DOI: https://doi.org/10.36910/6775-2524-0560-2022-49-01
УДК 004.42+004.45
Колпаков Пилип Сергійович, бакалавр,
https://orcid.org/0000-0001-6375-676X
Марченко Олександр Іванович, к.т.н., доцент,
https://orcid.org/0000-0002-4537-3420
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», м. Київ, Україна

СПОСІБ ОПТИМІЗАЦІЇ ПЕРЕДАЧІ ОДНОТИПНИХ JSON-ДОКУМЕНТІВ КОМП'ЮТЕРНИМИ МЕРЕЖАМИ

Колпаков П.С., Марченко О.І. Спосіб оптимізації передачі однотипних JSON-документів комп'ютерними мережами. У даній статі проаналізована проблема надлишковості та неефективності формату JSON. Розглянуті існуючі способи оптимізації та запропонований новий спосіб, що дозволяє досягти поставлених задач без втрати основних переваг зазначеного формату. Розглянутий спосіб базується на ідеї попереднього аналізу повідомлень в системі з метою їх структуризації та розділення на окремі класи. Результатом виконання дій нового способу є спеціальний бінарний формат документів, оптимізований для конкретної системи але з можливістю передачі будь-яких повідомлень.
Ключові слова: JSON-формат, JSON-документ, REST-архітектура, оптимізація передачі, бінарний формат, стиснення інформації.

Kolpakov P.S., Marchenko O.I. An optimization technique for transferring of identical JSON documents over computer networks. This paper contains the analysis of the problem of redundancy and inefficiency of the JSON format. The existing optimization techniques are considered and the new technique is proposed, which allows to solve the problem without losing the main advantages of this format. The proposed technique is based on the idea of preliminary analysis of messages in a system with the aim of structuring them and dividing them into separate classes. The result of performing the actions of the new technique is a special binary format optimized for the specific system but with a possibility of transmitting messages of any structure.
Keywords: JSON-format, JSON-document, REST-architecture, optimization of transfer, binary format, data compression.

Постановка проблеми

Формат JSON [1] широко використовується для передачі інформації комп'ютерними мережами. Завдяки своїй розповсюдженості він є простим у впровадженні, а завдяки текстовому представленню даних та динамічній типізації він є простим у використанні. Однак, формат JSON має також ряд серйозних недоліків:

1) оскільки JSON-формат є текстовим, а текстове (рядкове) представлення даних не є компактним, то дані у форматі JSON мають значну надлишковість;
2) оскільки JSON-формат є динамічно типізованим, структурна інформація (типи значень, назви полів) має передаватися в кожному повідомленні, що є також надлишковим у більшості випадків;
3) оскільки JSON-формат є текстовим та динамічно типізованим, процес зчитування даних потребує додаткової логіки обробки (додаткових обчислень).

Підсумовуючи, можна стверджувати: формат JSON є значною мірою надлишковим, і деякою мірою — повільним. У випадках передачі значних об'ємів інформації використання цього формату може стати тим самим "пляшковим горлечком", що суттєво обмежуватиме ефективність усієї програмної системи. Тому, розробка нових способів оптимізації передачі даних у форматі JSON є актуальною задачею.

Розглянемо основні випадки використання формату JSON, щоб знайти простір для оптимізації.

Найбільш широко формат JSON використовується у програмних системах, що реалізують REST-архітектуру [2]. JSON-формат, хоча й не є обов'язковою її складовою, але є типовим форматом передачі повідомлень для цієї архітектури. Як правило, розмір повідомлень є малим, і їх структура зводиться до невеликої множини подібних між собою класів. Фактично, повідомлення є однотипними.

Аналогічна ситуація виникає також з деякими документо-орієнтованими базами даних (MongoDB, DocumentDB), що, фактично, використовують JSON-формат для збереження та доступу до інформації. Ці бази даних вже використовують певні способи оптимізації [3], але ці

способи не є вичерпними, і простір для пропозиції нових рішень з метою збільшення швидкодії або зменшення об'єму стиснених даних все ще залишається.

Як підсумок зазначимо, що у основних випадках використання JSON-формату для передачі або збереження даних повідомлення є однотипними і, відповідно, подальший аналіз будемо проводити з урахуванням цього факту.

Термінологія

Формат JSON-— широковживаний текстовий формат даних, описаний у стандарті RFC8259 [1].

JSON-документ — дані у форматі JSON, представлені у текстовому вигляді об'єктом JSON (*object*, згідно стандарту).

JSON-повідомлення — те ж саме, що і JSON-документ, але з прив'язкою до процесу передачі даних комп'ютерною мережею.

Структурна інформація JSON-повідомлення — відомості щодо конкретного JSON-повідомлення, що вичерпно описують його структуру: імена полів, порядок вкладеності об'єктів/масивів, типи значень.

Атомарне значення — у контексті JSON-повідомлень — значення числового, рядкового типу або пусте значення (null).

Неатомарне значення — у контексті JSON-повідомлень — об'єкт або масив.

Аналіз існуючих рішень

Розглянемо рішення зазначеної проблеми серед існуючих способів. Порівняємо їх за такими критеріями: простота та швидкість кодування та декодування повідомлень, надлишковість повідомлень, простота впровадження оптимізації/технології, безпека даних, збереження основних переваг JSON-формату, наявність додаткових особливостей, переваг, обмежень.

Були проаналізовані наступні рішення:

1) використання бінарних аналогів формату JSON [3, 4, 5, 6].
2) використання алгоритмів архівації для стиснення повідомлень [8, 9].
3) використання статично типізованих аналогів JSON [10, 11].

У якості підсумку зазначимо наступні моменти.

1. Використання бінарних аналогів JSON суттєво пришвидшує обробку даних [7], але зменшення розміру повідомлення не є значним [7], оскільки ці формати не позбавляють від необхідності зберігати структурну інформацію в кожному повідомленні.

2. Використання алгоритмів архівації задля зменшення об'єму окремих повідомлень не є ефективним. Найчастіше JSON-формат використовується для передачі саме невеликих повідомлень, які по окремості мають невисокий рівень надлишковості.

3. Використання статично типізованих аналогів JSON – gRPC [10] та Apache Thrift [11] – дозволяє значно зменшити розмір повідомлень та прискорити їх обробку. Для збереження даних в них використовується бінарний формат, що не вимагає передачі структурної інформації. Але ці технології позбавлені основних переваг формату JSON — динамічної типізації та простоти впровадження, оскільки програміст вимушений витрачати багато часу на повний вичерпний опис усіх структур даних.

Виконаний аналіз показав, що жоден з проаналізованих способів передачі повідомлень або оптимізації передачі повідомлень не дозволяє вирішити поставлену задачу ефективно. Жоден зі способів не дозволяє достатньою мірою зменшити розмір повідомлення при щонайменше збереженні прийнятної швидкості обробки, або, навіть, прискоренні обробки.

На основі виконаного аналізу, розглянувши переваги та недоліки існуючих способів, можна зробити висновок, що з метою зменшення обсягу даних, що передаються, доцільним є використання бінарного формату даних, який не буде містити структурну інформацію (структура об'єктів, імена полів). Але при цьому, задля збереження основних переваг типової REST-архітектури, варто уникнути явного опису структур даних для кожного повідомлення.

Тому є актуальною розробка нового способу передачі великих обсягів інформації, який, з однієї сторони, вирішував би задачу оптимізації передачі JSON-документів за обсягом даних (в тому числі для прискорення процесу їх обміну), а, з іншої сторони, не вимагав би явного опису програмістом структур даних повідомлень та зберігав би прийнятну швидкодію передачі документів. При розробці нового способу треба орієнтуватись на такий цільовий випадок, як передача великої кількості окремих однотипних повідомлень.

Спосіб оптимізації передачі однотипних JSON-документів

Будемо розглядати поставлену задачу при умові, що програмна система, для якої буде розроблюватися спосіб оптимізації передачі повідомлень, вже існує, і, відповідно, ми вже маємо зразки типових JSON-повідомлень, які використовуються в цій системі.

Потрібно зауважити, що якщо при передачі окремого JSON-повідомлення до даних не додається структурна інформація, то вона має зберігатися або передаватися окремо. Частина програмної системи, що відповідає за кодування та декодування повідомлень, може мати перелік усіх можливих класів в собі з самого початку (як у випадку прототипів protobuf в технології gRPC), або отримувати його на початку кожного сеансу окремим повідомленням. Реалізація як першого, так і другого варіанту передачі структурної інформації не має бути складною. Головне питання полягає в тому, як отримати структурну інформацію для усіх можливих повідомлень у системі.

Оскільки, як було зазначено вище, варіант ручного опису програмістом усіх структур даних вимагає великих витрат часу, доцільним виглядає розробка способу, в якому збір структурної інформації буде виконуватися автоматично. Очевидно, що такий аналіз повідомлень повинен виконуватися попередньо. Маючи деяку множину зразків повідомлень, можна виділити з них необхідну структурну інформацію окремим алгоритмом та зберегти її. Використовуючи цю збережену структурну інформацію, можна лише посилатись на неї, а не передавати її в повному обсязі для кожного окремого повідомлення, і, таким чином, зменшити надлишковість інформації в повідомленнях.

Автори пропонують спосіб оптимізації передачі однотипних JSON-документів комп'ютерними мережами, який складається з двох етапів: етапу попереднього аналізу та етапу кодування/декодування даних.

Розглянемо спочатку **загальну ідею** цього нового способу оптимізації, що полягає у компактному кодуванні JSON-повідомлень.

Маючи попередньо зібрану структурну інформацію для конкретних повідомлень, потрібно знайти спосіб відрізнити одні повідомлення від інших, розділити повідомлення на окремі класи. Оскільки формат JSON має динамічну типізацію, прив'язуватися до конкретного переліку полів конкретного типу виглядає недоцільним. В окремих випадках деякі поля можуть бути відсутні або мати інший тип значення (наприклад, масив замість об'єкту), і, в результаті, перелік усіх можливих класів повідомлень буде занадто великий. Тому було вирішено зберігати структурну інформацію не на рівні об'єктів, а на рівні полів. Розглянемо цю загальну ідею на прикладі такого JSON-документа:

```
{
"foo": 1234,
"bar": {
      "a":  5678,
      "b": "abc"
}
}
```

На першому етапі способу (етапі попереднього аналізу) алгоритм має зібрати структурну інформацію з повідомлення. Для цього прикладу вона може виглядати наступним чином:

Запис_1: інформація про поле на рівні кореневого об'єкту: назва — *foo,* тип значення — *число.*

Запис_2: інформація про поле на рівні кореневого об'єкту: назва — *bar,* тип значення — *об'єкт.*

Запис_2.1: інформація про поле на рівні об'єкту bar: назва — *a*, тип значення — *число.*

Запис_2.2: інформація про поле на рівні об'єкту bar: назва — *b*, тип значення — *рядок.*

Тут «Запис_1», «Запис_2» і т.д. позначають записи з таблиці структурної інформації. Ця таблиця має бути остаточно сформована на етапі попереднього аналізу. Використовуючи посилання на записи у цій таблиці, на етапі кодування вищенаведений JSON-документ може бути записаний наступним чином:

```
Запис_1: 1234
Запис_2:
Запис_2.1: 5678
Запис_2.2: abc
Кінець файлу
```

Оскільки при такому кодуванні присутнє посилання на запис об'єкту з поля *bar* («Запис_2»), ми маємо можливість додатково спростити закодований документ, ввівши контекстно-залежні посилання та термінальний символ для об'єкту. Наприклад:

```
Запис_1: 1234
Запис_2:
      Запис_1: 5678
      Запис_2: abc
Кінець об'єкту
Кінець файлу
```

Програма, що зчитує повідомлення, зустрівши посилання на запис для об'єкту, переходить в контекст записів цього конкретного об'єкту, а після обробки термінального символу контекст замінюється на попередній. Тим самим ми досягаємо значного зменшення кількості можливих варіантів для посилань на записи й, відповідно, скорочення самих посилань.

Розглянемо тепер детально алгоритм збору необхідної структурної інформації, тобто **алгоритм попереднього аналізу** запропонованого способу.

Оскільки структурна інформація має бути розподілена по окремих контекстах, кожен з яких відповідає за конкретний об'єкт, доцільним буде представлення цієї інформації у вигляді дерева. Корінь цього дерева відповідає за кореневий об'єкт повідомлення та зберігає інформацію щодо усіх можливих полів та їх типів. Якщо значення поля є атомарним, то зберігається лише список можливих атомарних типів для значень цього поля. Якщо значення є неатомарним, то додатково зберігається посилання на наступний вузол дерева, що відноситься до відповідного об'єкта або масива. Таке дерево показане на рисунку 1.

Рис. 1. Алгоритм попереднього аналізу: загальна частина

Це дерево ініціалізується один раз для усіх повідомлень, що оброблюються, а обробка кожного окремого JSON-документа починається з кореня (кореневого контексту). При обробці конкретного об'єкта або масива відбувається перехід у контекст відповідного поля й рекурсивно викликається відповідна процедура (рис. 2 та рис. 3).

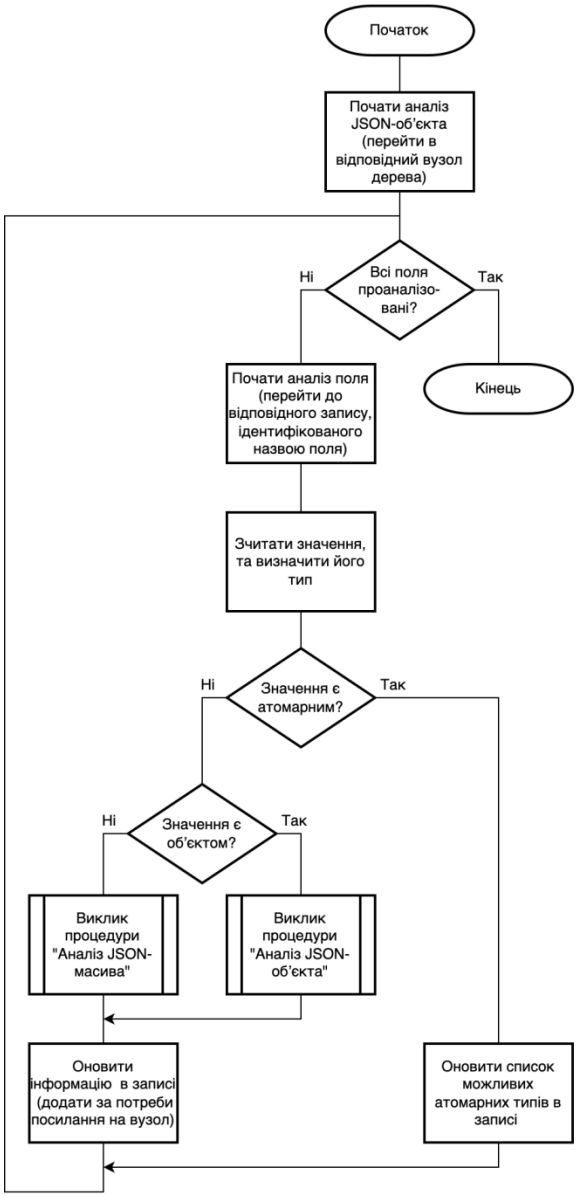

Рис. 2. Алгоритм попереднього аналізу: процедура «Аналіз JSON-об'єкта»

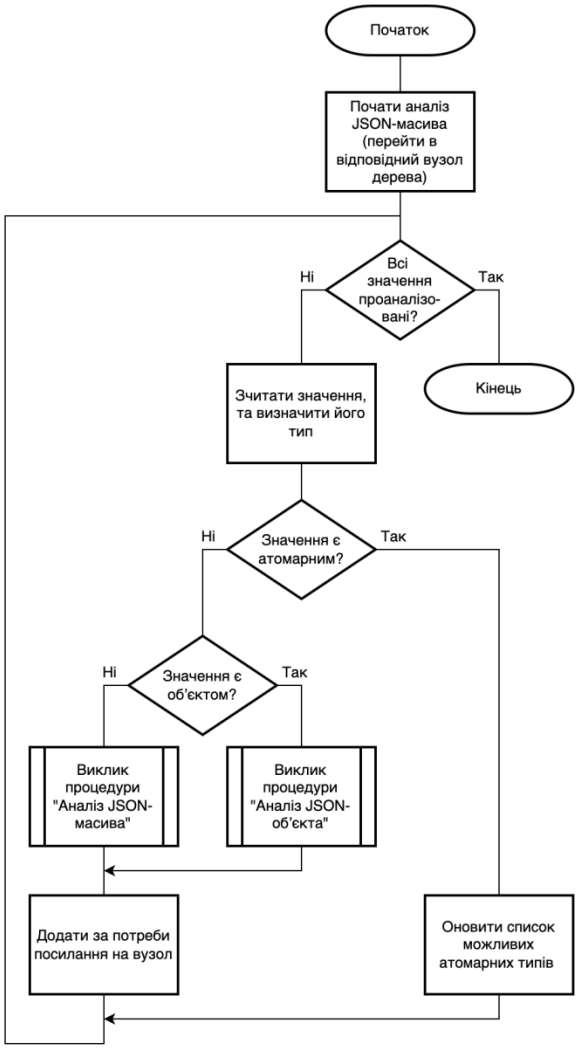

Рис. 3. Алгоритм попереднього аналізу: процедура «Аналіз JSON-масива»

Таким чином, маємо всю необхідну структурну інформацію. Після цього залишається зберегти її у зручному вигляді та згенерувати короткі унікальні ідентифікатори кожного поля для наступного етапу. Збережену структурну інформацію з проставленими ідентифікаторами надалі будемо називати **схемою повідомлень.**

Розглянемо тепер алгоритми кодування та декодування повідомлень.

Алгоритм кодування повідомлень є занадто складним для графічного представлення або детального опису в рамках цієї статті. Тому, опишемо його текстом в узагальненому вигляді.

1. Зчитування JSON-файлу виконується послідовно, а схема повідомлень зберігається в пам'яті.

2. Кореневий об'єкт ніяким чином не ідентифікується.

3. Послідовно зчитуються поля та елементи масиву.

4. Якщо значення поля не є атомарним, то контекст схеми перемикається на відповідний вузол.

5. Якщо назва поля та тип значення, що зчитуються, співпадають з даними у схемі, то записується байтовий ідентифікатор поля зі схеми, а після нього — *компактне детерміноване представлення відповідного значення*.

6. Якщо назва поля співпадає зі схемою, але тип значення інший, то записується байтовий ідентифікатор поля зі схеми, використовується *префікс зміни типу*, вказується конкретний тип значення й записується саме значення.

7. Якщо зчитане поле в схемі відсутнє, то записується *спеціальне зарезервоване значення ідентифікатора, що вказує на невідомий запис*. Після цього записується повна назва поля в рядковому вигляді, потім записується тип значення і тільки після цього — саме значення. Будемо називати такий запис поля *динамічним* (від "динамічна типізація").

8. Якщо об'єкт записується *динамічним* способом, то усі його поля записуються тим же самим способом.

9. При записі елементів масиву записується тип значення елемента, а після цього саме значення. Якщо значення є неатомарним — відповідно, перемикаємо контекст схеми на відповідний вузол.

Алгоритм декодування повідомлень є алгоритмом, зворотнім до попереднього. Опишемо його також узагальнено.

1. Документ зчитується послідовно, схема повідомлень зберігається в пам'яті.

2. Якщо зустрічається ідентифікатор поля, то зчитується інформація зі схеми та декодується значення *відповідним детермінованим способом в залежності від типу*.

3. Якщо значення не є атомарним, то переходимо до відповідного вузла дерева схеми.

4. Враховується *префікс заміни типу* у разі його наявності.

5. Якщо зустрічається *зарезервоване значення ідентифікатора*, то зчитується назва поля, тип та значення без вказування схеми. Зчитується *динамічний* запис.

6. Масив зчитується аналогічно до об'єкта.

В описах обох алгоритмів було зазначено, що атомарні значення записуються так званим "спеціальним детермінованим способом". Згідно цього способу рядкове значення записується як послідовність байтів (кодів символів у використовуваній кодовій таблиці), числове значення — як послідовність байтів самого числа, а null-значення записується послідовністю байтів нульової довжини, тобто взагалі не записується, оскільки інформації про тип вже достатньо для його ідентифікації.

Як зазначено в пункті 7 першого алгоритму та пункті 5 другого, **алгоритми кодування та декодування здатні коректно оброблювати дані з невідомою структурою**. Це і є основною відмінністю запропонованого способу від статично типізованих аналогів формату JSON. Звісно, що у разі даних з невідомою структурою використовується динамічне представлення, що є менш компактним (бо включає в себе структурну інформацію), але важливо, що таке представлення не містить додаткової інформації порівняно зі звичайним JSON-форматом. У загальному випадку застосування запропонованого способу завжди зменшує розмір повідомлення, а ступінь цього зменшення визначається тим, наскільки повними були дані на етапі попереднього аналізу. Якщо на цьому етапі програма не отримала деякі зразки документів, то алгоритм кодування буде менш ефективним саме для цих документів.

Висновки. В даній статті проаналізовані основні властивості існуючих способів передачі однотипних JSON-повідомлень та був запропонований новий спосіб, який відрізняється від існуючих враховуванням структури документів, що передаються, і використанням формату даних, який не містить структурну інформацію (структуру об'єктів, імена полів), та, в результаті, дозволяє оптимізувати передачу JSON-документів за обсягом, зменшуючи мережевий трафік та не вимагаючи явного опису програмістом структур даних повідомлень.

Запропонований спосіб поєднує у собі одночасно і статичну, і динамічну типізацію даних, що передаються. Якщо повідомлення або його частина мають одну з вже відомих структур, то використовується компактний бінарний формат без структурної інформації, аналогічний до розглянутих статично типізованих форматів, наприклад, protobuf [10]. Якщо повідомлення або його частина мають невідому структуру, то використовується менш компактний бінарний формат зі структурною інформацією, але який не поступається компактністю відомим бінарним аналогам формату JSON (BSON [3], Binary Ion [4] та інші).

Основна новизна запропонованого способу полягає в наявності етапу попереднього аналізу, оскільки саме на цьому етапі відбувається враховування структури документів, що передаються. Аналізуючи зразки повідомлень, формується схема повідомлень, яка дозволяє більшою чи меншою мірою використовувати саме перший спосіб кодування даних, який є більш компактним. Чим більше різних зразків повідомлень надано на етапі попереднього аналізу, тим більш компактним буде результуюче повідомлення, і, відповідно, тим краща буде загальна ефективність передачі однотипних повідомлень в цілому.

Описаний спосіб має гарні перспективи подальшого розвитку, оскільки в результуючому бінарному форматі може бути використаний будь-який підходящий спосіб представлення ідентифікаторів та будь-який підходящий спосіб представлення атомарних значень, а також не має обмежень на додавання нових типів даних (що може бути актуальним для, наприклад, представлення дат чи специфічних структур типу ObjectId [3]). Крім того, запропонований спосіб може використовуватись у поєднанні з потоковою архівацією, що може бути ефективним у разі повідомлень великого розміру.

Список бібліографічного опису

1. The JavaScript Object Notation (JSON) Data Interchange Format [Електроний ресурс] – Режим доступу до ресурсу: https://www.rfc-editor.org/rfc/rfc8259.
2. *Erik Wilde, Cesare Pautasso.* REST: From Research to Practice. — Springer Science & Business Media, 2011. — 528 p. — ISBN 978-1-4419-8303-9.
3. BSON (binary JSON): Specification [Електроний ресурс] – Режим доступу до ресурсу: https://bsonspec.org/spec.html.
4. Ion Binary Encoding [Електроний ресурс] – Режим доступу до ресурсу: https://amzn.github.io/ion-docs/docs/binary.html.
5. Concise Binary Object Representation (CBOR) [Електроний ресурс] – Режим доступу до ресурсу: https://www.rfc-editor.org/rfc/rfc8949.html.
6. MessagePack specification [Електроний ресурс] – Режим доступу до ресурсу: https://github.com/msgpack/msgpack/blob/master/spec.md..
7. Comparison of JSON Like Serializations – JSON vs UBJSON vs MessagePack vs CBOR [Електроний ресурс] – Режим доступу до ресурсу: http://zderadicka.eu/comparison-of-json-like-serializations-json-vs-ubjson-vs-messagepack-vs-cbor/.
8. GZIP file format specification version 4.3 [Електроний ресурс] – Режим доступу до ресурсу: https://www.rfc-editor.org/rfc/rfc1952
9. RFC 7932: Brotli Compressed Data Format [Електроний ресурс] – Режим доступу до ресурсу: https://www.rfc-editor.org/rfc/rfc7932.
10. Protocol Buffers Version 3 Language Specification | Google Developers [Електроний ресурс] – Режим доступу до ресурсу: https://developers.google.com/protocol-buffers/docs/reference/proto3-spec.
11. Apache Thrift - Home [Електроний ресурс] – Режим доступу до ресурсу: https://thrift.apache.org/.

References

1. "The JavaScript Object Notation (JSON) Data Interchange Format" *Internet Engineering Task Force (IETF),* https://www.rfc-editor.org/rfc/rfc8259. Accessed 30 Nov. 2022
2. *Erik Wilde, Cesare Pautasso.* REST: From Research to Practice. — Springer Science & Business Media, 2011. — 528 p. — ISBN 978-1-4419-8303-9.
3. "BSON (binary JSON): Specification" *bsonspec.org,* https://bsonspec.org/spec.html. Accessed 30 Nov. 2022
4. "Ion Binary Encoding" *Amazon, Amazon.com Inc.*, https://amzn.github.io/ion-docs/docs/binary.html. Accessed 30 Nov. 2022
5. "Concise Binary Object Representation (CBOR)" *Internet Engineering Task Force (IETF)*, https://www.rfc-editor.org/rfc/rfc8949.html. Accessed 30 Nov. 2022
6. "MessagePack specification" *Sadayuki Furuhashi,* https://github.com/msgpack/msgpack/blob/master/spec.md. Accessed 30 Nov. 2022
7. "Comparison of JSON Like Serializations – JSON vs UBJSON vs MessagePack vs CBOR" *Ivan Zderadicka*, http://zderadicka.eu/comparison-of-json-like-serializations-json-vs-ubjson-vs-messagepack-vs-cbor/. Accessed 30 Nov. 2022
8. "GZIP file format specification version 4.3" *Internet Engineering Task Force (IETF),* https://www.rfc-editor.org/rfc/rfc1952. Accessed 30 Nov. 2022
9. "RFC 7932: Brotli Compressed Data Format" *Internet Engineering Task Force (IETF),* https://www.rfc-editor.org/rfc/rfc7932. Accessed 30 Nov. 2022
10. "Protocol Buffers Version 3 Language Specification" *Google Developers,* https://developers.google.com/protocol-buffers/docs/reference/proto3-spec. Accessed 30 Nov. 2022
11. "Apache Thrift — Home" *Apache Software Foundation*, https://thrift.apache.org/. Accessed 30 Nov. 2022