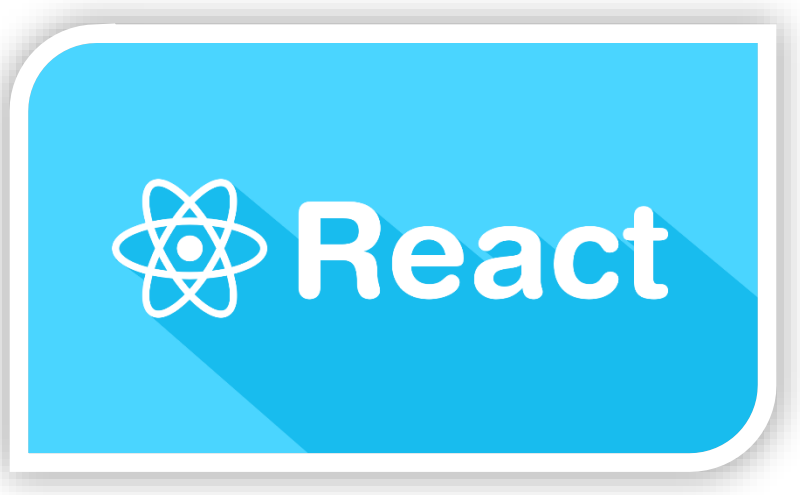


Задачи за домашно:

1) Преминете през примерите от:

- https://jscomplete.com/playground/rgs3.1

- https://jscomplete.com/playground/rgs3.2

- https://jscomplete.com/playground/rgs3.3

- https://jscomplete.com/playground/rgs3.4

- https://jscomplete.com/playground/rgs3.5

- https://jscomplete.com/playground/rgs3.6

- https://jscomplete.com/playground/rgs3.7

- https://jscomplete.com/playground/rgs3.8

- https://jscomplete.com/playground/rgs3.9

2) Опитайте се да пресъздадете локално, като използвате Vite и разделите кода на отделни компоненти.

- npm create vite@latest star-match-game

- cd star-match-game

- npm install

- npm run dev

3) Разгледайте теорията на следващите страници и изпробвайте примерите, описани в края.

Теория

React е библиотека написана на JavaScript с отворен код, създадена за изграждане на потребителски интерфейси или компоненти на потребителския интерфейс. Поддържа се от Facebook и общност от отделни разработчици и компании. React може да се използва като основа при разработването на едностранични или мобилни приложения. React обаче се занимава само с управление на състоянието и визуализиране на това състояние към DOM, така че създаването на приложения на React обикновено изисква използването на допълнителни библиотеки. Обикновено се смята за "изгледен" слой на приложение. Точно както jQuery манипулира елементи на потребителския интерфейс, React променят какво потребителят вижда. В общия случай имаме някаква логика на приложението, която генерира данни. Искаме да предоставим тези данни на потребителския интерфейс? Предаваме ги на React Component, който се справя със задачата да изгради HTML страницата.

React е разделен на две основи:

- React Component API: Това са частите на страницата, които всъщност са изведени от React DOM. React ви позволява да дефинирате компоненти като класове или функции. За да дефинирате клас React компонент, трябва да наследите React.Component. Единственият метод, който трябва да дефинирате в подклас React.Component, се нарича render ().

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- React DOM: Това е API, което се използва за реално изобразяване в страница.

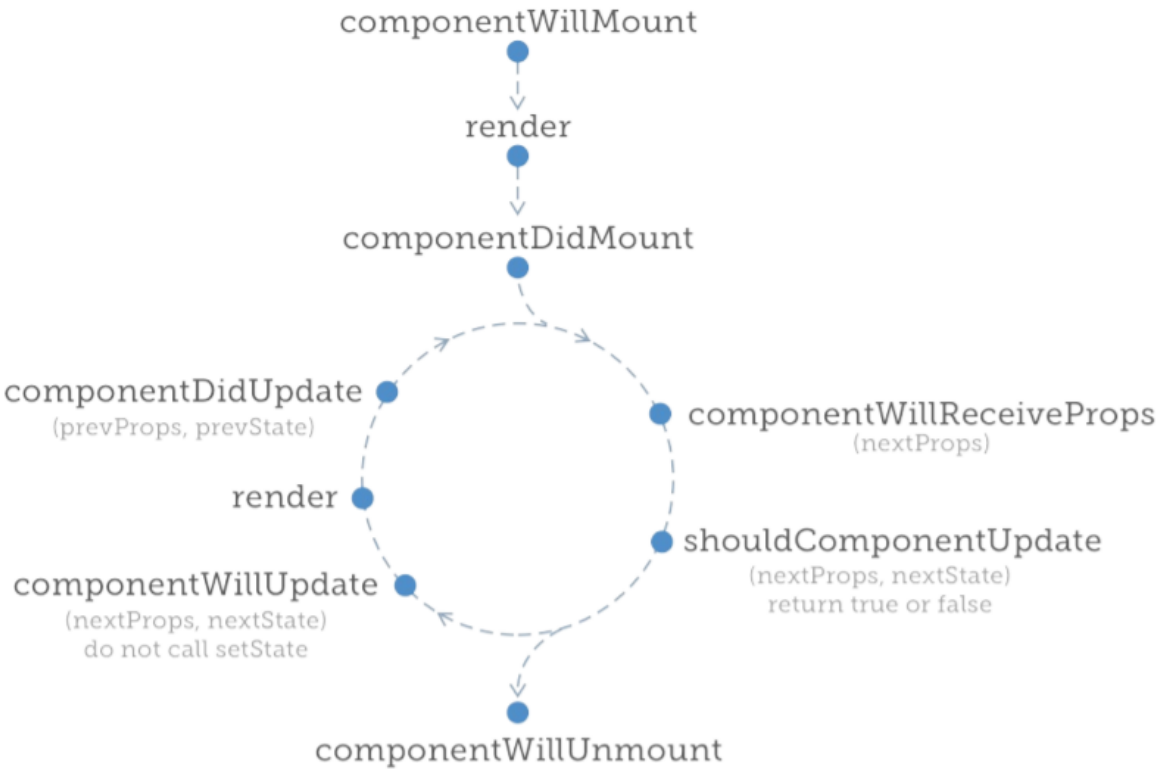
```
ReactDOM.render(element, container[, callback])
```

ReactDOM.render () използва DOM node във HTML, за да го замени с вашия JSX. Очаква два аргумента: Първият аргумент е JSX, който се визуализира. Вторият аргумент указва мястото, където Реакт се „закача“ за приложението.

```
ReactDOM.render(
  <h1>Hello React World</h1>,
  document.getElementById('root')
);
```

React компонент има следните области:

- Данни: Идват от някъде и се визуализират от компонента.
- Жизнен цикъл: Всеки компонент има няколко „метода на жизнения цикъл“, които можете да замените, за да стартирате код в определени моменти от процеса. Например една фаза от жизнения цикъл е когато компонент е на път да бъде изобразен.



- Събития: Това е кодът, който пишем за отговор на потребителските взаимодействия.
- JSX: Това е синтаксисът на React компонентите, използвани за описване на UI структури.

```
const element = <h1>Здравей, свят!</h1>;
```

Представяне(Rendering) чрез JSX

JSX е XML / HTML синтаксисът за маркиране, който е вграден във вашия JavaScript код и използван за деклариране на вашите React компоненти. В най-ниското ниво, ще използвате HTML маркиране, за да опишете частите от потребителския си интерфейс.

```
import React from 'react';
import { render } from 'react-dom';

render(
 <p>
 Hello, <strong>JSX</strong>
 </p>,
 document.getElementById('root')
);
```

Функцията render () приема JSX като първи аргумент и го изобразява на подадения DOM като втори аргумент. В този пример показва абзац с някакъв удебелен текст вътре. Функцията render казва на React да вземе вашият JSX код и да го трансформира в JavaScript, който актуализира потребителския интерфейс по възможно най-ефективния начин. Ето как React позволява да декларирате структурата на вашия потребителски интерфейс, без да се налага да мислите за изпълнение на стъпките за актуализиране на елементи на екрана. React поддържа стандартните HTML тагове, които бихте намерили на всяка HTML страница. За разлика от статичния HTML, React има уникални конвенции, които трябва да се спазват при използване. Когато изобразявате HTML тагове в JSX, трябва да използвате малки букви за името на маркера. Имената на маркерите са чувствителни към малки и малки букви, а не-HTML елементите се пишат с главни букви.

```
import React from 'react';
import { render } from 'react-dom';

import MySection from './MySection';
import MyButton from './MyButton';

render(
  <MySection>
    <MyButton>My Button Text</MyButton>
  </MySection>,
  document.getElementById('root')
);
```

Използването на JSX е полезно за описване на UI структури, които имат родител-дете взаимоотношения. Например, <li> таг е полезен само като дете на <ul> таг или <ol>.

Примера включва 2 React компонента: MySection и MyButton. Сега, ако погледнете JSX маркировката, ще забележите, че <MyButton> е дете на <MySection>.

Също така ще забележите, че компонентът MyButton приема текст като свое дъщерно устройство, вместо други JSX елементи.

```
import React, { Component } from 'react';

export default class MySection extends Component {
  render() {
    return (
      <section>
        <h2>My Section</h2>
        {this.props.children}
      </section>
    );
  }
}
```

MySection компонентът показва стандартен <section> HTML елемент, заглавие и след това {this.props.children}. Това е последното парче, което позволява на компонентите да имат достъп до вложени елементи или текст и да ги изобразява.

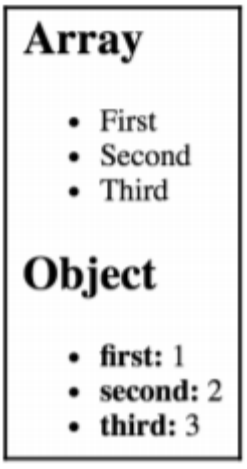
```
import React, { Component } from 'react';

export default class MyButton extends Component {
  render() {
    return <button>{this.props.children}</button>;
  }
}
```

Нека разгледаме и компонента MyButton. Този компонент използва точно същия модел като MySection; вземете {this.props.children} стойност и я обградете с маркиране. React се справя с детайлите.

В този пример текстът на бутона е дъщерно устройство на MyButton, което от своя страна е дете на MySection.

```
import React from 'react';
import { render } from 'react-dom';

const array = ['First', 'Second', 'Third'];

const object = {
  first: 1,
  second: 2,
  third: 3
};

render(
  <section>
    <h1>Array</h1>
    <ul>
      {array.map(i => (
        <li key={i}>{i}</li>
      ))}
    </ul>

    <h1>Object</h1>
    <ul>
      {Object.keys(object).map(i => (
        <li key={i}>
          <strong>{i}: </strong>
          {object[i]}
        </li>
      ))}
    </ul>
  </section>,
  document.getElementById('root')
);
```

Първата колекция е масив, наречен array, запълнен с низови стойности. Преминавайки надолу към маркировката JSX, можете да видите извикването на array.map(), което връща нов масив. Mapping функцията всъщност връща JSX елемент (<li>), което означава, че всеки елемент в масивът вече е



представен в маркировката.

React 16 представя концепцията за JSX фрагменти. Фрагментите са начин да се групират заедно парчета маркировка, без да се налага да добавяте ненужна структура към вашата страница. Например, често срещан подход е да се върне съдържание на React компонент, увито в елемент. Този елемент няма реална цел и добавя безпорядък към DOM.

```
import React, { Component, Fragment } from 'react';

class WithFragments extends Component {
  render() {
    return (
      <Fragment>
        <h1>With Fragments</h1>
        <p>Doesn't have any unused DOM elements.</p>
      </Fragment>
    );
  }
}

export default WithFragments;
```

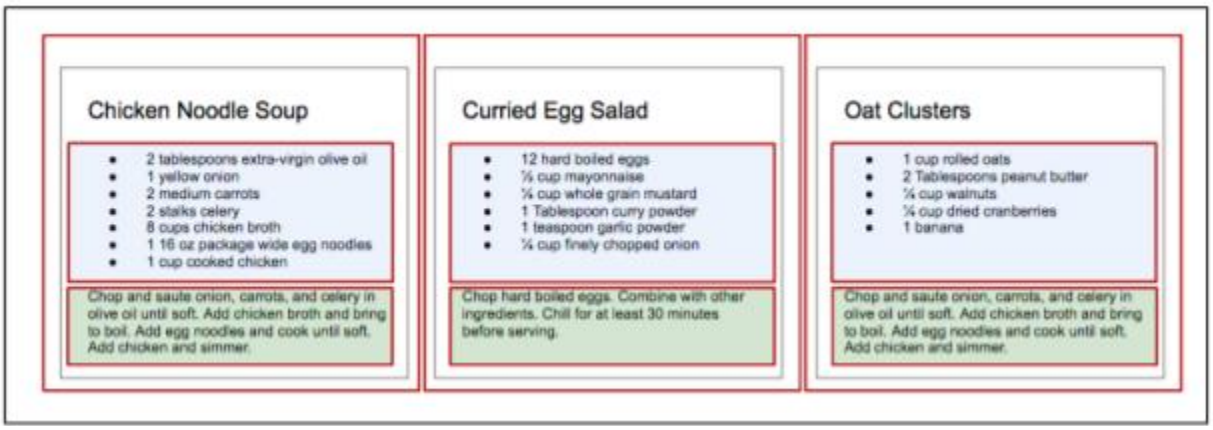
Вместо да увива съдържанието на компонента в ,

```
import React, { Component } from 'react';

class WithoutFragments extends Component {
  render() {
    return (
      <div>
        <h1>Without Fragments</h1>
        <p>
          Adds an extra <code>div</code> element.
```

се използва елементът <Fragment>. Това е специален тип елемент, който показва, че трябва да бъдат изобразени само неговите деца. Стенографичен начин за изразяване на фрагменти в JSX: <>My Content</>.

Components

Независимо от размер, съдържание или технологии , потребителският интерфейс е съставен от части. Бутони. Списъци. Заглавия. Всички те, когато бъдат поставени заедно, съставят потребителски интерфейс. Помислете за приложение с три различни рецепти. Данните са различни във всяко поле, но частите, необходими за създаване на рецепта, са еднакви. В React описваме всяка от тези части като компонент. Компонентите ни позволяват да се използва повторно една и съща структура и след това да попълним тези структури с различни набори от данни. Когато обмисляте потребителски интерфейс, който искате да изградите с React, потърсете връзки за разбиване на вашите елементи на многократно използваеми парчета. Например рецептите в има заглавие, списък на съставките и инструкции. Всички те са част от по-голяма рецепта или компонент на приложението. Можем да създадем компонент за всяка от подчертаните части:
съставки, инструкции и т.н.



Помислете колко мащабируемо е това. Ако искаме да покажем една рецепта, нашият компонент-структурата ще подкрепи това. Ако искаме да покажем 10 000 рецепти, просто ще създадем 10 000 нови екземпляра на този компонент. Ще създадем компонент, като напишем функция, която връща неподреден списък:

```
function IngredientsList() {
 return React.createElement(
 "ul",
 { className: "ingredients" },
 React.createElement("li", null, "1 cup unsalted butter"),
 React.createElement("li", null, "1 cup crunchy peanut butter"),
 React.createElement("li", null, "1 cup brown sugar"),
 React.createElement("li", null, "1 cup white sugar"),
 React.createElement("li", null, "2 eggs"),
 React.createElement("li", null, "2.5 cups all purpose flour"),
 React.createElement("li", null, "1 teaspoon baking powder"),
 React.createElement("li", null, "0.5 teaspoon salt")
 );
}
ReactDOM.render(
 React.createElement(IngredientsList, null, null),
 document.getElementById("root")
);
```

Името на компонента е Списък на съставките и функцията извежда елементи, които изглежда така:

```
<IngredientsList>
  <ul className="ingredients">
    <li>1 cup unsalted butter</li>
    <li>1 cup crunchy peanut butter</li>
    <li>1 cup brown sugar</li>
    <li>1 cup white sugar</li>
    <li>2 eggs</li>
    <li>2.5 cups all purpose flour</li>
    <li>1 teaspoon baking powder</li>
    <li>0.5 teaspoon salt</li>
  </ul>
</IngredientsList>
```

Това е доста готино, но сме кодирали твърдо тези данни в компонента. Сега ще предадем данни в този компонент като свойства:

```
const secretIngredients = [
  "1 cup unsalted butter",
  "1 cup crunchy peanut butter",
  "1 cup brown sugar",
  "1 cup white sugar",
  "2 eggs",
  "2.5 cups all purpose flour",
  "1 teaspoon baking powder",
  "0.5 teaspoon salt"
];
```

```
function IngredientsList() {
  return React.createElement(
    "ul",
    { className: "ingredients" },
    items.map((ingredient, i) =>
      React.createElement("li", { key: i }, ingredient)
    )
  );
}

ReactDOM.render(
  React.createElement(IngredientsList, { items: secretIngredients }, null),
  document.getElementById("root")
);
```

Често използвани методи

render()

Методът render () е единственият задължителен метод в класов компонент.

Когато бъде извикан, той трябва да разгледа this.props и this.state и да върне един от следните типове:

- **React elements**

Обикновено се създава чрез JSX. Например и <MyComponent /> са елементи на React, които инструктират React да изобрази DOM node или друг дефиниран от потребителя компонент.

- **Arrays and fragments**

- **Portals**

- **String and numbers**

- **Booleans or `null`**

Функцията render () трябва да е чиста, което означава, че тя не променя състоянието на компонента, тя връща същия резултат всеки път, когато е извикана, и не взаимодейства директно с браузъра. render () няма да бъде извикан, ако shouldComponentUpdate () върне false.

constructor()

```
constructor(props)
```

Ако не инициализирате състояние не е необходимо да внедрявате конструктор във вашия React компонент. Конструкторът за React компонент се извиква, преди той да бъде представен. Когато внедрявате конструктора за подклас React.Component, трябва да извикате super (props) преди всеки друг оператор. В противен случай this.props ще бъде недефиниран в конструктора, което може да доведе до грешки. Обикновено в React конструкторите се използват само за:

1) Инициализиране на локално състояние чрез присвояване на обект на this.state.
2) Обвързване на методи за обработка на събития към екземпляр.

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Конструкторът е единственото място, където трябва да присвоите директно this.state. Във всички останали методи вместо това трябва да използвате this.setState ().

Никога не копирате в свойствата в състоянието! Това е често срещана грешка:

```
constructor(props) {
 super(props);
 // Don't do this!
 this.state = { color: props.color };
}
```

Проблемът е, че е едновременно ненужно (вместо това можете да използвате директно this.props.color) и създава грешки (актуализациите на props.color няма да бъдат отразени в състоянието).

componentDidMount()

componentDidMount () се извиква веднага след монтиране на компонент (вмъкване в дървото). Инициализацията, която изисква DOM nodes, трябва да отиде тук. Това е методът, който ще ви помогне ако трябва да заредите данни от сървър.

componentDidUpdate()

```
componentDidUpdate(prevProps, prevState, snapshot)
```

Извиква се веднага след актуализирането. Този метод не е извикан при първоначално изобразяване. Използвайте това като възможност за работа с DOM, когато компонентът е актуализиран. Това също е добро място за мрежови заявки, стига да сравнявате текущите свойства с предишни.

```
componentDidUpdate(prevProps) {
  // Typical usage (don't forget to compare props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

componentWillUnmount()

componentWillUnmount () се извиква непосредствено преди демонтирането и унищожаването на компонент. Не трябва да извиквате setState (), защото компонентът никога няма да бъде повторно изобразен.

Примери с т.нар. Browser APIs

Timers

setTimeout

Executes a function after waiting a specified number of milliseconds.

```
const myTimeout = setTimeout(() => {
  console.log('Executed after 2 seconds!');
}, 2000);
```

clearTimeout

Clears a timeout set with *setTimeout*, preventing the function from executing.

```
clearTimeout(myTimeout);
```

setInterval

Executes a function repeatedly, with a fixed time delay between each call.

```
const myInterval = setInterval(() => {
  console.log('This will log every 2 seconds');
}, 2000);
```

clearInterval

Clears an interval set with *setInterval*, stopping the function from executing repeatedly.

```
clearInterval(myInterval);
```

Storage

localStorage

Allows web applications to store key-value pairs in a web browser persistently across sessions.

```
localStorage.setItem('myKey', 'myValue');
const retrievedValue = localStorage.getItem('myKey');  for 'myKey'
localStorage.removeItem('myKey');
localStorage.clear();
```

sessionStorage

Allows web applications to store key-value pairs in a web browser for a single session.

```
sessionStorage.setItem('sessionKey', 'sessionValue');
const sessionValue = sessionStorage.getItem('sessionKey');
sessionStorage.removeItem('sessionKey');
sessionStorage.clear();
```

IndexedDB

A low-level API for storing structured data, including large datasets.

```
const openRequest = indexedDB.open("myDatabase", 1);

openRequest.onupgradeneeded = event => {
  const db = event.target.result;
  const store = db.createObjectStore("myStore");
};

openRequest.onsuccess = event => {
  const db = event.target.result;
  const transaction = db.transaction("myStore", "readwrite");
  const store = transaction.objectStore("myStore");
  store.add("value", "key");
};
```

fetch

A modern way to make network requests, replacing the older *XMLHttpRequest*.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
fetchData();
```

GET Request with Custom Headers

```
async function fetchDataWithHeaders() {
  const headers = new Headers({
    'Content-Type': 'application/json',
    'Authorization': 'Bearer YOUR_TOKEN'
  });

  try {
    const response = await fetch('https://api.example.com/data', { headers });
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}
fetchDataWithHeaders();
```

POST Request

```
async function postData() {
  const payload = {
    key1: 'value1',
    key2: 'value2'
  };

  try {
    const response = await fetch('https://api.example.com/data', {
```

```
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(payload)
  });
  const result = await response.json();
  console.log(result);
 } catch (error) {
  console.error('Error posting data:', error);
 }
}
postData();
```

Error Handling by Checking the Response Status

```
async function fetchAndHandleErrors() {
 const response = await fetch('https://api.example.com/data');
 if (!response.ok) {
  throw new Error(`HTTP error! Status: ${response.status}`);
 }
 const data = await response.json();
 console.log(data);
}


fetchAndHandleErrors()
```

Uploading a File

```
async function uploadFile(file) {
  const formData = new FormData();
  formData.append('file', file);

  try {
    const response = await fetch('https://api.example.com/upload', {
      method: 'POST',
      body: formData
    });
    const result = await response.json();
    console.log(result);
  } catch (error) {
    console.error('Error uploading file:', error);
  }
}
// Assuming there's an input element with type "file" and ID "myFile"
const fileInput = document.querySelector('#myFile');
fileInput.addEventListener('change', event => {
  const file = event.target.files[0];
  uploadFile(file);
});
```

Other Web APIs

WebSocket API

Enables real-time communication between a client and server.

```
const socket = new WebSocket('ws://example.com/socket');

socket.onmessage = (event) => {

 console.log(event.data);

};

socket.send('Hello Server!');
```

There are also web APIs for WebRTC (Web Real-Time Communications) API, which Provides real-time audio, video, and data communication.

Service Workers

Service workers are a type of web worker.

They're essentially JavaScript scripts that run in the background, separate from a web page, providing features without needing to have the page open or even being connected to the internet.

They act as a proxy between a web application and the network, allowing for features like caching, push notifications, and background data syncing.

Core Features of Service Workers

- **Offline Caching**: By intercepting network requests and serving cached results, service workers allow web applications to function offline.
- **Push Notifications**: Service workers can handle push notifications from servers, even when the application is not open in a browser.
- **Background Data Sync**: They can synchronize data in the background, ensuring that user data is always up-to-date.

Example in JavaScript

```
async function registerServiceWorker() {

 if ('serviceWorker' in navigator) {

  try {

   const registration = await navigator.serviceWorker.register('/sw.js');

   console.log('Service Worker registered with scope:', registration.scope);
```

```
  } catch (error) {
    console.log('Service Worker registration failed:', error);
  }
 }
}
registerServiceWorker();
```

Caching Files and Web Requests

Service workers use the Cache API to cache files and web requests. Here's a basic example of how to cache assets using async/await:

```
const CACHE_NAME = 'my-site-cache';
const urlsToCache = [
 '/',
 '/styles/main.css',
 '/script/main.js'
];

async function handleInstall(event) {
 const cache = await caches.open(CACHE_NAME);
 await cache.addAll(urlsToCache);
}

async function handleFetch(event) {
 const cachedResponse = await caches.match(event.request);
 if (cachedResponse) {
  return cachedResponse;  // Return cached response
```

```
 }
 return fetch(event.request);  // Fetch from the network
}


self.addEventListener('install', (event) => {
 event.waitUntil(handleInstall(event));
});


self.addEventListener('fetch', (event) => {
 event.respondWith(handleFetch(event));
});
```

Web Audio API

Allows for the processing and synthesizing of audio in web applications.

```
const audioContext = new AudioContext();
const oscillator = audioContext.createOscillator();
oscillator.type = 'sine';
oscillator.frequency.setValueAtTime(440, audioContext.currentTime);
oscillator.connect(audioContext.destination);
oscillator.start();
oscillator.stop(audioContext.currentTime + 1);
```

Canvas API

Enables drawing graphics via JavaScript and the HTML *<canvas>* element.

```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
```

```
ctx.fillStyle = 'red';

ctx.fillRect(10, 10, 100, 100);
```

Geolocation API

Allows users to share their location with web applications.

```
if ("geolocation" in navigator) {

 navigator.geolocation.getCurrentPosition(position => {

  const { latitude, longitude } = position.coords

  console.log(`Latitude: ${latitude}, Longitude: ${longitude}`);

 });

}
```

Notifications API

Enables web pages to display notifications to users.

```
async function showNotification() {

 const permission = await Notification.requestPermission();

 if (permission === 'granted') {

  const notification = new Notification('Hello!',

   { body: 'This is a notification.' }

  );

 }

}

showNotification();
```

Intersection Observer API

Provides a way to asynchronously observe changes in the intersection of a target element with its parent or the viewport.

```
const observer = new IntersectionObserver(entries => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      console.log('Element is in view!');
    }
  });
});
const target = document.querySelector('.targetElement');
observer.observe(target);
```

MediaStream (getUserMedia)

Accesses the user's camera and microphone.

```
async function getMediaStream() {
  try {
    const stream = await navigator.mediaDevices.getUserMedia({ video: true, audio: true });
    const videoElement = document.querySelector('video');
    videoElement.srcObject = stream;
  } catch (error) {
    console.error('Error accessing media devices.', error);
  }
}
getMediaStream();
```

Performance API

Allows measurement of the performance of web pages and web apps.

```
const performanceEntries = performance.getEntriesByType("resource");
```

```
performanceEntries.forEach(entry => {
  console.log(`Name: ${entry.name}, Duration: ${entry.duration}`);
});
```