Урок 3

Асинхронные запросы

Основы асинхронного JavaScript. AJAX, JSON и Promises.

HTTP-запросы

AJAX – асинхронный JavaScript

Объект XMLHttpRequest

JSON и XML

Callback – функция обратного вызова

Promise

Практика

Практическое задание

Дополнительные материалы

Используемая литература

Когда Интернет только появился, всё общение между компьютером и удалённым сервером происходило в едином потоке: после отправки запроса страница блокировалась до тех пор, пока сервер не ответит, а как только приходил ответ, страница перезагружалась. Никакие операции нельзя было выполнять в фоновом режиме. Чтобы решить эту проблему, был придуман AJAX.

HTTP-запросы

Чтобы стандартизировать общение между компьютерами, в Интернете используются протоколы. **Протокол** – это набор правил, описывающих, в каком виде передаются и принимаются данные. Чаще всего используется протокол HTTP. Типы запросов различаются в зависимости от назначения:

1. **GET**. Запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.
2. **POST**. Используется для отправки сущностей к ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.
3. **PUT**. Заменяет все текущие представления ресурса данными запроса.
4. **DELETE**. Удаляет указанный ресурс.
5. **HEAD**. Запрашивает ресурс так же, как и метод GET, но без тела ответа.
6. **CONNECT**. Устанавливает «туннель» к серверу, определённому по ресурсу.
7. **OPTIONS**. Используется для описания параметров соединения с ресурсом.
8. **TRACE**. Выполняет вызов возвращаемого тестового сообщения с ресурса.
9. **PATCH**. Используется для частичного изменения ресурса.

Чаще всего на практике используются запросы GET и POST.

AJAX – асинхронный JavaScript

AJAX – это набор инструментов для работы с сервером без перезагрузки страницы. Расшифровывается как Asynchronous JavaScript and XML. При использовании AJAX запрос к серверу не блокирует страницу и не перезагружает её. Получив ответ, приложение может обновить отдельный компонент страницы.

При AJAX-запросе общение с сервером может проходить в одном из нескольких форматов:

1. **JSON**. Самый популярный формат. Основан на JavaScript и внешне похож на запись обычного JavaScript-объекта.
2. **XML**. Расширенный язык разметки, похожий на HTML.
3. **HTML/text**. Обычная разметка в HTML или чистый неформатированный текст.
4. **Бинарные данные**. Набор байтов. Обычно используется для передачи файлов.

Объект XMLHttpRequest

Для отправки запросов в браузер встроен объект **XMLHttpRequest**. В Internet Explorer он называется иначе, поэтому при определении следует использовать такую конструкцию:

```
var xhr;

if (window.XMLHttpRequest) {
// Chrome, Mozilla, Opera, Safari
  xhr = new XMLHttpRequest();
} else if (window.ActiveXObject) {
```

```
// Internet Explorer
  xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Чтобы поймать момент, когда ответ от сервиса получен, можно воспользоваться свойством **onreadystatechange**:

```
xhr.onreadystatechange = function () {
// Этот код выполнится после получения ответа
}
```

Чтобы определить, куда отправить запрос, используется метод **.open()**:

```
xhr.open('GET', 'http://example.com', true);
// Первый параметр - тип запроса
// Второй параметр - адрес ресурса
// Третий параметр - указатель асинхронности
```

Если указатель асинхронности выставлен в **true**, то запрос не будет блокировать выполнение других скриптов на странице.

У каждого запроса можно определить таймаут – время, в течение которого мы ждём ответ:

```
xhr.timeout = 15000;
```

Если запрос не выполняется за отведенное время, срабатывает функция, переданная в поле **ontimeout**:

```
xhr.ontimeout = function () {
// Этот код выполнится, если превышено время ожидания
}
```

Отправить запрос можно методом **.send()**. В качестве аргумента можно передать тело запроса:

```
xhr.send();
```

Параметры запроса отделяются от тела запроса знаком **?** и соединяются между собой **&**.

```
xhr.open('GET', 'http://example.com?param1=foo&param2=bar', true);
```

Также при отправке запроса можно выставить заголовки. Заголовки содержат служебную информацию, чтобы серверу было проще обработать запрос. Делается это с помощью метода **setRequestHeader.** Например, при отправке POST-запроса нужно выставить тип данных:

```
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

Метод **onreadystatechange** на самом деле срабатывает, не только когда запрос выполнен, но и тогда,

когда меняется его статус. Статус запроса хранится в поле **readyState**:

- 0 – запрос не инициирован;
- 1 – загрузка;
- 2 – запрос принят;
- 3 – обмен данными;
- 4 – запрос выполнен.

Таким образом, если мы хотим выполнить действие после завершения запроса, то необходимо добавить проверку:

```
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4) {
// Этот код выполнится после выполнения запроса
  }
}
```

Ещё после завершения можно проверить статус запроса. Он хранится в **xhr.status**. По коду статуса можно понять, что ответил сервер:

- 200 – запрос выполнен успешно;
- 404 – запрашиваемый ресурс не найден;
- 500 – на сервере произошла ошибка.

Кодов запросов очень много, запоминать их не нужно.

Наконец, ответ сервера можно получить в виде обычной строки или XML:

```
xhr.responseText    // Текстовая строка
xhr.responseXML     // XMLDocument
```

JSON и XML

Для передачи сообщений в запросе чаще всего используется один из двух самых популярных форматов – JSON и XML.

XML (eXtensible Markup Language) – расширяемый язык разметки, по синтаксису напоминающий HTML. Он появился ещё в конце 90-х и применялся для стандартизации общения через Интернет.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE recipe>
<recipe name="хлеб" preptime="5min" cooktime="180min">
   <title>
      Простой хлеб
   </title>
   <composition>
      <ingredient amount="3" unit="стакан">Мука</ingredient>
      <ingredient amount="0.25" unit="грамм">Дрожжи</ingredient>
      <ingredient amount="1.5" unit="стакан">Тёплая вода</ingredient>
   </composition>
   <instructions>
```

```
    <step>
       Смешать все ингредиенты и тщательно замесить.
    </step>
    <step>
       Закрыть тканью и оставить на один час в тёплом помещении.
    </step>
    <step>
       Замесить ещё раз, положить на противень и поставить в духовку.
    </step>
  </instructions>
</recipe>
```

JSON (JavaScript Object Notation) – формат передачи данных, который использует запись, внешне очень похожую на обычный JavaScript-объект. Разница лишь в том, что все ключи обрамлены кавычками.

```
{
  "foo": "bar",
  "marvin": 42,
  "dharma": [4, 8, 15, 16, 23, 42],
  "avg": {
    "cap": "Steve Rogers"
  }
}
```

Формат JSON значительно удобнее для чтения, чем XML, поэтому сейчас он популярнее. К тому же для упрощения работы с JSON в JavaScript есть два метода – **JSON.parse()** и **JSON.stringify()**, переводящие JSON в объект и объект в JSON соответственно.

```
var obj = { a: 1, b: '12' };
JSON.stringify(obj);                  // '{ "a": 1, "b": "12" }'
JSON.parse('{ "a": 1, "b": "12" }')   // { a: 1, b: '12' }
```

Callback – функция обратного вызова

Запрос к серверу – асинхронная операция. Это значит, что при его выполнении нам нужно знать, когда он завершится. Один из способов это контролировать – функция обратного вызова, или callback. Создадим простейшую асинхронную функцию с помощью **setTimeout**:

```
const async = (a) => {
  setTimeout(() => {
    const b = a + 1;
    return b;
  }, 200);
}
```

Если сейчас мы вызовем **async(5)**, то получим **undefined**. Чтобы это исправить, передадим в качестве второго аргумента функцию и вызовем её внутри **setTimeout**:

```
const async = (a, cb) => {
  setTimeout(() => {
    const b = a + 1;
    cb(b);
  }, 200);
}
```

Функция **cb()** и есть функция обратного вызова. По истечении таймаута она будет вызвана с аргументом, равным **b**:

```
async(5, (b) => {
  console.log(b);  // 6
});
```

Внутри callback-функции можно работать со значением, которое она вернула. Проблемы возникают, когда асинхронных много операций и одна зависит от другой. Например, мы получили данные от сервера, изменили, отправили обратно, получили новые и отправили на какой-нибудь другой сервер. Получится примерно такая конструкция:

```
getData(params, (data) => {
  sendData(newData, (res) => {
    sendToAnother(res, (resData) => {
      ...
    })
  })
})
```

Большое количество вложенных callback-функций на сленге называется «callback hell». Такая вложенность приводит к ошибкам, поэтому её нужно стараться избегать. К счастью, в стандарте ES2015 появилось решение – промисы.

Promise

Promise – это специальный объект, который находится в одном из трёх состояний: **pending** («ожидание»), **fulfilled** («выполнено успешно») и **rejected** («выполнено с ошибкой»). При создании промиса нужно указать два колбэка – для успешного выполнения и для ошибки:

```
const promise = new Promise((resolve, reject) => {
// Здесь пишем асинхронный код
// В случае успешного выполнения вызываем колбэк resolve()
// В случае ошибки вызываем reject()
});
```

После того, как промис объявлен, можно написать обработчик. Колбэки для успешного и ошибочного выполнения передаются в качестве аргументов метода **then()**:

```
promise.then(() => {
// Колбэк для resolve()
```

```
},
() => {
// Колбэк для reject()
});
```

Как правило, для работы с асинхронными операциями пишется функция, возвращающая промис. После вызова функции у её результата вызывается метод **then()**. Попробуем переписать наш пример с функцией **async**, используя промисы:

```
const async = (a) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (a) {
        const b = a + 1;
        resolve(b);
      } else {
        reject('Error');
      }
    }, 200);
  });
}
```

Теперь если аргумент в **async** есть, будет срабатывать колбэк **resolve,** а если нет – **reject**. Попробуем вызвать функцию **async()** с одинаковыми обработчиками, но разными аргументами:

```
async(5).then((b) => {
  console.log(b);      // Сработает первый колбэк и выведет в консоль 6
}, (error) => {
  console.log(error)
});

async().then((b) => {
  console.log(b);
}, (error) => {
  console.log(error)   // Сработает второй колбэк и выведет в консоль 'Error'
});
```

Если обрабатывать ошибки не нужно, колбэк **reject()** можно опустить.

Самое главное преимущество промисов – их можно объединять в цепочки вызовов. Колбэки будут выполняться один за другим, и это поможет избежать callback hell'a:

```
promise
  .then(() => { // Сначала выполнится этот обработчик })
  .then(() => { // Потом этот })
  .then(() => { // А потом вот этот });
```

Причём, если обработчик возвращает значение, оно прокидывается в следующий:

```
const giveMeNumber = (number) => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(number);
    }, 200)
  });
}

giveMeNumber(5)
  .then((number) => { return number + 1; })
  .then((number) => { return number + 10; })
  .then((number) => { console.log(number) }); // В консоль выведется 16
```

Практика

Сегодня мы научимся отправлять запросы к серверу нашего интернет-магазина. Сперва завернём **XMLHttpRequest** в функцию:

```
function makeGETRequest(url, callback) {
  var xhr;

  if (window.XMLHttpRequest) {
    xhr = new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
  }

  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      callback(xhr.responseText);
    }
  }

  xhr.open('GET', url, true);
  xhr.send();
}
```

Теперь можно отправлять запросы. Пока у нас нет реального API, воспользуемся заглушками:

- https://raw.githubusercontent.com/GeekBrainsTutorial/online-store-api/master/responses – адрес API;
- /catalogData.json – получить список товаров;
- /getBasket.json – получить содержимое корзины;
- /addToBasket.json – добавить товар в корзину;
- /deleteFromBasket.json – удалить товар из корзины.

Чтобы получить содержимое корзины, воспользуемся функцией **makeGETRequest** и заменим фейковые данные в методе **fetchGoods** на настоящие:

```
const API_URL =
```

```
'https://raw.githubusercontent.com/GeekBrainsTutorial/online-store-api/master/responses';

class GoodsList {
// ...
  fetchGoods() {
    makeGETRequest(`${API_URL}/catalogData.json`, (goods) => {
      this.goods = JSON.parse(goods);
    })
  }
// ...
}
```

Если посмотреть содержимое ответа от сервера, то можно увидеть такой JSON:

```
[
  {
    'id_product': 123,
    'product_name': 'Ноутбук',
    'price': 45600
  },
  {
    'id_product': 456,
    'product_name': 'Мышка',
    'price': 1000
  }
]
```

Как видно, название товара хранится в поле **product_name**, а не **title**. Поэтому заменим название поля в методе **render()** класса **GoodsItem** и класса **GoodsList**:

```
class GoodsItem {
// ...
  render() {
    return `<div
class="goods-item"><h3>${this.product_name}</h3><p>${this.price}</p></div>`;
  }
}

class GoodsList {
// ...
  fetchGoods() {
    makeGETRequest(`${API_URL}/catalogData.json`, (goods) => {
      this.goods = JSON.parse(goods);
    })
  }
  render() {
    let listHtml = '';
    this.goods.forEach(good => {
      const goodItem = new GoodsItem(good.product_name, good.price);
```

```
      listHtml += goodItem.render();
    });
    document.querySelector('.goods-list').innerHTML = listHtml;
  }
}
```

Обратите внимание, что **fetchGoods()** теперь асинхронная операция. Вызвать её как раньше не выйдет.

```
const list = new GoodsList();
list.fetchGoods();
list.render(); // render() сработает раньше, чем товары попадут в this.goods
```

Поэтому нам нужно дождаться, когда запрос выполнится, и только после этого запустить **render()** списка. Сделаем это с помощью колбэка:

```
class GoodsList {
// ...
  fetchGoods(cb) {
    makeGETRequest(`${API_URL}/catalogData.json`, (goods) => {
      this.goods = JSON.parse(goods);
      cb();
    })
  }
// ...
}

const list = new GoodsList();
list.fetchGoods(() => {
  list.render();
});
```

Практическое задание

1. Переделайте **makeGETRequest()** так, чтобы она использовала промисы.
2. Добавьте в соответствующие классы методы добавления товара в корзину, удаления товара из корзины и получения списка товаров корзины.
3. * Переделайте **GoodsList** так, чтобы **fetchGoods()** возвращал промис, а **render()** вызывался в обработчике этого промиса.

Дополнительные материалы

1. AJAX и COMET.
2. Ад обратных вызовов.

Используемая литература

1. Современный учебник Javascript.
2. Alberto Montalesi. The Complete Guide to Modern JavaScript.