НЕТОЛОГИЯ

PROMISES, ASYNC/AWAIT

ЛЕКТОР / ДОЛЖНОСТЬ

ЛЕКТОР

должность

ПЛАН ЗАНЯТИЯ

1. Асинхронный код
2. Promises
3. async/await
4. Тестирование асинхронного кода

СИНХРОННЫЙ КОД

JS исполняет приложения в одном потоке*, т.е. может выполнять одну операцию в единицу времени.

Это нас ограничивает с точки зрения создания современных приложений, особенно связанных с обработкой различных событий.

СИНХРОННЫЙ КОД

```
const response = getResponse(...);
const data = processResponse(...);
```

Если оба вызова представляют собой быстрые операции, то никаких проблем не возникает.

ДЛИТЕЛЬНЫЕ ОПЕРАЦИИ

Оба вызова могут представлять собой достаточно длительные операции.

Если в это время не обрабатывать другие события, то получится «очередь», т.к. JS будет ждать завершения каждого вызова.

МНОГОПОТОЧНОСТЬ

Многие языки программирования предлагают инструменты для создания и управления несколькими потоками выполнения.

Традиционно этот раздел считается одним из самых сложных и подверженных ошибкам.

ЗАДАЧА

Перед нами стоит следующая задача: загрузить аналитические данные с сервера и произвести обработку данных на стороне пользователя, выдав ему аналитический отчёт.

Почему на стороне пользователя, а не на сервере?

> Не всегда у нас есть возможность получить доступ к серверу. Возможно, мы используем API Вконтакте для получения этих данных.
> А разработчики Вконтакте вряд ли вам дадут написать на их серверах свою аналитическую функцию ☺

АСИНХРОННЫЙ КОД

АСИНХРОННЫЙ КОД

Поэтому в JS присутствует механизм выполнения асинхронного кода, который позволяет «упростить» обработку подобных длительных операций, не прибегая к использованию примитивов многопоточности.

CALLBACKS

Callbacks — подход, при котором вместо ожидания какого-либо события (например, завершения операции) либо обработки какого-то элемента, мы передаём функцию (callback), которую нужно выполнить после наступления этого события, либо для обработки этого элемента.

```
function getResponse(args, callback) {
  // где-то внутри функции getResponse
  const response = ...;

  callback(response);
}

getResponse(..., (response) => { // наш callback
  ...
});
```

Важно: понятие callback используется не только в контексте асинхронности. Callback является функцией, передаваемой в качестве аргумента другой функции, для вызова внутри этой функции. Для Built-in объектов это выполнение каких-либо операций (например, для Array — поиск, сравнение и т.д.).

CALLBACKS

Определение на MDN звучит следующим образом: «A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action».

CALLBACK HELL

```
function getResponse(args, callback) {
  // где-то внутри функции getResponse
  const response = ...;

  callback(response);
}

function processResponse(args, callback) {
  // где-то внутри функции processResponse
  const data = ...;

  callback(data);
}

getResponse(..., (response) => { // наш первый callback
  processResponse(..., (data) => { // наш второй callback

  });
});
```

CALLBACK HELL

Нетрудно себе представить, что будет если вызовов у нас будет не 2,
а хотя бы 10.

Структура кода превращается в большое количество вложенных вызовов.

Для этого даже придумали отдельный термин — Callback Hell.

КЛЮЧЕВОЕ

Среда, в которой будет исполняться ваш JS-код (будь это браузер или Node.js) сама берёт на себя заботу по вызову вашего callback'а в нужный момент времени.

PROMISES

PROMISES

Использование Promise (обещания) — механизм, позволяющий упростить написание асинхронного кода и решить ряд проблем callback’ов.

PROMISES

```
function getResponse(args) {
  // Do something
  return new Promise((resolve, reject) => {
    ...
  });
}
```

Теперь функции не принимают callback для вызова, а возвращают объект класса `Promise`, который и будет играть ключевую роль.

ИДЕЯ PROMISE

Ключевая идея `Promise` — это объект, который может находится всего в трёх состояниях:

- `pending`
- `fulfilled`
- `rejected`

И единственное, что может произойти с `Promise` — это переход из состояния `pending` в состояние `fullfilled` или `rejected`.

Произойти этот переход может только один раз.

ИДЕЯ PROMISE

Поскольку функция, выполняющая асинхронную операцию не может вернуть значение этой операции, она возвращает `Promise`, который и «заворачивает» результат выполнения этой операции.

СОЗДАНИЕ PROMISE

```
function getResponse(args) {
  // Do something
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('value');
    }, 500);
  });
}

const responsePromise = getResponse(args);
```

`resolve`, `reject` – функции, вызываемые по завершении операции и переводящие `Promise` в состояние `fullfiled` или `rejected`, соответственно.

THEN

Метод, принимающий callback, который должен вызваться в случае перехода `Promise` в состояние `fullfilled`:

```
const responsePromise = getResponse(args);
responsePromise.then((response) => {
  ...
});
```

ОБРАБОТКА ОШИБОК

При переходе `Promise` в состояние `rejected` вызывается callback, указанный вторым параметром в методе `then`:

```
1  const responsePromise = getResponse(args);
2  responsePromise.then((response) => {
3    ...
4  }, (error) => { // callback for rejected
5    ...
6  });
```

CATCH

Метод, принимающий callback, который должен вызваться в случае перехода `Promise` в состояние `rejected` или выбрасывания исключения (если оно произошло в коде `then`):

```
const responsePromise = getResponse(args);
responsePromise.catch((error) => {
  ...
});
```

THEN + CATCH

```
1  const responsePromise = getResponse(args);
2  responsePromise.then((response) => {
3    ...
4  }).catch((error) => {
5  // calback for `rejected` и обработчик ошибок в `then`
6
7  });
```

FINALLY

Метод, принимающий callback, который должен вызваться в случае перехода `Promise` в состояние `fullfilled` или `rejected` (вне зависимости от того, в какое состояние перешёл `Promise`):

```
const responsePromise = getResponse(args);
responsePromise.then((response) => {
  ...
}).catch((error) => {
  ...
}).finally(() => {
  // final actions
});
```

Используется для исключения дублирования кода в `then` и `catch`

PROMISIFICATION

Использование `Promise` потребовало переписывания старого кода.

Переписывание старого кода (без `Promise`) с использованием `Promise` обозначают термином **Promisification**

ЦЕПОЧКИ PROMISE

`Promise` можно объединять в цепочки, если `then` возвращает тоже `Promise`:

```
function getResponse(args) {
  // Do something
  return new Promise((resolve, reject) => { ... });
}

function processResponse(response) {
  // Do something
  return new Promise((resolve, reject) => { ... });
}

getResponse(args).then((response) => {
  return processResponse(response);
}).then((data) => {
  // do something
}).catch((error) => {
  // handle error
}).finally(() => {
  // final handlings
})
```

ИТОГИ ПО PROMISE

Зачем нужны `Promise`, почему не делать всё на callback'ах?

> 1. Использование `Promise` упрощает работу с асинхронным кодом, помогая избежать Callback Hell
> 2. Современное API написано с использованием `Promise`, поэтому важно уметь использовать этот инструмент
> 3. `Promise` не отменяют callback'и — их всё равно придётся использовать

ИТОГИ ПО PROMISE

В каком порядке вызывается `then`, `catch`?

> В том, в котором записаны

```
const promise = getResponse();
promise.then((data) => {
  throw new Error(); // <- выброс ошибки, сработает следующий по блоку `catch`
}).catch((error) => {
  console.log('first error happened:');
})
.then((data) => {
  console.log(data); // <- сработает `then`
}).catch((error) => {
  console.log('second error happened:'); // <- не сработает
});

// undefined
```

Почему так?

THEN И CATCH

Методы `then` и `catch` тоже возвращают `Promise`, благодаря чему возможно построение цепочки `Promise`.

Особенности `then`:

- `then` возвращает `Promise`
- если из `then` возвращается значение, то оно автоматически заворачивается в `Promise`, который переходит в состояние `fullfilled`
- соответственно, если из `then` ничего не возвращается, то в `Promise` кладётся значение `undefined`
- если в `then` выбрасывается ошибка, то ошибка автоматически заворачивается в `Promise`, который переходит в состояние `rejected`
- если из `then` возвращается `Promise`, то последующие вызовы `then` и `catch` будут обрабатывать его состояние

THEN И CATCH

Методы `then` и `catch` тоже возвращают `Promise`, благодаря чему возможно построение цепочки `Promise`.

Особенности `catch`:

- `catch` возвращает `Promise`
- если из `catch` возвращается значение, то оно автоматически заворачивается в `Promise`, который переходит в состояние `fullfilled`
- соответственно, если из `catch` ничего не возвращается, то в `Promise` кладётся значение `undefined`
- если в `catch` выбрасывается ошибка, то ошибка автоматически заворачивается в `Promise`, который переходит в состояние `rejected`
- если из `catch` возвращается `Promise`, то последующие вызовы `then` и `catch` будут обрабатывать его состояние

СТАТИЧЕСКИЕ МЕТОДЫ PROMISE

Класс `Promise` содержит ещё ряд статических методов, предоставляющих удобную функциональность:

- `Promise.all(iterable)` – возвращает `Promise`, который переходит в состояние `fullfilled`, только если все `Promise` из `iterable` перешли в состояние `fullfiled` (либо в `iterable` не было `Promise`)
- `Promise.race(iterable)` – возвращает `Promise`, который переходит в состояние `fullfilled` или `rejected` как только любой из `Promise`, содержащихся в `iterable` переходит в `fullfilled` или `rejected`

ASYNC/AWAIT

ASYNC/AWAIT

С `Promise` всё достаточно хорошо, но есть ли механизмы ещё более упростить этот код?

Ключевые слова `async` / `await` позволяют сделать работу с `Promise` более удобной.

Рассмотрим сразу на примере.

ASYNC/AWAIT

```
const response = await getResponse(args);
const data = await processResponse(response);
```

И это вместо:

```
1  const promise = getResponse();
2  promise.then((response) => {
3    return processResponse(response);
4  .then((data) => {
5    // Do something
6  });
```

Удобно?

ОБРАБОТКА ОШИБОК И FINALLY

Здесь тоже всё хорошо, используем конструкцию `try...catch...finally`

```
try {
  const response = await getResponse(args);
  const data = await processResponse(response);
} catch {
  ...
} finally {
  ...
}
```

ASYNC

На использование `await` есть одно ключевое ограничение: `await` можно использовать только внутри `async` функций:

```
(async () => {
  try {
    const response = await getResponse(args);
    const data = await processResponse(response);
  } catch {
    ...
  } finally {
    ...
  }
})();
```

ASYNC

Ключевое слово `async` определяет, что функция выполняется асинхронно — т.е. всегда возвращает `Promise`, но может выглядеть, как стандартная функция.

Что значит *как стандартная функция*? Это значит, что если вы просто возвращаете из такой функции значение, то оно заворачивается в `Promise`.

Кроме того, вы можете использовать `await` внутри `async` функции, которое дожидается перехода `Promise` (`await` ставится перед `Promise`) в состояние `fullfilled` или `rejected`.

ДЛЯ ЧЕГО ЭТО?

Для упрощения структуры кода, сравним:

(на двух слайдах)

```
(async () => {
  try {
    // ожидаем изменение состояния `Promise`
    const response = await getResponse(args);
    // ожидаем изменение состояния `Promise`
    const data = await processResponse(response);
  } catch {
    ...
  } finally {
    ...
  }
})();
```

ДЛЯ ЧЕГО ЭТО?

(продолжение)

```
getResponse(args).then((response) => {
  return processResponse(response);
}).then((data) => {
  // do something
}).catch((error) => {
  // handle error
}).finally(() => {
  // final handlings
})
```

Первый вариант намного более лаконичный за счёт того, что позволяет избежать нагромождения `then`, `catch`.

ASYNC/AWAIT

Почему бы тогда совсем не отказаться от `Promise`?

> Потому что в основе работы `async` / `await` лежат `Promise`.
> `async` / `await` позволяет нам лишь удобнее с ними работать.

BABEL

```
$ npm install @babel/polyfill
```

В `.babelrc`:

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "useBuiltIns": "usage"
      }
    ]
  ]
}
```

ТЕСТИРОВАНИЕ АСИНХРОННОГО КОДА

ТЕСТИРОВАНИЕ АСИНХРОННОГО КОДА

Jest предлагает для всех рассмотренных нами вариантов (callback'и, `Promise`, `async` / `await`) удобные методы для тестирования. Рассмотрим их.

CALLBACKS

```
1  test('should call our callback', (done) => { // <- специальный аргумент
2    getData((data) => {
3      expect(data).toBe(...);
4      done(); // <- указание на завершение теста
5    });
6  });
```

`done` – функция, вызова которой Jest будет ожидать в течение времени, определённого `jest.setTimeout` (по умолчанию – 5 секунд).

Если вызова не будет, получим FAIL

```
FAIL test/data.test.js (5.352s)
  ✕ should call our callback (5015ms)

  ● should call our callback

    Timeout - Async callback was not invoked within the 5000ms timeout specified by jest.setTimeout.

      1 | import { getData } from '../src/js/demo';
      2 |
    > 3 | test('should call our callback', (done) => { // <- специальный аргумент
        | ^
      4 |   getData((data) => {
      5 |     expect(data).toBe('...');
      6 |     done(); // <- указание на завершение теста

      at Spec (node_modules/jest-jasmine2/build/jasmine/Spec.js:85:20)
      at Object.test (test/data.test.js:3:1)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
```

PROMISE И ASYNC/AWAIT

При работе с `Promise` и `async` / `await` достаточно использовать асинхронные тестовые функции (и работать как обычно):

```
test('should work with promise and async/await', async () => { // <- async
  const data = await getData();
  expect(data).toBe(...);
});
```

ERROR HANDLING

```
1  test('should handle errors', async () => { // <- async
2    // <- сообщаем Jest, что у нас один assert, который нужно проверить
3    expect.assertions(1);
4    try {
5      const data = await getData();
6    } catch (e) {
7      expect(e).toBe(...);
8    }
9  });
```

ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей:

1. Асинхронный код
2. Promises
3. async/await
4. Тестирование асинхронного кода

НЕТОЛОГИЯ

Спасибо за внимание!

Время задавать вопросы ☺

ЛЕКТОР