Java Script

NEXT

GENIUS space

План

TCP/IP

HTTP/ HTTPS

CRUD

REST API

Fetch

async/await

try...catch

Протоколи передачі даних

Протокол — набір правил та угод, які використовуються під час передачі даних у мережі.
Перед тим як користувач побачить вміст сайту на екрані, браузер робить запит на сервер, щоб отримати цей вміст. HTML-файл, зображення, стилі, скрипти - кожен елемент надходить із сервера.

TCP/IP

Основний протокол мережі Internet. Це два протоколи тісно пов'язані між собою.
TCP (Transmission Control Protocol) - протокол керування передачею. Визначає, яким чином інформація має бути розбита на пакети та надіслана каналами зв'язку. TCP має пакети в потрібному порядку, а також перевіряє кожен пакет на наявність помилок при передачі.

IP (Internet Protocol) — кожен інформаційний пакет містить IP-адреси комп'ютера-відправника та комп'ютера-отримувача. Спеціальні комп'ютери, які називають маршрутизаторами, використовуючи IP-адреси, направляють інформаційні пакети в потрібну сторону, тобто до зазначеного в них одержувача.

HTTP

При HTTP-з'єднанні клієнт (браузер) робить запит на сервер, в результаті чого отримує відповідь. Сервер не може бути ініціатором запиту лише клієнт. Ця модель не підходить у випадку, коли серверу необхідно повідомити клієнт про якусь подію, наприклад нове повідомлення в чаті.

Кожне з'єднання фінальне, це означає, що сервер, після автету, забуває все про клієнт. У такій моделі кожне HTTP-з'єднання (відповідь чи запит) несе у собі велику кількість службової інформації про клієнта та сервер. Відкриття з'єднання та доставка службової інформації у кожному повідомленні впливають на швидкість запиту.

Це означає, що для розробки програм з роботою в реальному часі, технологія клієнт-сервер, за стандартним HTTP-протоколом, не підходить.

HTTPS - це надстойка над HTTP-протоколом, в якій всі повідомлення між клієнтом та сервером шифруються з метою підвищення безпеки. Забезпечує захист від атак, що базуються на прослуховуванні з'єднання. Дані передаються поверх криптографічних протоколів SSL чи TLS. Дефолтний TCP-порт - 443.
При спілкуванні через звичайне з'єднання HTTP всі дані передаються у вигляді тексту і можуть бути прочитані всіма, хто отримав доступ до з'єднання між клієнтом і сервером.
Якщо користувач робить покупки онлайн і заповнює форму замовлення, що містить інформацію про кредитну картку, їх фінансові дані набагато легше вкрасти, якщо вони передаються у вигляді тексту. З HTTPS дані будуть зашифровані, і хакер не зможе їх розшифрувати, тому що для розшифровки потрібний доступ до закритого ключа, який зберігається на сервері.
Інформація про клієнта, наприклад номери кредитних карток, зашифрована і не може бути перехоплена у розшифрованому вигляді.
Відвідувачі можуть підтвердити, що сайт безпечний, подивившись на іконку ліворуч від адресного рядка, захищені дії позначаються і іконкою замку.
Клієнти з більшою ймовірністю будуть довіряти і купувати з сайтів, що використовують HTTPS.

Веб-сокети (Web Sockets) – це технологія, яка дозволяє створювати та підтримувати неприривне, двонаправлене з'єднання між клієнтом (браузером) та сервером для обміну даними у реальному часі.

Все починається з HTTP-запиту, в якому є хедер upgrade, що повідомляє сервер про те, що клієнт хоче встановити з'єднання через веб-сокет. Вебсокет використовує ws:// схему (протокол) для HTTP і wss:// як еквівалент HTTPS.
Якщо сервер підтримує веб-сокет, підтверджується відповідь. Це називається handshake - рукостискання.

Після рукостискання, HTTP-з'єднання замінюється на WebSocket-з'єднання яке використовує той самий TCP/IP-протокол. Після чого клієнт та сервер можуть надсилати оповіщення без обмежень.
Дані передаються у вигляді повідомлень (message), кожне з яких складається з одного або кількох кадрів (frame). Кожен кадр супроводжується невеликим відрізком даних, що містять мінімальну службову інформацію, щоб на клієнті кадри можна було зібрати в правильному порядку.

CRUD - створення (create), читання (read), оновлення (update) та видалення (delete), чотири основні методи для взаємодії з ресурсами REST API.
У REST-архітектурі CRUD відповідає наступним HTTP-методам.

POST (create) – створити новий ресурс

GET (read) - отримати набір ресурсів або певний ресурс за ідентифікатором

PUT (update) або PATCH (update) – оновити ресурс за ідентифікатором

DELETE (delete) - видалити ресурс за ідентифікатором

Читання
Якщо пост має унікальне поле, як title в цьому випадку, його можна отримати безпосередньо через get():

```
title = "CRUD"
crud_post = Post.objects.get(title=title)
```

Такий запис наочніший і зручніший, ніж використання фільтра Post.objects.filter(title=title)[0].

Якщо посту із заголовком "CRUD" не опиниться в базі, ви отримаєте помилку:

```
models.DoesNotExist: Post matching query не існує.
```

Це звичайний виняток, його можна перехопити та обробити, хоча є й деякі особливості.

створення
Щоб створити нову посаду, є метод create. Ось так створюється новий пост

```
Post.objects.create(title="Заголовок нового посту", text="Текст нового посту")
```

Тепер він збережений і лежить у БД:

```
post = Post.objects.get(title="Заголовок нового поста")
print(post.title)
# Заголовок нового посту
```

Вилучення
Видаляти можна як окремі пости, так і цілі вибірки Queryset. Робиться це методом delete():

```
post = Post.objects.get(title="Новий заголовок")
post.delete() # Видалили пост з БД
all_posts = Post.objects.all()
all_posts.delete() # Видалили всі пости з БД
```

Редагування

Для редагування об'єкта посту потрібно спочатку витягнути його з БД. Наприклад, так:

```
post = Post.objects.get(title="Заголовок нового поста")
```

Тепер у нас є об'єкт посту, і ми можемо змінювати його атрибути. Так могло б виглядати редагування заголовка:

```
post = Post.objects.get(title="Заголовок нового поста")
post.title = "Від чого вимерли динозаври?"
```

Але це не спрацює. Зміни потрібно надіслати до бази даних, інакше вони пропадуть разом із завершенням програми. За збереження відповідає метод .save():

```
post = Post.objects.get(title="Заголовок нового поста")
post.title = "Від чого вимерли динозаври?"
post.save()
```

REST API

Сервер – комп'ютер із спеціальним програмним забезпеченням. Контролем роботи сервера займається системний адміністратор.

Дата-центр - спеціально обладнаний майданчик із цілодобовою технічною підтримкою для розміщених серверів. Забезпечує їхню безперебійну роботу.

Бекенд – програма, розташована на сервері і здатна обробити вхідні HTTP-запити і має набір готових дій на певні запити.

API (інтерфейс прикладного програмування) – набір чітко визначених правил зв'язку між різними програмними компонентами. Інтерфейс описує, що можна попросити програму зробити і що вийде в результаті.

REST (representational state transfer) — стиль бекенд-архітектури, що ґрунтується на наборі принципів, які описують як мережеві ресурси визначаються та адресуються.

REST API – бекенд побудований за принципом REST. Служить прошарком між веб-програмою та базою даних. Має стандартний інтерфейс звернення до ресурсів. Працюючи як веб-сайт, ми надсилаємо HTTP-запит із клієнта на сервер, а у відповідь замість HTML-сторінки отримуємо дані в JSON-форматі.

Формат запиту

REST-сервіс вимагає, щоб клієнт робив запит на додавання, вилучення або зміну даних.
Запит зазвичай складається з:
HTTP-метод - визначає, яку операцію виконувати.
Заголовок — дозволяє клієнту надсилати інформацію про запит.
Шлях – шлях до ресурсу. Доступні шляхи описуються у документації бекенда.
Тіло — додатковий блок запиту, який містить дані.

Заголовки

Заголовки містять службову інформацію, що стосується користувача або контенту запиту, і як клієнт збирається обробляти надану йому інформацію.
Наприклад тип контенту, який клієнт може обробити у відповіді від сервера (заголовок Accept) або який описує тип ресурсу, який клієнт відправляє серверу або сервер відправляє клієнту (заголовок Content-Type).
MIME-типи - варіанти типів контенту. Використовуються для вказівки вмісту зарослі та відповіді, складаються з типу та підтипу, які розділені косою рисою /.
Наприклад, текстовий файл, який містить HTML, буде описаний типом text/html. Якщо файл містить CSS, він буде описаний як text/css. Дані у форматі JSON будуть описані як application/json. Якщо клієнт чекає на text/css, а отримує application/json, він не зможе розпізнати і обробити контент.

Коди відповідей

На запит клієнта сервер відправляє відповідь. Відповідь містить код стану, щоб інформувати клієнта про результат операції. Коди поділяються на групи.

1XX - несуть інформаційне призначення

2XX - коди успішного проведення операції

3XX - описують все, що пов'язано з перенапрямком

4XX - вказують на помилки клієнта

5XX - вказують на помилки на стороні сервера

Не потрібно пам'ятати кожен код стану (їх багато), але необхідно знати найпоширеніші та їх значення.

200 (OK) – стандартна відповідь для успішних HTTP-запитів

201 (Created) - стандартна відповідь для HTTP-запиту, що призвела до успішного створення ресурсу

400 (Bad Request) - запит не може бути оброблений через неправильний синтаксис запиту або іншої помилки клієнта.

401 (Unauthorized) – для доступу до ресурсу потрібна авторизація.

403 (Forbidden) - клієнт не має дозволу на доступ до цього ресурсу.

404 (Not Found) – в даний час ресурс не знайдений. Можливо, він був вилучений чи ще не існує.

Fetch API

Fetch API — надає інтерфейс, набір методів та властивостей для відправлення, отримання та обробки ресурсів від сервера.
Це XMLHttpRequest нового покоління. Він надає покращений інтерфейс для складання запитів до сервера та побудований на обіцянках (promise).
fetch(url, options)
url - обов'язковий шлях до даних, які ви хочете отримати.
options — необов'язковий об'єкт налаштувань запиту. Містить службову інформацію: метод (за умовчанням GET), заголовки, тіло тощо.
Повертає проміс, який містить відповідь сервера.

method - рядок з назвою методу запиту: "GET", "HEAD", "POST".
headers - об'єкт заголовок запиту.
body - тіло запиту.
mode – режим крос-доменості: "same-origin", "no-cors", "cors".
credentials – чи відправляти куки: "same-origin", "include", "omit".
cache – режим кешування: "default", "no-store", "reload", "no-cache", "force-cache", "only-if-cached"
redirect – режим переадресації: "follow" або "error".

fetch() метод об'єкта window який дозволяє робити AJAX запити на основі Promise.

fetch() повертає Promise який при отримані відповіді від сервера виконує вказану функцію в якій передається об'єкт response як параметр.

response містить наступні властивості і методи:
clone() - повертає копію об'єкту.
arrayBuffer() - повертає Promise результатом якого буде масив даних.
blob() - повертає Promise результатом якого будуть двійкові дані.
formData() - повертані Promise результатом якого будуть дані FormData.
json() - повертає Promise результатом якого будуть дані формату JSON.
text() - повертає Promise результатом якого будуть текстові дані.
type - рядок з назвою типу даних.
url - URL адреса відповіді від сервера.

status - код статусу відповіді.
statusText - назва статусу відповіді.
ok - логічне значення,чи завантажено без помилок тобто статус від 200 до 299.
bodyUsed - логічне значення, чи завантажено body.

headers - об'єкт для роботи з заголовком. Містить наступні методи:
- append() - додає значення у заголовок.
- delete() - видаляє значення з заголовка.
- get() - повертає вказане значення заголовка.
- set() - задає значення.
- getAll() - повертає усі заголовки

fetch() не підтримує синхроного запиту, а лише асинхроний.
Тому якщо Вам необхідний синхроний запит - використовуйте XMLHttpRequest

https://uk.javascript.info/xmlhttprequest.

Async/await

Існує спеціальний синтаксис для більш зручної роботи з промісами, який називається “async/await”. Його напрочуд легко зрозуміти та використовувати.

Асинхронні функції

Почнемо з ключового слова async. Його можна розмістити перед функцією, наприклад:

```
async function f() {return 1;}
```

Слово async перед функцією означає одну просту річ: функція завжди повертає проміс. Інші значення автоматично загортаються в успішно виконаний проміс. Наприклад, ця функція повертає успішно виконаний проміс з результатом 1; протестуймо:

```
async function f() {return 1;}f().then(alert); // 1
```

...Ми могли б явно повернути проміс, результат буде таким самим:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

Отже, async гарантує, що функція повертає проміс і обгортає в нього не-проміси. Досить просто, правда? Але це ще не все. Є ще одне ключове слово, await, яке працює лише всередині async-функцій, і воно досить круте.

Await

Синтаксис:

```
// працює лише всередині async-функцій

let value = await promise;
```

Ключове слово await змушує JavaScript чекати, поки проміс не виконається, та повертає його результат.

Обробка помилок

Якщо проміс виконується нормально, то await promise повертає результат. Але у випадку завершення з помилкою він генерує помилку, ніби в цьому рядку був оператор throw.

Цей код:

```
async function f() {
  await Promise.reject(new Error("Упс!"));
}
```

...робить те ж саме, що й цей:

```
async function f() {
  throw new Error("Упс!");
}
```

У реальних ситуаціях може пройти деякий час, перш ніж проміс завершиться з помилкою. У цьому випадку буде затримка, перш ніж await видасть помилку.

Ключове слово async перед функцією має два ефекти:

Змушує її завжди повертати проміс.
Дозволяє використовувати в ній await.
Ключове слово await перед промісом змушує JavaScript чекати, поки цей проміс не виконається, а потім:
Якщо це помилка, генерується виняток — так само, ніби throw error було викликано саме в цьому місці.
В іншому випадку він повертає результат.
Разом вони забезпечують чудову структуру для написання асинхронного коду, який легко і читати, і писати.

За допомогою async/await нам рідко потрібно писати promise.then/catch, але ми все одно не повинні забувати, що вони засновані на промісах, тому що іноді (наприклад, на верхньому рівні вкладеності) нам доводиться використовувати ці методи. Також Promise.all стає в нагоді, коли ми чекаємо на виконання багатьох завдань одночасно.

Конструкція try...catch дозволяє обробляти помилки, що виникають протягом роботи скрипту. Це, в прямому сенсі, дозволяє “спробувати” виконати код та “перехопити” помилки, що можуть виникнути.

Синтаксис:

```
try {
  // виконання коду
} catch (err) {
  // якщо трапилась помилка,
  // передати виконання в цей блок
} finally {
  // завжди виконається після try/catch
}
```

Також ми можемо пропустити секцію catch чи finally, тому скорочені конструкції try...catch та try...finally теж валідні.

Об’єкт помилки має наступні властивості:

message – розбірливе повідомлення про помилку.
name – рядок з іменем помилки (назва конструктора помилки).
stack (нестандартна, але широко підтримувана) – стек викликів на момент створення помилки.
Ми можемо пропустити отримання об’єкту помилки, якщо використати catch { замість catch (err) {.

Також ми можемо генерувати власні помилки за допомогою оператору throw. Технічно, будь-що можна передати аргументом в throw, але, зазвичай, використовується об’єкт, успадкований від вбудованого класу Error. Детальніше про розширення помилок в наступному розділі.

Повторне викидання – важливий шаблон в роботі з помилками: переважно блок catch знає як обробляти помилки певного типу, тому він повинен знову викидати невідомі типи помилок.

Навіть, якщо ми не використовуємо try...catch, більшість середовищ дозволяють встановити “глобальний” обробник помилок. В браузерах це window.onerror.

Додаткові матеріали

- https://www.youtube.com/playlist?list=PLzdnOPI1iJNfMRZm5DDxco3UdsFegvuB7
- https://www.youtube.com/watch?v=AEaKrq3SpW8&list=PL8dPuuaLjXtNlUrzyH5r6jN9ulIgZBpdo
- https://uk.wikipedia.org/wiki/CRUD
- https://softpro.ua/nalashtuvannja-notifikaci-pri-crud-operacijah_24870640147863978884
- https://www.sitepoint.com/rest-api/
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- https://uk.javascript.info/websocket
- https://uk.javascript.info/fetch-api
- https://uk.javascript.info/xmlhttprequest
- https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- https://uk.javascript.info/try-catch