### ЛЕКЦІЯ 6. Архітектура RESTful. Принципи REST. Побудова RESTful API. HTTP методи та коди стану

Архітектура REST (Representational State Transfer) стала основним підходом до проєктування API в сучасних веб-додатках. REST забезпечує простий та масштабований спосіб обміну даними між клієнтами та серверами через HTTP, використовуючи різні методи та формати даних, такі як JSON або XML. У цій лекції ми розглянемо принципи REST, структуру RESTful API, методи HTTP, коди стану та надамо приклади їх використання.

**1. Що таке REST?**

REST (Representational State Transfer) — це стиль архітектури для побудови мережевих додатків, який запропонував Рой Філдінг у 2000 році. REST орієнтований на використання стандартних методів HTTP для взаємодії між клієнтами та серверами.

Ключові принципи REST:

* **Клієнт-серверна архітектура**: Клієнт і сервер є незалежними компонентами. Клієнт посилає запити, а сервер відповідає на них.
* **Статевість (statelessness)**: Кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для обробки, тому сервер не зберігає стан між запитами.
* **Кешування**: Сервер може вказати, чи можна кешувати відповідь. Це дозволяє знизити навантаження на сервер і прискорити взаємодію.
* **Єдині інтерфейси (Uniform Interface)**: Взаємодія між компонентами виконується за допомогою єдиних, стандартизованих механізмів, зокрема через ресурси, URL-адреси та HTTP-методи.
* **Шарова система**: REST може використовувати шарову архітектуру, де клієнти взаємодіють з сервером через посередницькі вузли, такі як проксі чи шлюзи.
* **Ресурси**: У REST кожен ресурс ідентифікується унікальним URL. Наприклад, запис у базі даних користувачів може бути доступним за адресою /users/1.

**2. Принципи REST**

**2.1 Ресурси**

У контексті REST ресурсом може бути будь-яка сутність, така як користувач, продукт, замовлення тощо. Кожен ресурс має унікальну адресу (URL), через яку до нього можна звернутися. Для ідентифікації ресурсу використовуються стандартні механізми адресації, наприклад:

bash

GET /users/1 - отримати інформацію про користувача з ID 1

**2.2 Репрезентації**

Ресурс може мати кілька форматів репрезентацій, наприклад, у форматі JSON або XML. Клієнт може вибирати формат за допомогою заголовка Accept у HTTP-запиті.

http

GET /users/1 HTTP/1.1

Host: example.com

Accept: application/json

У цьому прикладі клієнт запитує репрезентацію ресурсу у форматі JSON.

**2.3 Маніпуляція ресурсами через представлення**

Зміни у ресурсах здійснюються через HTTP-методи (GET, POST, PUT, DELETE). Кожен метод відповідає певній операції:

* **GET**: Отримання інформації про ресурс.
* **POST**: Створення нового ресурсу.
* **PUT**: Оновлення ресурсу.
* **DELETE**: Видалення ресурсу.

Наприклад, щоб створити нового користувача, використовується метод POST:

http

POST /users HTTP/1.1

Host: example.com

Content-Type: application/json

{

 "name": "John Doe",

 "email": "john@example.com"

}

**2.4 Stateless (Безстанова взаємодія)**

У REST кожен запит повинен містити всю інформацію для його обробки. Це означає, що сервер не зберігає стан між запитами, і кожен запит є незалежним. Наприклад, у запиті на авторизацію потрібно передавати токен автентифікації в кожному запиті:

http

GET /profile HTTP/1.1

Host: example.com

Authorization: Bearer <токен>

**3. Побудова RESTful API**

**3.1 Архітектура RESTful API**

RESTful API — це програмний інтерфейс, який побудований на основі принципів REST. Він дозволяє клієнтам виконувати операції CRUD (Create, Read, Update, Delete) через стандартні HTTP-запити.

Основні елементи RESTful API:

* **База URL**: API має бути доступне через унікальну базову URL-адресу, наприклад, https://api.example.com.
* **Ресурси**: Кожен тип ресурсу доступний через певну кінцеву точку. Наприклад:
	+ GET /users — отримати список користувачів.
	+ POST /users — створити нового користувача.
	+ GET /users/{id} — отримати інформацію про конкретного користувача.
	+ PUT /users/{id} — оновити інформацію користувача.
	+ DELETE /users/{id} — видалити користувача.

**3.2 Приклад побудови RESTful API на Flask (Python)**

python

from flask import Flask, jsonify, request

app = Flask(\_\_name\_\_)

users = [

 {"id": 1, "name": "John Doe", "email": "john@example.com"},

 {"id": 2, "name": "Jane Smith", "email": "jane@example.com"}

]

@app.route('/users', methods=['GET'])

def get\_users():

 return jsonify(users)

@app.route('/users/<int:user\_id>', methods=['GET'])

def get\_user(user\_id):

 user = next((user for user in users if user["id"] == user\_id), None)

 if user:

 return jsonify(user)

 else:

 return jsonify({"message": "User not found"}), 404

@app.route('/users', methods=['POST'])

def create\_user():

 new\_user = request.get\_json()

 new\_user["id"] = len(users) + 1

 users.append(new\_user)

 return jsonify(new\_user), 201

if \_\_name\_\_ == '\_\_main\_\_':

 app.run(debug=True)

У цьому прикладі використовуються основні HTTP-методи для взаємодії з ресурсом користувачів (отримання, створення).

**3.3 RESTful практики**

* **Унікальні URL-адреси для кожного ресурсу**: Кожен ресурс повинен бути доступним за унікальною URL-адресою.
* **Використання HTTP-методів**: Використовуйте відповідні методи для різних операцій (GET, POST, PUT, DELETE).
* **Статус-коди**: Використовуйте коректні коди стану HTTP для відповіді сервера (200 OK, 404 Not Found, 201 Created тощо).

**4. HTTP методи та їх використання**

**4.1 GET**

Метод GET використовується для отримання даних з сервера. Він не має побічних ефектів і є ідемпотентним, тобто виконання запиту кілька разів не змінить стан сервера.

Приклад запиту:

http

GET /users/1 HTTP/1.1

Host: example.com

**4.2 POST**

Метод POST використовується для створення нового ресурсу на сервері. Він не є ідемпотентним: повторний запит може створити новий ресурс.

Приклад запиту:

http

POST /users HTTP/1.1

Host: example.com

Content-Type: application/json

{

 "name": "John Doe",

 "email": "john@example.com"

}

**4.3 PUT**

Метод PUT використовується для оновлення існуючого ресурсу. Він ідемпотентний: повторне виконання запиту матиме той самий результат.

Приклад:

http

PUT /users/1 HTTP/1.1

Host: example.com

Content-Type: application/json

{

 "name": "John Doe",

 "email": "john.doe@example.com"

}

**4.4 DELETE**

Метод DELETE використовується для видалення ресурсу на сервері. Він також є ідемпотентним.

Приклад:

http

DELETE /users/1 HTTP/1.1

Host: example.com

**5. HTTP коди стану**

**5.1 200 OK**

Відповідь із кодом 200 свідчить про успішне виконання запиту. Наприклад, при успішному виконанні запиту GET сервер поверне відповідь з кодом 200.

**5.2 201 Created**

Цей код свідчить про успішне створення нового ресурсу. Використовується разом із методом POST.

**5.3 400 Bad Request**

Сервер повертає код 400, якщо запит клієнта містить помилки.

**5.4 401 Unauthorized**

Код 401 вказує на те, що клієнт не автентифікований або передав некоректні дані для автентифікації.

**5.5 404 Not Found**

Цей код означає, що ресурс, запитаний клієнтом, не знайдено на сервері.

**5.6 500 Internal Server Error**

Код 500 вказує на помилку на стороні сервера.

**6. Приклади**

**6.1 Створення ресурсу за допомогою POST**

http

POST /users HTTP/1.1

Host: example.com

Content-Type: application/json

{

 "name": "Alice",

 "email": "alice@example.com"

}

У відповідь сервер повертає код 201 Created:

http

Копіювати код

HTTP/1.1 201 Created

Location: /users/3

**6.2 Оновлення ресурсу за допомогою PUT**

http

PUT /users/1 HTTP/1.1

Host: example.com

Content-Type: application/json

{

 "name": "John Updated",

 "email": "john.updated@example.com"

}

Отже, архітектура RESTful є основою сучасної клієнт-серверної взаємодії, забезпечуючи простий та ефективний спосіб доступу до ресурсів за допомогою стандартних HTTP-методів. Принципи REST дозволяють створювати масштабовані та підтримувані API, що робить їх незамінними для веб-розробки.

Для повного опанування та ефективного використання архітектури RESTful необхідно володіти кількома додатковими знаннями та навичками, які розширюють розуміння її роботи та можливості застосування в реальних проектах. Наприклад:

**1. Глибше розуміння HTTP і його протоколів**

RESTful API використовує HTTP як основу для взаємодії між клієнтами та серверами. Тому важливо розуміти структуру HTTP-запитів та відповідей, типи заголовків, що використовуються для передачі додаткової інформації, а також такі протоколи, як:

* **HTTP/1.1 і HTTP/2**: різниця між версіями, як нові функції (наприклад, мультиплексування) можуть покращити продуктивність API.
* **HTTPS**: важливість захищених з’єднань для забезпечення конфіденційності та безпеки даних.

**2. Автентифікація та авторизація**

У сучасних API необхідно мати механізми для автентифікації та авторизації користувачів. Основні методи включають:

* **OAuth 2.0**: протокол авторизації, який використовується для надання стороннім додаткам обмеженого доступу до ресурсів.
* **JWT (JSON Web Tokens)**: метод передачі інформації між сторонами у вигляді JSON-токенів для автентифікації користувачів.
* **Basic Auth**: простий спосіб автентифікації через заголовки HTTP, хоча він рідко використовується для сучасних API через низький рівень безпеки.

**3. Робота з форматами даних**

RESTful API зазвичай використовує формат JSON для обміну даними, проте важливо знати й інші формати:

* **JSON**: найпоширеніший формат для передачі даних у RESTful API, який легко читається та використовується мовами програмування.
* **XML**: менш поширений у сучасних API, але все ще використовується в певних випадках, наприклад у старіших системах або коли потрібно підтримувати специфічні стандарти.
* **YAML**: часто використовується для конфігурації API, наприклад, у документах OpenAPI (Swagger).

**4. Документування API**

Забезпечення належної документації для вашого RESTful API є надзвичайно важливим для спрощення його використання іншими розробниками. Використання інструментів для автоматизації документування, таких як:

* **Swagger/OpenAPI**: стандарт для документування RESTful API, який дозволяє створювати інтерактивну документацію, тестувати запити та генерувати коди для клієнтів.

**5. Інструменти для тестування API**

Тестування RESTful API є важливим етапом розробки. Існує безліч інструментів, які дозволяють тестувати запити до API, аналізувати відповіді та виявляти помилки. Серед них:

* **Postman**: потужний інструмент для створення та виконання HTTP-запитів, тестування та моніторингу API.
* **cURL**: командний інструмент для виконання HTTP-запитів, корисний для тестування API без використання графічного інтерфейсу.
* **Insomnia**: легкий інструмент для роботи з REST API, що пропонує інтуїтивно зрозумілий інтерфейс для взаємодії з API.

**6. Управління помилками та коди стану**

Для того, щоб RESTful API було зрозумілим і легким у використанні, важливо забезпечити коректне управління помилками. Розуміння правильних кодів стану HTTP дозволяє краще описати стан відповіді та надавати більш детальні повідомлення про помилки:

* **4xx** коди (наприклад, 400 Bad Request, 401 Unauthorized, 404 Not Found) вказують на помилки з боку клієнта.
* **5xx** коди (наприклад, 500 Internal Server Error, 503 Service Unavailable) повідомляють про проблеми на стороні сервера.
* Окрім кодів стану, важливо надавати змістовні повідомлення про помилки в тілі відповіді, що допомагають користувачам розуміти, як виправити запит.

**7. Патерни проєктування API**

Розуміння патернів, які застосовуються при проєктуванні RESTful API, допоможе уникнути помилок і зробити API більш зручним для використання. Серед них:

* **Versioning (версіювання)**: різні підходи до підтримки різних версій API, такі як використання заголовків або включення версії в URL.
* **Rate limiting (обмеження кількості запитів)**: встановлення обмежень на кількість запитів до API, щоб запобігти перевантаженню сервера.
* **HATEOAS (Hypermedia as the Engine of Application State)**: концепція, яка передбачає, що кожен відповідь API містить інформацію про подальші доступні дії для клієнта (наприклад, посилання на інші ресурси).

**8. Кешування та оптимізація продуктивності**

Ефективне кешування допомагає знизити навантаження на сервер і зменшити затримки для користувачів:

* **HTTP кешування**: використання заголовків Cache-Control, Expires та ETag для контролю кешування відповідей API.
* **API шлюзи**: такі як AWS API Gateway або Kong можуть використовуватися для управління та оптимізації API, включаючи кешування.

**9. Масштабування та безпека**

У великих системах важливо розуміти аспекти масштабування та безпеки:

* **Load balancing**: забезпечення рівномірного розподілу навантаження на сервери.
* **Rate limiting**: запобігання DDoS-атакам і обмеження зловживань.
* **Використання HTTPS**: усі API, особливо ті, що працюють із конфіденційними даними, повинні використовувати шифрування для захисту даних.

**10. Підтримка WebSockets**

Хоча REST підходить для багатьох задач, є сценарії, коли потрібно встановити постійне з'єднання між клієнтом і сервером, наприклад для реального часу (чат, нотифікації). У таких випадках корисно знати, як працюють **WebSockets** і коли їх використовувати замість REST API.

**Висновок**

Опанування архітектури RESTful вимагає розуміння основних принципів її побудови та взаємодії, але для повноцінного використання потрібно також вивчити супутні технології, як-от HTTP-протоколи, безпека даних, кешування та тестування API. Навички в цих галузях дозволять вам ефективно працювати з RESTful API і створювати масштабовані, безпечні та зрозумілі API для сучасних додатків.