерах:

```
<script type="module" src="./main.js"> </script>
<script nomodule> alert("Ваш браузер не поддерживается =("); </script>
```

2. Применить инструменты сборки. О них говорится во втором уроке курса.

Синтаксис ECMAScript Modules

Подробнее рассмотрим синтаксис. Вспомним функцию sayHello.

Файл greeting.js:

```
const translates = {
      en: "Hello, ",
      ru: "Привет, ",
}

export function sayHello(name, lang = "en") {
      return translates[lang] + name;
}
```

Оператор export объявляется на самом верхнем уровне файла. Его запрещается вкладывать в функции и любые другие конструкции (ветвление, циклы), иначе это приведёт к неочевидному экспорту по условиям.

Он прямо указывает, что именно экспортируется. Все остальные сущности будут доступны только в рамках этого файла (модуля).

В нашем примере это константа translates. Она недоступна ни из main.js, ни из index.html. Чтобы можно было получить к ней доступ, требуется также указать перед ней оператор export.

Допускаются следующие варианты экспорта:

```
export var user = "Ivan";
export const age = 20;
export let role = true;
export function bar() { };
export const baz = () => { };
export class Foo { };
```

Это принято называть «именованным экспортом».

Если эта переменная объявилась, указываем фигурные скобки при экспорте:

```
const email = "test@ya.ru";
function check (){};

export { email, check };
```

Чтобы переименовать экспортируемую сущность, применяется ключевое слово as:

```
const email = "test@ya.ru";
function check (){};

export { email as login, check };
```

Для импорта какой-либо сущности в модуле требуется указать оператор import:

```
import { user } from "./path/to/file/filename.js";
```

И после этой строчки используем user в своём коде. Не забываем о регистре символов, он важен, как и у любой другой переменной.

Импортируются также несколько сущностей из одного файла или из разных:

```
import { user, age } from "./path/to/file/filename1.js";
import { email } from "./path/to/file/filename2.js";
```

Импортируемая сущность переименовывается, если, например, есть конфликт имён:

```
import { user as login, age } from "./path/to/file/filename1.js";
```

Импортируются и все экспортируемые свойства модуля:

```
import * as userModule from "./path/to/file/filename1.js";
```

Если нужно запустить какой-либо код в модуле без импорта, просто указываем его имя:

```
import "./path/to/file/filename1.js";
```

Есть также экспорт и импорт «по умолчанию». Он отличается от приведённого выше именованного экспорта, тем, что при экспорте не указывается конкретное имя сущности:

```
export default "любой текст";

export default () => {};
```

Для импорта по умолчанию указывается:

```
import Name from "./path/to/file/filename1.js";
```

В этом случае имя сущности задаётся при импорте, а не экспорте. Обратим внимание на различие в синтаксисе импорта «по умолчанию» и синтаксиса импорта всех экспортируемых сущностей модуля.

Применение системы модулей на практике

Напишем часть нашего приложения с применением системы модулей ECMAScript на практике. Начнём с калькулятора дат. Она будет предельно простая, чтобы сосредоточиться на особенностях работы с модулями.

Итак, у нас два поля для ввода даты (input type="date") и кнопка «рассчитать промежуток». Если пользователь не заполнит одно или оба поля, покажем ошибку. Для расчёта дат воспользуемся сторонней библиотекой работы с датой Luxon.

Начнём с вёрстки, файл **index.html:**

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Datetime superapp</title>
    <script src="./src/main.js" type="module"></script>
</head>

<body>
    <form id="datecalc">
        <h3>Калькулятор дат</h3>
        <label>
            <strong>Первая дата:</strong>
            <input type="date" name="firstDate" />
        </label>
        <label>
            <strong>Вторая дата:</strong>
            <input type="date" name="secondDate" />
        </label>
        <button type="submit">Расчитать промежуток</button>
        <p id="datecalc__result"></p>
    </form>
</body>

</html>
```

Здесь всё просто и без сюрпризов. На внешний вид вёрстки не отвлекаемся.

Теперь напишем основную логику файла src/main.js с выводом данных из формы в консоль.

Файл src/main.js:

```
const dateCalcForm = document.getElementById("datecalc");
const dateCalcResult = document.getElementById("datecalc__result");

dateCalcForm.addEventListener("submit", handleCalcDates);

function handleCalcDates(event) {
    dateCalcResult.innerHTML = "";
    event.preventDefault();

    let { firstDate, secondDate } = event.target.elements;
    firstDate = firstDate.value, secondDate = secondDate.value;

    if (firstDate && secondDate) console.log(firstDate, secondDate);
    else console.log("Для расчета промежутка необходимо заполнить оба поля");
}
```

Важно! *По умолчанию модули всегда выполняются только после полной загрузки DOM-дерева, соответственно не блокируют отрисовку вёрстки и позволяют обращаться к DOM-элементам без обработки события DOMContentLoaded.*

Для начала получаем нужные нам DOM-элементы, устанавливаем обработчик формы и пишем основную логику валидации поля.

Теперь заведём функцию, которая отформатирует вывод как ошибку. Её мы поместим в файл src/utils.js. В этом сейчас нет большого смысла, мы делаем это для практики.

Файл src/utils.js:

```
export const formatError = text => `
<span style="color: red;">
    ${text}
</span>
`;
```

Далее напишем функцию расчёта промежутка между датами. Для этого воспользуемся инструментарием luxon. Так как функция diff возвращает объект, нам также понадобится функция, превращающая его в html.

Файл src/datecalc.js:

```
import { DateTime } from "./luxon.js"; // 1

export function diffDates(firstDate, secondDate) {
```

```
    firstDate = DateTime.fromISO(firstDate);
    secondDate = DateTime.fromISO(secondDate);

    if (firstDate > secondDate)
        secondDate = [firstDate, firstDate = secondDate][0]; // 2

    return secondDate.diff(firstDate, ['years', 'months', 'days']).toObject();
}

// 3
export const diffToHtml = diff => `
    <span>
        ${diff.years ? 'Лет: ' + diff.years : ''}
        ${diff.months ? 'Месяцев: ' + diff.months : ''}
        ${diff.days ? 'Дней: ' + diff.days : ''}
    </span>
`;
```

Разберём помеченные цифрами строчки кода в этом листинге:

1. Загружаем локально сохранённый исходный код luxon. Он находится здесь.
2. Меняем даты, если первая дата больше второй. Это нужно, чтобы не появились цифры с минусами.
3. Функция diffToHtml форматирует объект как html. Она имеет вид стрелочки, чтобы показать разные варианты экспорта. Придерживаемся одинакового синтаксиса объявления функций в проекте.

Теперь объединим всё в файле src/main.js:

```
import { diffDates, diffToHtml } from "./datecalc.js"; // 1
import { formatError } from "./utils.js"; // 2

const dateCalcForm = document.getElementById("datecalc");
const dateCalcResult = document.getElementById("datecalc__result");

dateCalcForm.addEventListener("submit", handleCalcDates);

function handleCalcDates(event) {
    dateCalcResult.innerHTML = "";
    event.preventDefault();

    let { firstDate, secondDate } = event.target.elements;
    firstDate = firstDate.value, secondDate = secondDate.value;

    if (firstDate && secondDate) {
        const diff = diffDates(firstDate, secondDate); // 3
        dateCalcResult.innerHTML = diffToHtml(diff); // 4
    }
```

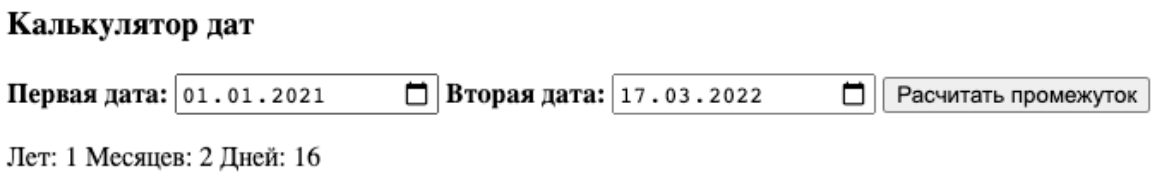
```
    else dateCalcResult.innerHTML = formatError("Для расчета промежутка необходимо заполнить оба поля"); // 5
}
```

Разберём помеченные цифрами строчки кода:

1. Импортируем 2 функции из файла src/datecalc.js.
2. Импортируем функцию из файла src/utils.js.
3. Вычитаем промежуток между датами.
4. Форматируем промежуток, как html и вставляем в DOM.
5. Если одно из полей не заполнено (или оба), выводим ошибку.

Каждый из файлов исходного кода загружается отдельно:

Приложение считает промежуток:



И выводит ошибку, если одно из полей пустое:

Вот так легко мы воспользовались системой модулей. Однако работает это только в современных браузерах. На следующем уроке мы узнаем, как усовершенствовать этот проект с использованием инструментов сборки кода.

Практическое задание

1. Реализуйте таймер. Включите в него общую кодовую базу с калькулятором дат, реализованным в этом уроке. Вложите его в тот же index.html, что и калькулятор дат, реализовав код переключения между разделами страницы.
    a. Вынесите код переключения между разделами (таймер/калькулятор дат) в отдельный модуль.
    b. Вынесите общие функции обоих разделов в отдельный модуль.
    c. Включите в таймер поле для установки времени и кнопки «Старт», «Стоп».
    d. Для обновления времени на странице используйте setInterval().
    e. Добавьте звуковое сопровождение, когда время заканчивается. Для работы со звуком воспользуйтесь сторонней библиотекой, например, Howler.js.
    f. Не используйте другие сторонние библиотеки.
2. В качестве эксперимента доработайте нашу функцию loadScript
    a. Её аргументы:
        i. Первый аргумент: коллбек или строка с url до файла или массив с url до файлов зависимостей;
        ii. Второй аргумент: необязательный коллбек (только если первый аргумент строка или массив)
    b. Её задачи:
        - обнаруживать повторно запрашиваемые зависимости и не загружать их: ситуация, когда модуль A зависит от B, и C зависит от B.
        - Подумайте, как реализовать вызов callback модуля A после того, как разрешатся все зависимости модуля B, и отработает его callback. Попробуйте реализовать логику работы функции define из системы модулей AMD (RequireJS).

Используемые источники

1. Статья «MDN: script defer.»
2. Статья «MDN: Модули JavaScript.»
3. Статья «CommonJS Modules.»
4. Статья «Requirejs documentation.»
5. Статья «UMD (Universal Module Definition).»
6. Статья «Википедия: Асинхронное определение модуля.»

Дополнительные материалы

1. Статья «Learn JavaScript. Модули.»
2. Статья «Habr: Эволюция модульного JavaScript.»
3. Статья «Habr: Путь JavaScript модуля.»
4. Статья «Habr: Тонкости модульной системы ECMAScript 2015 (ES6).»
5. Перевод книги Эдди Османи «Паттерны для масштабируемых JavaScript-приложений».