Web Developers Baseline

Зміст:

1. Вступ . . . 2
2. Вміння налагодити запит від клієнта до серверу . . . 2
    2.1. Поняття, якими потрібно володіти . . . 2
    2.2. Практичні завдання для виконання . . . 3
3. Вміння налаштувати доступ до сервера . . . 9
    3.1. Поняття, якими потрібно володіти . . . 9
    3.2. Практичні завдання для виконання . . . 9
4. Шифрування/кодування/хешування. У чому різниця? . . . 9
    4.1. Поняття, якими потрібно володіти . . . 9
    4.2. Практичні завдання для виконання . . . 9
5. Вміння використовувати Проміси (Promises) . . . 11
    5.1. Поняття, якими потрібно володіти . . . 11
    5.2. Практичні завдання для виконання (на javascript) . . . 11
6. Уміння аналізувати алгоритмічну складність свого коду . . . 11
    6.1. Поняття, якими потрібно володіти . . . 12
    6.2. Практичні завдання для виконання . . . 12
7. Розуміння роботи операційної системи та докера . . . 13
    7.1. Поняття, якими потрібно володіти . . . 13
    7.2. Практичні завдання для виконання . . . 13
8. Робота з базами даних . . . 14
    8.1. Поняття, якими потрібно володіти . . . 14
    8.2. Практичні завдання для виконання . . . 14
9. Знання про основні вразливості та методи захисту . . . 15
    9.1. Поняття, якими потрібно володіти . . . 15
    9.2. Практичні завдання для виконання . . . 15
10. Web vitals (page speed insights) . . . 15
    10.1. Поняття, якими потрібно володіти . . . 16
    10.2. Практичні завдання для виконання . . . 16
11. Уміння користуватися профайлером . . . 16
    11.1. Поняття, якими потрібно володіти . . . 16
    11.2. Практичні завдання для виконання . . . 16
12. Архітектура стандартного веб-застосунку . . . 16
    12.1. Поняття, якими потрібно володіти . . . 16
    12.2. Практичні завдання для виконання . . . 17
13. Кешування . . . 17
    13.1. Поняття, якими потрібно володіти . . . 17

13.2. Практичні завдання для виконання . . . 17
14. Базові поняття для процесу розробки . . . 17
14.1. Поняття, якими потрібно володіти . . . 18
14.2. Практичні завдання для виконання . . . 19

1. Вступ

Цей документ описує мінімальний набір загальних знань для розробника (від рівня junior і вище). Це список того, що дається в університеті й що може стати в пригоді на роботі, допоможе у проєктуванні, допоможе у налагодженні веб-застосунків, допоможе у співбесідах.

Матеріал складається з розділів. Кожен розділ покриває якийсь базовий аспект розробки і складається з двох підрозділів.

- Поняття, якими потрібно володіти
- Практичні завдання для виконання

Щодо кожного поняття важливо наступне:

1. Вміти розповісти що це
2. Вміти розповісти навіщо конкретна сутність існує. Навіщо вона потрібна.

Для роботи необхідно створити git-репозиторій. Для кожного завдання потрібно створити директорію та по виконанню практичного завдання коммітити артефакти (це може бути текстовий документ, код, скріншоти). Головне завдання, щоб можна було перевірити, як усі завдання були виконані. Для випадків, коли потрібно переглянути відео або прочитати статтю, то зробити файлик з резюме контенту на 1-5 речень (ключові ідеї).

Важливо: якщо щось не виходить, сміливо просити по допомогу в студентів, викладачів. Так, завдання потрібно зробити самому, але гуглити, просити по допомогу і робити все, що допоможе краще розібратися в темі за меншу кількість часу. Сміливо показувати проміжні результати викладачу, команді, друзям тощо.

2. Вміння налагодити запит від клієнта до серверу

2.1. Поняття, якими потрібно володіти

1. IP address, netmask
2. Default gateway
3. DNS Server
4. DNS record types (A, AAAA, MX)

5. DHCP
6. TCP vs UDP
7. HTTP використовує TCP або UDP?
8. DNS використовує TCP або UDP?
9. TCP socket
10. NAT
11. ICMP

2.2. Практичні завдання для виконання

1. З консолі переглянути поточні мережеві налаштування
    a. ip-address комп'ютера
    b. DNS сервер, який використовується
    c. Default gateway
2. Подивитися чи доступний сервер
    a. За допомогою утиліти dig подивитися, куди вказує A запис DNS сервера для домену google.com
    b. За допомогою утиліти dig подивитися подивитися, куди вказує AAAA запис DNS сервера для домену google.com
    c. Пропінгувати сервер по ipv4, ipv6
    d. Просканувати відкриті порти на сервері за допомогою утиліти nmap
3. Перевірити доступність веб-сайту
    a. Спробувати запросити сторінку google.com за допомогою curl
    b. За допомогою curl подивитися куди вказує http редирект під час заходу на google.com
    c. Прочитати статтю "Швидкість світла та Веб"

Швидкість світла та Веб

Майже завжди, під час розмови про продуктивність веб-сайту, ми говоримо про час очікування користувачем якоїсь події («First contentful paint» («Largest contentful paint» і тд). Ми часто обговорюємо оптимізації фронтенда і бекенда. І це круто, але залишається корінь усіх бід - швидкість світла. Часто JavaScript розробники випускають цю проблему з уваги.

Припустимо наступне:

1. Неахай бекенд рендерить сторінку (або формує JSON) за 20мс.
2. Не існує ніякого WiFi, провайдерів, маршрутизації і тд. Є просто оптоволоконний кабель, який одним кінцем вставлений у ноутбук юзера, а другим безпосередньо в сервер у Сан-Франциско (SF). Відстань по прямій від Києва (K)) до Сан-Франциско (SF) - 9,848 км (візьмемо 10 тис км. для простоти рахунку).
3. Швидкість світла у вакуумі 300 тис. км/сек, швидкість світла в оптоволокні буде нижчою

- 200 тис. км/сек.

Якщо ми порахуємо час, який проведе наш запит у дорозі, то ми отримаємо: 2 * (10 тис км. / 200 тис. км/сек.) = =0.1сек або 100 мс. Множення на 2 відбувається через те, що ми враховуємо шлях в обидві сторони: від клієнта до сервера і назад від сервера до клієнта.

1. Запит спочатку йде від вашого комп'ютера (клієнта) до сервера — це перша половина шляху.
2. Потім сервер надсилає відповідь назад до клієнта — це друга половина шляху.

Таким чином, загальний час дорівнює часу, який потрібен для проходження запиту туди й назад (round-trip time, RTT). Швидше отримати відповідь не дозволить швидкість світла. Додаємо час опрацювання запиту і ми отримаємо 120 мс - у 6 разів довше, ніж наш запит обробляє наш бекенд.

Запит до бекенда
50ms: Kyiv -------запит----→ SF
20ms: робота бекенда
50ms: Kyiv ←-----відповідь------- SF

Добре, ми вже з'ясували, що ніколи не пограємо в CS:GO з хлопцями з Сан-Франциско з пінгом нижче 100 мс. Давайте далі :)

Перед тим як запросити дані з сервера ми маємо встановити мережеве з'єднання. Протокол HTTP працює поверх TCP, отже нам потрібне TCP-з'єднання із сервером.

Для встановлення TCP з'єднання використовується так зване «потрійне рукостискання» («TCP 3-way handshake») і тепер наш запит має такий вигляд:

TCP з'єднання
50ms: Kyiv -------syn-------→ SF
50ms: Kyiv ←-----syn/ack----- SF
50ms: Kyiv -------ack-------→ SF
Запит до бекенда
Kyiv -------запит----→ SF
20ms: робота бекенда
50ms: Kyiv ←-----відповідь------- SF

Ми не витрачаємо додаткові 50ms після TCP хендшейка, оскільки ми можемо одразу почати надсилати запит після надсилання ack, нам не потрібно чекати на відповідь від сервера. Сервер, як прийме ack, вважатиме з'єднання відкритим і одразу почне обробляти наш запит.

Тобто відповідь користувач отримає через 220ms, в 11 разів довше, ніж відпрацьовував наш бекенд.

Але ми використовуємо HTTPS і нам потрібне SSL/TLS-з’єднання, і воно встановлюється поверх TCP, і в нього є свій механізм рукостискання для обміну ключами шифрування, і це потрібно зробити до того моменту, як ми надішлемо наш запит на сервер.

```
Наша схема перетворюється на:
```

TCP з’єднання
50ms: Kyiv -------syn-------→ SF
50ms: Kyiv ←-----syn/ack----- SF
50ms: Kyiv -------ack-------→ SF
TLS з’єднання
Kyiv ---представлення-→ SF
50ms: Kyiv ←-сертифікати----- SF
50ms: Kyiv ---обмін ключами-→ SF
50ms: Kyiv ←-обмін ключами--- SF
Запит до бекенда
50ms: Kyiv -------запит----→ SF
20ms: робота бекенда
50ms: Kyiv ←-----відповідь------- SF

Тобто в умовах, які не можуть навіть існувати, коли користувач має оптоволоконний кабель завдовжки в 10 тисяч км від свого ноутбука до сервера, він отримає відповідь за 420 мс, що в 21 раз довше, ніж відпрацьовує наш бекенд. Це без урахування того, що нам потрібно ще спочатку збігати до DNS, щоб отримати ip-адресу сервера.

Якщо ми розробляємо веб-застосунки (не важливо фронтенд або бекенд), то зобов’язані розуміти ази роботи вебу.

Ми вже розібралися, що є швидкість світла і вона впливає на затримки під час передачі даних. У нас є затримки на TCP і TLS рукостискання, також є час на шляху запиту і відповіді. Чи можемо ми говорити, що це максимальні затримки, які ми отримуємо?

Насправді все складніше, і навіть за найвищої пропускної спроможності мережі в нас будуть додаткові затримки в передаванні даних.

Є 2 нюанси, які важливі:

1. TCP контролює доставлення пакетів і для того, щоб зрозуміти, що пакети було доставлено, потрібне якесь підтвердження від одержувача. Для цього у відповідь надсилається пакет із прапором «ack» (acknowledge).
    a. Клієнт і сервер від самого початку не знають доступної на цей час пропускної здатності мережі. Вона залежить від можливостей сервера, від можливостей проміжних вузлів, від активності інших вузлів у цій же мережі тощо. Єдиний спосіб

дізнатися - це пробувати передавати дані з різною швидкістю і дивитися, чи доходять вони (чекати підтвердження, що друга сторона отримала їх).

Як це працює?

Коли ми робимо запит до сервера, він спочатку надсилає нам частину даних, потім чекає на підтвердження, потім збільшує обсяг даних, що передаються, вдвічі і знову чекає на відповідь. Якщо все ок, ще раз збільшує і так далі до моменту, поки він не досягне максимального обсягу даних, які готовий приймати клієнт.

Як це все називається?

- Механізм поступового збільшення швидкості передачі даних називається «TCP Slow Start»
- Ліміт відправника на обсяг даних у дорозі називається «Congestion window size» (CWND). Після відправлення цього обсягу даних, відправник повинен чекати підтвердження про те, що дані дійшли. Збільшення цього ліміту і є «TCP Slow Start». ВАЖЛИВО: про цей ліміт знає тільки відправник і він сам для себе його регулює. CWND вимірюється в «сегментах» (сегмент зазвичай не більше 1,46KB). Стартове значення за стандартом - 10 сегментів (14.6KB)
- Також є обмеження одержувача на обсяг даних, який він може прийняти - «Receiver window size» (RWND). Одержувач надсилає відправнику RWND у кожному пакеті з підтвердженням (з прапором ack). Оскільки передача динних відбувається в обидві сторони, то кожна сторона може виступати як одержувачем, так і відправником. Одержувач може передати RWND, що дорівнює нулю, це свідчить про те, що відправник повинен призупинити передачу.

Обидві змінні обмежують кількість даних, яку можна відправити, це завжди мінімум із CWND і RWND.

Тепер давайте намалюємо, що насправді відбувається, коли браузер хоче завантажити наш JavaScript файл на 50KB. Візьмемо ті самі локації - Київ (K) і Сан-Франциско (SF).

TCP з'єднання
50ms: Kyiv -------syn-------→ SF
50ms: Kyiv ←-----syn/ack----- SF
50ms: Kyiv -------ack-------→ SF
TLS з'єднання
Kyiv ---представлення-→ SF
50ms: Kyiv ←-сертифікати----- SF
50ms: Kyiv ---обмін ключами-→ SF
50ms: Kyiv ←-обмін ключами--- SF
HTTP запит до сервера
50ms: Kyiv -------запит----→ SF

20ms: робота бекенда
50ms: Kyiv ←----14.6KB------- SF
50ms: Kyiv -------ack-------→ SF
50ms: Kyiv ←----29.2KB------- SF
50ms: Київ -------ack-------→ SF
50ms: Київ ←----6.2KB-------- SF

Швидкість у 100 Мбіт/с говорить про те, що ми отримаємо 50KB через 4ms, але насправді у нас це займе 620ms. Найцікавіше, що якби наш JS файл був би 40KB, то ми отримали б його на 100 мс раніше.

Нам може здаватися, що трохи більший розмір даних не впливає ні на що, якщо у користувачів швидкий інтернет, але ми бачимо, що це не так.

Тому слід використовувати Gzip компресію с HTTP, слідкувати за Cookie (вони можуть бути великими), стискати картинки і видаляти з них метадані. Звичайно, не забувати про CDN (може дати істотний виграш).

Далі я спробую описати детальніше, що в нас є, щоб зробити наші веб-застосунки швидшими.

Але є ще одна проблема, про яку все-таки варто сказати - «Head-of-line Blocking». Насправді коли говорять про «Head-of-Line Blocking», то можуть мати на увазі різне.

Є 2 варіанти цієї проблеми:

«Head-of-line Blocking» на рівні TCP

Ми розглянули ситуацію, коли у нас немає втрат пакетів, але на практиці пакети завжди губляться. Більш того, TCP Slow Start збільшує швидкість поки не почнуть губитися пакети, потім значно зменшує швидкість і починає підіймати повільніше.

Втрати пакетів можуть призводити до «Head-of-line Blocking» на TCP рівні.

Спробуємо описати основну ідею.

TCP відповідає за те, щоб пакети прийшли в додаток у правильному порядку. Якщо сервер відправив: [1][2][3][4][5], а отримали ми (або в іншому порядку) [2][3][4][5].

То ці пакети перебувають у TCP буфері одержувача, поки сервер відправляє нам повторно пакет [1]. Тобто, завдання TCP-протоколу вибудувати пакети в правильну чергу перед тим, як вони потраплять у додаток. Це зручно, але далеко не завжди потрібно.

«Head-of-line Blocking» на рівні HTTP/1.x

Тут трохи інша ситуація.

Припустимо, нам потрібно зробити 10 HTTP-запитів. Браузер надсилає запити один за одним і виходить, щоб надіслати новий, він має дочекатися результату попереднього.

Схематично це виглядає так:

50ms: Kyiv ------запит 1---→ SF
20ms: робота бекенда (запит 1)
50ms: Kyiv ←----відповідь 1------ SF
50ms: Kyiv ------запит 2---→ SF
20ms: робота бекенда (запит 2)
50ms: Kyiv ←----відповідь 2------ SF
50ms: Kyiv ------запит 3---→ SF
20ms: робота бекенда (запит 3)
50ms: Kyiv ←----відповідь 3------ SF

Для спрощення я проґавив усі моменти, пов’язані зі встановленням з’єднання (TCP-handshake, TLS-handshake, TCP Slow Start).

У зв’язку з цим, у HTTP/1.1 з’явився «HTTP Pipelining». Суть - відправити одразу пачку запитів і чекати відповіді. «HTTP Pipelining» має такий вигляд:

50ms: Kyiv ------запит 1---→ SF
Kyiv ------запит 2---→ SF
Kyiv ------запит 3---→ SF
20ms: робота бекенда (запит 1)
робота бекенда (запит 2)
робота бекенда (запит 3)
50ms: Kyiv ←----відповідь 1------ SF
Kyiv ←----відповідь 2------ SF
Kyiv ←----відповідь 3------ SF

Це корисна штука (120мс проти 360мс), але на практиці вона відключена в більшості браузерів через те, що реалізації серверів часто містять баги. Але навіть якби це працювало, все одно ми маємо проблему «Head of line blocking»: якщо обробка першого запиту триватиме 1 секунду, то відповіді не зможуть повернутися раніше ніж за секунду (оскільки перший запит блокує повернення інших).

Так, браузер може паралельно відкривати 4-6 з’єднань (це з налаштуваннями за замовчуванням), але це лише частково рятує ситуацію.

Проблеми з DNS.

- У 99% випадків для DNS використовується UDP (за рідкісними винятками, коли відповідь не влазить у датаграму, тоді може бути ініційоване TCP-з’єднання). Тобто нам майже ніколи не потрібна установка з’єднання, що сильно зменшує нашу проблему. Питання безпеки поки що опустимо.

- Найімовірніше, ми звертаємося до DNS сервера провайдера і сервер цей розташований досить близько. Так, це все одно окремий запит, який теж впливає на те, як швидко користувач побачить сторінку, але в деталі поки що вдаватися не будемо.

Автор статті Віктор Турський, Senior Software Engineer at Google Non-Executive Director and co-founder at WebbyLab.

3. Вміння налаштувати доступ до сервера

3.1. Поняття, якими потрібно володіти

- SSH
- Асиметричні та симетричні шифри (RSA, AES)
- Який ключ (приватний чи публічний) використовується для цифрового підпису і чому?

3.2. Практичні завдання для виконання

1. Згенерувати пару SSH ключів в окремій директорії на локальному комп'ютері (не в .ssh)
2. Запустити ubuntu server через multipass - https://multipass.run
3. Додати публічний ключ на сервер для доступу по SSH
4. Увійти на сервер за ip-адресою, використовуючи ключ для аутентифікації

Важливо: робота з gitlab/github репозиторіями відбувається за таким самим принципом.

4. Шифрування/кодування/хешування. У чому різниця?

4.1. Поняття, якими потрібно володіти

- base64
- md5, sha1, sha256
- JWT (потрібно для практичної частини)

4.2. Практичні завдання для виконання

1. Без застосування зовнішніх бібліотек написати парсер JWT (без перевірки підпису). У браузері та NodeJS уже є функції для роботи з base64, їх можна використовувати.
2. Закодувати, а потім розкодувати рядок «Hello World» за допомогою CLI утиліти base64.
3. Подивитися всередину пейлоада JWT за допомогою CLI утиліти base64.
4. Порахувати хеш суму будь-якого файлу за допомогою утиліти md5sum.

5. Прочитати статтю про "Шифрування/кодування/хешування"

Шифрування/кодування/хешування.

Завдання розібратися з цими термінами. Дуже часто ми стикаємося з цим на фронтенді, бекенді і навіть мобільних додатках. Але у людей виникає плутанина в головах. Шифрування/кодування/хешування. Часто розробники плутають ці поняття. Наприклад, людина дивиться на JSON Web Token (JWT) і думає, що дані в ньому зашифровані. Або що логін і пароль у HTTP Basic Auth зашифрований, оскільки виглядає як набір випадкових символів.

Давайте розберемося в теорії, а потім подивимося на приклади.

Шифрування.

Що таке шифрування ми зазвичай всі розуміємо. Тут важливо тільки зауважити, що є симетричні шифри (для шифрування і розшифрування використовується один і той самий ключ) і асиметричні шифри (коли у нас є пара ключів, відкритий і закритий). Також асиметрична криптографія може використовуватися для цифрового підпису. Приклади: AES, chacha20, RSA

Хешування.

Основна ідея, що є якась функція (хеш-функція), яка перетворює довільної довжини набір даних у набір даних фіксованої довжини. Тобто, ми можемо 1ТБ захешувати в 10 байт (наприклад, порахувати контрольну суму даних). Головна відмінність від шифрування - це те, що хеш-функція працює в один бік. Ми не можемо з 10 байт контрольної суми потім отримати назад наші вихідні дані. Приклади: md5, bcrypt, MurmurHash

Кодування. Кодування не має на меті приховування інформації, а просто представлення даних в іншій формі. Зазвичай це робиться для зручної передачі, зручнішого зберігання тощо. (можна шифрування та стиснення віднести до кодування, але часто саме кажуть «стиснення», «шифрування», оскільки тоді одразу зрозумілі додаткові властивості алгоритмів кодування). Приклади: base64, multipart/form-data, urlencoded

Часті непорозуміння.

- **JWT пейлоад закодований base64 і підписаний, але не зашифрований**

JWT складається з трьох секцій, розділених крапкою. header.payload.signature. Кожна частина закодована base64. Ідея base64 в тому, щоб бінарні дані представити у вигляді друкованих символів таблиці ASCII і відповідно клієнт може зчитати всі дані з JWT.

- **Паролі в базі хешуються, а не шифруються**

Паролі в базі мають зберігатися у вигляді хешів. З хеша не можна отримати пароль назад (тільки перебором) і для ускладнення перебору використовуються сіль і хеш-функції, призначені для хешування саме паролів (scrypt, argon2 etc)

- **HTTP BasicAuth кодує логін і пароль у base64, але не шифрує**

Передає в base64 = передає у відкритому вигляді. Digest Authentication працює по іншому і використовує вже md5 хешування.

- **У SSH під час аутентифікації за ключем ми не передаємо приватний ключ на сервер**

Якщо вас просять надати приватний і публічний ключ, щоб налаштувати доступ до сервера, то ніколи не давайте приватний. Потрібен тільки публічний. Крім того, навіть під час аутентифікації, приватний ключ завжди залишається тільки на вашому комп'ютері.

- **Для шифрування трафіку в HTTPS використовується симетричний шифр, а не пара з публічного і приватного ключів**

Багато хто думає, що для шифрування трафіку використовується пара з публічного і приватного ключів, але насправді використовується сесійний ключ і симетричний шифр (типу AES або Chacha20). Публічний і приватний ключ використовується тільки під час встановлення TLS-з'єднання

5. Вміння використовувати Проміси (Promises)

5.1. Поняття, якими потрібно володіти

- Promise
- async/await
- try/catch
- Що повертає метод then?
- Чи можна замінити catch методом then?
- Чи може ми робити throw new Error(«») або ми повинні робити тільки return Promise.reject(new Error(«")) в async функціях?

5.2. Практичні завдання для виконання (на javascript)

1. Написати функцію sleep. Використовуватися буде так «await sleep(ms)», основний потік не блокує.

6. Уміння аналізувати алгоритмічну складність свого коду

6.1. Поняття, якими потрібно володіти

- Big O нотація (Time complexity, Space complexity)
- O(1) vs O(n) vs O(n^2) vs O(log(n)) vs O(n*log(n))
- Чому O(n^2) алгоритм може виконується швидше, ніж O(n*log(n))
- Hash table. Це чи не найбільш ключова структура даних у computer science. Знати, як реалізувати самостійно. Алгоритмічна складність різних операцій.

6.2. Практичні завдання для виконання

1. Переглянути доповідь Володимира Агафонкіна про алгоритмічну складність.
2. Є масив messages на 2000 повідомлень і є масив users на 100 користувачів. У кожному повідомленні є userId. Завдання повернути список повідомлень, але в кожне повідомлення додати ще ім'я користувача. Важливо: складність алгоритму має бути O(messagesCount + usersCount)

```
// Приклад інпуту
const messages = [ {id: 1, text: "Hello", userId: 1}];
const users = [ {id: 1, userName: "Artem"} ];

// Приклад результату
const messagesWithUsernames = [
   {id: 1, text: "Hello", userId: 1, userName: "Artem"}
];

// Необхідно реалізувати таку функцію
function prepareMessages(messages, users) {}
```

3. Реалізувати бінарний пошук і виміряти продуктивність (результати заміру теж закоммітити в репозиторій)

```
//Функція для створення відсортованого масиву чисел
function createArray(count) {
    const array = [];

    for (let i = 0; i < count; i++) {
        array.push(i*2);
    }

    return array;
}

// Ініціалізуємо масив для пошуку
const array = createArray(50_000_000);
const searchValue = 1000_000_000;
```

```
// Виміряти швидкість виконання пошуку через some
const result1 = array.some(v => v === searchValue);

// Виміряти швидкість виконання пошуку через binarySearch

// Функцію binarySearch необхідно реалізувати
// Time O(log n), Space O(1)
const result2 = binarySearch(array, searchValue);
```

4. Реалізувати самостійно хеш-таблицю. Це завдання доповнює розділ про хешування/шифрування/кодування. По суті потрібно написати найпростішу хеш-функцію (не використовувати готові) і зробити клас на кшталт Map (ключ тільки строковий). Важливо врахувати можливість колізій. Найпростіша хеш-функція буде виглядати так - підсумовуємо коди символів і беремо залишок від ділення на довжину масиву.

```
const user = new HashTable({size: 20});

user.set('email', 'sudent@chnu.edu.ua');
user.set('firstName', 'John');
user.set('lastName', 'Doe');
user.set('company', 'Some Company');
user.get('email');
user.get('firstName');
user.get('lastName');
user.get('company');
```

7. Розуміння роботи операційної системи та докера

7.1. Поняття, якими потрібно володіти

- Переглянути доповідь Діпдайв у докер для JavaScript розробників. Доповідь може здатися трохи складною, але ці знання досить фундаментальні, щоб бути корисними під час розв'язання абсолютно різних завдань.

7.2. Практичні завдання для виконання

1. Встановити docker
2. Запустити контейнер ubuntu:latest
3. Усередині контейнера встановити htop
4. Запустити htop і подивитися на список процесів

8. Робота з базами даних

8.1. Поняття, якими потрібно володіти

- SQLite/Postgres/MySQL
- Індекс у базі даних (і як влаштовано B-tree).
- Інвертований індекс (або повнотекстовий індекс)

8.2. Практичні завдання для виконання

1. Запустити mysql і adminer через docker (нижче готовий docker-compose). Усі подальші завдання можна зробити через веб-інтерфейс adminer

```yaml
version: '3.3'
services:
   db:
       image: mysql:latest
       command: --default-authentication-plugin=mysql_native_password
       restart: always
       ports: ['3306:3306']
       environment:
           MYSQL_DATABASE: mydb
           MYSQL_ROOT_PASSWORD: password
   adminer:
       image: adminer
       restart: always
       ports: ['8888:8080']
```

2. Створити базу даних з однією таблицею products на 3 колонки: id (autoincrement) , name, description.
3. Імпортувати файл на 1000 записів із випадковими даними (1k_records.sql).
4. Виконати запит, який знаходить продукти з назвою «Incredible Fresh Hat Awesome Concrete Shirt». Записати час виконання запиту.
5. Імпортувати файл на 10 млн записів (10mln_records.sql). Імпорт файлу займає 20-60 хвилин.
6. Повторити запит на отримання продуктів з name рівним «Incredible Fresh Hat Awesome Concrete Shirt». Записати час виконання.
7. Додати індекс за полем name.
8. Повторити запит на отримання продуктів з name рівним «Incredible Fresh Hat Awesome Concrete Shirt». Записати час виконання. Порівняти час виконання з цим же запитом, але до додавання індексу. У скільки разів різниця?
9. Порахувати кількість продуктів, у яких назва починається на «Handmade Soft Keyboard». Записати час виконання запиту.

10. Порахувати кількість продуктів у яких у назві є фраза «Soft Keyboard Generic». Записати час виконання запиту. Порахувати у скільки разів він повільніший за запит із попереднього пункту. Чому цей запит працює повільно, якщо є індекс? Як зробити запит швидшим?

*Як імпортувати дані в mysql, який запущений у контейнері? * Це можна зробити наступною командою docker exec -i cf1606d409ba mysql -uroot -ppassword mydb < data.sql Тільки потрібно підставити інше ім’я контейнера (або ідентифікатор)

9. Знання про основні вразливості та методи захисту

9.1. Поняття, якими потрібно володіти

- XSS (cross site scripting)
- HTML Sanitizer
- CSRF (cross site request forgery)
- SQL injection
- IDOR (insecure direct object referencing)
- HTTP проти HTTPS

9.2. Практичні завдання для виконання

1. Зробити HTML сторінку для демонстрації XSS. Спочатку файлу визначаємо змінну COMMENT зі шкідливим коментарем, який показує алерт. Потім відображаємо вміст змінної в 3-х блоках:
   a. Блок 1: вразливий рендеринг
   b. Блок 2: захищений рендеринг
   c. Блок 3: захищений рендеринг, але з підтримкою форматування (теги «b», «i», «img»)
2. У нас є запит до бази «SELECT * FROM users WHERE email=${email} AND password=${password}». Завдання написати SQL ін’єкцію, яка дасть змогу зайти під користувачем admin, не знаючи його пароля.
3. Експлуатація CSRF. Встановити в кукі змінну authenticated=1 для одного домену (наприклад, mysite.localhost). Зробити html-сторінку, яку потрібно відкривати з іншого домену (наприклад, attacker.localhost) і з цієї сторінки зробити сабміт html-форми на mysite.localhost. Перевірити, чи надсилаються в запиті зі сторінки attacker.localhost куки для домену mysite.localhost. Домени можна налаштувати в /etc/hosts

10. Web vitals (page speed insights)

10.1. Поняття, якими потрібно володіти

- Web Vitals - https://web.dev/vitals

10.2. Практичні завдання для виконання

1. Запустити аналіз https://www.chnu.edu.ua/ у Page Speed Insights (https://developers.google.com/speed/pagespeed/insights/)
2. Запустити аналіз https://www.chnu.edu.ua/ у Chrome Devtools Lighthouse

11. Уміння користуватися профайлером

Важливо, що цей же профайлер використовується і для NodeJS.

1. Якщо ви Frontend розробник, то запускайте профайлер у Chrome.
2. Якщо ви NodeJS розробник, то потрібно використовувати --inspect під час запуску nodejs застосунку https://nodejs.org/en/docs/guides/debugging-getting-started/
3. Якщо ви php розробник використовуйте будь-який звичний для вас профайлер (xhprof, blackfire, xdebug)

11.1. Поняття, якими потрібно володіти

- Профайлер
- Self time vs Total time

11.2. Практичні завдання для виконання

1. Підключись до свого проекту через chrome dev tools.
2. Запусти JavaScript Profiler (окрема вкладка, не Performance), зроби якусь дію в застосунку (або запит до бекенду, якщо це бекенд) і знайди найтривалішу операцію за допомогою профайлера.
3. Подивися на вкладку memory, спробуй погратися з різними опціями.
4. Додатково подивися вкладку Performance (працює тільки для фронтенда).

12. Архітектура стандартного веб-застосунку

12.1. Поняття, якими потрібно володіти

1. CDN
2. MySQL/Postgres

3. Vertical scaling (Вертикальне масштабування)
4. Horizontal scaling (Горизонтальне масштабування)
5. Autoscaling
6. Load Balancing (Балансування навантаження)
7. AWS S3
8. C4 Model (https://c4model.com/, подивитися 30 хвилинне відео на головній)

12.2. Практичні завдання для виконання

1. Намалювати діаграму solution архітектури Twitter-like (мікроблог на пости по 140 символів) застосунку. Інструмент для малювання та підхід можна використовувати будь-який. Можна використати c4 containers diagram і plantuml розширення для малювання https://github.com/plantuml-stdlib/C4-PlantUM (є і чудовий plantuml плагін для vscode), але можна малювати в чому подобається. Приклади https://c4model.com/diagrams

13. Кешування

13.1. Поняття, якими потрібно володіти

1. LRU Cache
2. Memcached

13.2. Практичні завдання для виконання

1. Необхідно реалізувати клас «DNSResolver», який опціонально прийматиме «cacheOptions» і використовуватиме LRU cache replacement policy (взяти готову імплементацію LRU кешу на npm) для кешування DNS запитів.

```
// Приклад використання
const dnsResolver = new DNSResolver({cacheOptions: {
   max: 100,
   maxAge : 1000 * 60 * 60 // 1 hour
}});

// Повертає перший ipv4 (із «A» запису)
// Якщо запису немає, то повертає null
const ipv4 = await dnsResolver.resolve('google.com');
```

14. Базові поняття для процесу розробки

14.1. Поняття, якими потрібно володіти

- SDLC (Software Development Life Cycle) - методологія, яка структурує процес створення програмного забезпечення.
- Agile (https://agilemanifesto.org/) - це сукупність підходів і моделей поведінки, орієнтованих на використання ітеративної розробки, time boxes (часових рамок), динамічне формулювання вимог і забезпечення реалізації ПЗ в результаті взаємодії всередині високо самоорганізованої робочої групи із фахівців різних профілів.
- Scrum vs Kanban vs Waterfall - це три різні методології управління проектами, які використовуються для організації роботи в командах, особливо в сфері розробки програмного забезпечення. Вони відрізняються підходами до планування, виконання та контролю проектів.
- CI/CD - є поширеною DevOps-практикою. CI (Continuous Integration) — це неперервна інтеграція, а CD (Continuous Delivery) — неперервна доставка. Цей набір методик дозволяє розробникам частіше і надійніше розгортати зміни в програмному забезпеченні.
- Функціональні вимоги vs нефункціональні вимоги.

Характеристика	**Функціональні вимоги**	**Нефункціональні вимоги**
Що описують	Які дії або функції повинна виконувати система	Як система повинна працювати, характеристики її якості
Приклади	Логін, реєстрація, пошук, додавання в кошик	Продуктивність, безпека, масштабованість, час відгуку
Мета	Забезпечити функціональність системи	Забезпечити якісні характеристики роботи системи
Фокус	Що робить система	Як добре система це робить
Вимірюваність	Зазвичай легко перевірити (завдання виконане чи ні)	Може бути складно перевірити (вимагає тестування якості, наприклад, стрес-тестів)

Різниця:

- Функціональні вимоги відповідають за основну поведінку та можливості системи.
- Нефункціональні вимоги визначають, як добре або як ефективно система повинна виконувати свої функції.
- TDD vs BDD - підходи до розробки пз, коли спочатку пишуться тести, а потім код.
    - DD (щось Driven Development) - розробка, заснована на чомусь.
    - TDD (Test Driven Development) - розробка на основі тестів.
    - BDD (Behavior Driven Development) - розробка на основі поведінки.

- Функціональні вимоги vs нефункціональні вимоги.
- Власник продукту (Product owner)
- Зацікавлена сторона - це окрема особа, група чи організація, на яку може вплинути, подіяти (позначитися) або прийняти на себе вплив, рішення, діяльність чи результат проекту. Зацікавлені сторони або активні учасники у проектах, або їхні інтереси можуть вплинути на результат проекту. Зазвичай це включає членів команди проекту: керівників проектів, спонсорів проекту, керівників, клієнтів або користувачів.
- Three-Point Estimation (Трибальна оцінка) - суть цього методу полягає в тому, щоб знайти найкращі та найгірші умови роботи для вашої команди.

14.2. Практичні завдання для виконання

Написати 5 прикладів Functional і 5 прикладів Non functional requirements для Twitter-like застосунку.