

Міністерство освіти і науки України
Національний університет "Запорізька політехніка"

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт з дисципліни
"WEB-програмування"
для студентів спеціальності
122 "Комп'ютерні науки"
всіх форм навчання

2019

Методичні вказівки до виконання лабораторних робіт з дисципліни "WEB-програмування" для студентів спеціальності 122 "Комп'ютерні науки" всіх форм навчання / Укл. Скрупський С.Ю., Шкарупило В.В. – Запоріжжя: НУ "Запорізька політехніка", 2019. – 56 с.

Укладачі: С.Ю. Скрупський, к.т.н., доцент
В.В. Шкарупило, к.т.н., доцент

Рецензент: А.О. Олійник, к.т.н., доцент

Відповідальний за випуск: С.Ю. Скрупський, к.т.н., доцент

Затверджено:
на засіданні кафедри
"Програмні засоби"
Протокол № 1 від 30.08.2019

Рекомендовано до видання
НМК ФКНТ
Протокол № 2 від 02.09.2019

ЗМІСТ

1 ЛАБОРАТОРНА РОБОТА № 1. РОЗМІТКА МАКЕТУ ВЕБ-СТОРІНКИ ЗА ДОПОМОГОЮ HTML5 ТА CSS3	4
2 ЛАБОРАТОРНА РОБОТА № 2. РОЗРОБКА SOAP WEB-SERVICES	7
3 ЛАБОРАТОРНА РОБОТА № 3. РОЗРОБКА XML-RPC WEB-SERVICES	14
4 ЛАБОРАТОРНА РОБОТА № 4. ОСНОВИ РОБОТИ З БІБЛІОТЕКОЮ REACT	19
5 ЛАБОРАТОРНА РОБОТА № 5. АРХІТЕКТУРНИЙ СТИЛЬ REST І ТЕХНОЛОГІЯ GRAPHQL	39
ЛІТЕРАТУРА	56

1 ЛАБОРАТОРНА РОБОТА № 1. РОЗМІТКА МАКЕТУ ВЕБ-СТОРІНКИ ЗА ДОПОМОГОЮ HTML5 ТА CSS3

Мета роботи: ознайомитися з підходами та технологіями, що використовуються для розмітки сучасних веб-сторінок, та розмітити веб-сторінку за допомогою мови розмітки HTML5 та каскадних таблиць стилів CSS3.

Коротка теорія. Розмітка (верстка) веб-сторінки – це процес створення веб-сторінки засобами мови HTML та CSS із попередньо створеного макету дизайну сайту, заздалегідь намальованого за допомогою графічних редакторів.

Таблична розмітка раніше була основним методом створення веб-сторінки (HTML3). Однак сьогодні вона використовується в основному лише для відображення табличних даних, або в системах CRM, що відправляють електронні листи. Це пов'язано з тим, що не всі поштові сервери мають змогу відображати стилі CSS в електронних листах.

Розмітка за допомогою блоків (тег `<div>`) та каскадних таблиць стилів (CSS), що описують їх оформлення, реалізує концепцію семантичної розмітки. При цьому розділяється зміст сторінки та її оформлення. Концепція семантичної розмітки знайшла найбільше втілення в мові HTML5. Блоки являють собою структурні елементи, які можна розмішувати на веб-сторінці шляхом накладання їх один на одного з точністю до пікселя.

Скрипти (наприклад, на мові JavaScript) дозволяють динамічно змінювати параметри блоків. Блоки можна переміщувати, ховати та відображати без перезавантаження всієї сторінки. Це надає можливість створювати на сторінці різні ефекти, такі як анімація, спадаюче меню, спливаючі підказки, плаваючі вікна тощо.

Кожна веб-сторінка, як правило, містить наступні стандартні блоки:

- header;
- logo;
- navigation/menu;
- content area;
- footer.

Крім того, основа макету сторінки зазвичай складається з однієї, двох, трьох або чотирьох колонок. Всі макети сайтів, тобто типи розмітки сторінок можна розділити на такі принципові групи:

- фіксовані (fixed),
- гумові (fluid),
- респонсивні (responsive),
- адаптивні (adaptive).

Фіксований тип макету – дизайн, в якому ширину елементів сторінки (стовпців, рисунків) задано точно в пікселях. Переваги: легко розробляти, заздалегідь відомо, як буде виглядати сайт. Недоліки: для сайту існує лише один ідеальний розмір екрану, в деяких випадках може з'явитися скролінг.

Гумовий тип макету – дизайн, в якому ширину елементів сторінки задано у відсотках від поточного розміру екрану чи батьківського елемента. Переваги: сайт буде заповнювати весь простір браузера, що значно поліпшує його вигляд, сайт буде однаково виглядати на різних розмірах екрану. Недоліки: розробка такого дизайну дуже складна, на великих екранах можлива поява дуже довгих рядків.

Для підлаштування макету під розмір екрану пристрою використовують респонсивний та адаптивний типи макетів. Адаптивний дизайн сегментує користувачів на категорії за типом пристрою (desktop, tablet, mobile). На сервері в залежності від цього готується відповідна версія дизайну. Респонсивний тип макету – дизайн, який динамічно масштабує вміст сторінки під розмір екрану, в тому числі може відбуватися переміщення блоків з одного місця на інше, або їх заміна іншими блоками, що відображаються лише при визначеній роздільній здатності. Респонсивна верстка прийшла на зміну ідеї створення спеціальних мобільних версій сайтів, розташованих на окремих піддоменах (наприклад, m.wikipedia.org). Переваги: сайт буде відображатися на різних розмірах екрану так, як це найбільш зручно користувачеві. Недоліки: потребує старанного опрацювання декількох макетів для різних розмірів екранів. Респонсивний дизайн будується за допомогою медіа запитів (media queries), флексбокс (flexbox) елементів та на так званих решітках (grid-layouts). Також може використовуватися змішаний тип дизайну.

Валідність HTML-верстки – це її відповідність стандартам організації The World Wide Web Consortium (W3C). Відсутність

помилки в розміщенні документу – це один з головних показників якості розмітки. Перевірка розмітки на помилки та відповідність стандарту може бути проведена автоматично за допомогою онлайн сервісу W3C чи за допомогою різних програм "валідаторів".

Порядок виконання роботи.

1. Навчитися позиціювати блочні елементи сторінок. Навчитися формувати макети веб-сторінок фіксованої ширини та такі, що розтягуються; макети з 1, 2 чи 3 колонок.

2. Обрати безкоштовно розповсюджуваний макет сайту в форматі PSD та розмітити за допомогою блоків веб-сторінку.

3. Перевірити відображення: макету в різних браузерах. При роботі для налагодження макету скористатися вбудованими в браузері інспекторами коду.

4. Провести валідацію макету.

Зміст звіту.

1. Результати роботи: загальний вигляд макету, схематичне подання макету, в якому відобразити виділені змістовні блоки, повний код HTML та CSS.

2. Окремо виділити елементи, що формують адаптивний чи респонсивний тип верстки.

3. Відповіді на контрольні питання.

Контрольні питання.

1. Що таке HTML та CSS? В чому полягає підхід розмежування змісту та подання веб-сторінки?

2. З чого складається HTML документ? Що таке тег та елемент документу? Наведіть приклади тегів.

3. В чому різниця між block та inline елементами HTML сторінки? Наведіть приклади.

4. Які недоліки та переваги розмітки сторінок за допомогою таблиць та блоків?

5. Які є способи додання стилів до документу?

6. Які типи селекторів використовуються в CSS?

7. Що мається на увазі під каскадним застосування стилів до елементів документу?

8. Що таке крос-браузерна верстка?

2 ЛАБОРАТОРНА РОБОТА № 2. РОЗРОБКА SOAP WEB-SERVICE

Мета роботи: познайомитися зі специфікацією SOAP. Навчитися створювати розподілені застосунки за допомогою Java реалізації SOAP.

Коротка теорія. Розподіленим називається застосунок, окремі компоненти якого працюють на різних комп'ютерах і використовують різні мережеві засоби. Вони взаємодіють так, що застосунок виглядає як єдине ціле, ніби всі його компоненти розташовані на одній машині. Найпростіша архітектура розподіленого застосунка, яка називається архітектурою клієнт-сервер, передбачає, що застосунок складається з двох частин: серверної частини, яка надає послуги, і клієнтської частини, яка користується послугою. У Web-застосунку, побудованому по архітектурі клієнт-сервер, послуги надає Web-сервер, а клієнтом служить браузер, підключений до Інтернету, або інший клієнтський застосунок, пов'язаний з Web-сервером. В архітектурі клієнт-сервер дуже важливо правильно розділити роботу між клієнтом і сервером. Можна зробити клієнта "тонким", який тільки відображає результати запиту. Це зручно для організації клієнта. Його можна розмістити на найпростішому дешевому комп'ютері. Йому не потрібно складного програмного забезпечення, в більшості випадків досить звичайного браузера. Але тоді сервер стає "товстим", на нього лягає велике навантаження, йому доводиться виконувати всі запити до джерела даних і всю обробку цих даних. Можна, навпаки, зробити клієнта "товстим", виконуючим всю обробку результатів запиту, а сервер "тонким", який тільки розсилає необроблені дані клієнтам. В цьому випадку для клієнта потрібен потужний дорогий комп'ютер. В разі зміни алгоритму обробки даних або виявлення помилок доведеться оновлювати програмне забезпечення на всіх клієнтських машинах.

Для того щоб позбутися цих недоліків архітектури клієнт-сервер, програми, які обробляють дані, виділяють в окремий, проміжний (middleware) шар програмного забезпечення. Цей шар може працювати на тій же машині, що і серверний шар, працювати на іншій машині або навіть на декількох машинах. Розподілений додаток стає тришаровим. В технології Java проміжний шар зазвичай

реалізований сервером застосунків (application server). Випускається багато промислових серверів застосунків: BEA WebLogic, JBoss, IBM WebSphere, IIS (Internet Information Services) та інші

Кількість шарів можна збільшувати, але важливіше встановити надійний і швидкий зв'язок між усіма компонентами розподіленого застосунку. Добре, якщо всі компоненти створені для однієї платформи, але така ситуація рідко зустрічається в Інтернеті. Набагато частіше компоненти розподіленого застосунку працюють на різних платформах: клієнтська частина розроблена для MS Windows, Linux або Apple Mac, серверна частина – для Solaris, Linux, Free BSD або для інших UNIX. Більш того, зараз спостерігається явна тенденція створювати додатки, незалежні від якої б то не було платформи. Вся технологія Java створена в руслі цієї тенденції.

Розподілений застосунок, компоненти якого можуть працювати на різних платформах і замінити один одного, називається слабо пов'язаним застосунком. Наприклад, браузер і Web-сервер слабо пов'язані один з одним. Вони не тільки можуть працювати на різних платформах, а й обмінюються найрізноманітнішою інформацією з різними MIME-типами. Крім того, зв'язок між ними дуже короткий, він складається тільки з запиту і відповіді. Фактично, спільне у браузера і Web-сервера тільки те, що вони працюють по одному протоколу HTTP і те, що вони повинні бути на зв'язку одночасно. Для слабо пов'язаних додатків необов'язкова навіть остання умова. Вони можуть працювати асинхронно, їх компоненти можуть виходити на зв'язок в зручний для них час. Так працює електронна пошта, яку можна вважати класичним прикладом слабо пов'язаного застосунка.

Творці Web Services вирішили, що Web-послуги будуть надаватися в рамках слабо пов'язаних застосунків. Вони вирішили взяти за основу тільки спільні засоби, наявні на всіх платформах. Таким "найменшим спільним знаменником" різних обчислювальних платформ, які використовують WWW, виявилася мова HTML і протокол HTTP. Однак, мови HTML явно недостатньо для опису викликів розподілених процедур і методів розподілених об'єктів. Тому зручно використовувати мову XML. Дійсно, вона не залежить від платформи. Засоби обробки документів XML, є на всіх платформах. Документи XML легко передаються по мережі будь-яким прикладним протоколом, оскільки це байтовий ASCII-текст.

Таким чином, Web Services – це послуги, що надаються з використанням тієї чи іншої реалізації мови XML і протоколу HTTP. Однак Web Service – це не всяка послуга, яка надається за допомогою Web-технології. Пересилання файлів, навіть XML-документів, до неї не належить. Звичайними Web-послугами користуються незалежні програми: браузері, пов'язані з Web-сервером тільки на час надання послуги. На відміну від них послуги, названі "Web Services", надаються, як правило, в рамках розподіленого застосунка. Можна сказати, що звичайні Web-послуги надаються клієнту-людині, а Web Services – це послуги, що надаються клієнту-програмі. Як правило, Web Service застосовується як компонент розподіленої інформаційної системи, розкиданої по комп'ютерам з різною архітектурою і різними засобами мережевого зв'язку.

Класичними і перевіреними протоколами Web Services вважаються XML-RPC і SOAP, оформлені як реалізація мови XML. Протокол SOAP виник в 1998 році в фірмі UserLand і корпорації Microsoft, але потім його розробка була передана в консорціум W3C. Протокол SOAP не розрізняє виклик методу і відповідь на нього, а просто визначає формат послання (message) у вигляді документа XML. Послання може містити виклик методу, відповідь на нього, запит на виконання якихось інших дій або просто текст. Специфікацію SOAP не цікавить зміст послання, вона задає тільки його оформлення. Кореневий елемент документа XML <Envelope> містить необов'язковий заголовок <Header> і обов'язкове тіло <Body>. Схема SOAP-послання така:

```
<?xml version='1.0'?>
<env:Envelope
xmlns:env="http://www.w3.org/2002/06/soap-
envelope">
  <env:Header>
    <!-- Блоки заголовку -->
  </env:Header>
  <env:Body>
    <!-- Вміст послання -->
  </env:Body>
</env:Envelope>
```

У заголовку міститься один або кілька блоків, оформлення і зміст яких ніяк не регламентуються. Так само нічого не говориться про зміст тіла послання. Проте, розрізняють процедурний стиль

повідомлення SOAP, призначений для виклику віддалених процедур, і документний стиль, призначений для обміну документами XML. Процедурний стиль часто називають RPC-стилем, а документний стиль - XML-стилем.

При процедурному стилі повідомлення SOAP при виклику віддалених процедур в заголовку повідомлення встановлюють параметри виклику, наприклад, номер транзакції, в якій виконується процедура. У тілі повідомлення перераховуються аргументи процедури, що викликається.

При документному стилі повідомлення SOAP повідомлення містить не аргументи виклику якоїсь процедури, а просто документ XML або структуру даних.

Однією з класичних та найпопулярніших реалізацій SOAP для Java є сервер Axis (<http://xml.apache.org/axis/>), до складу якого входить SOAP-сервер – Java-клас SimpleAxisServer.

Порядок виконання роботи.

1. Пропишіть змінну середовища свого користувача path в папку bin встановленої jdk.

2. В папки C:\Program Files\Java\jdk\jre\lib\ext та C:\Program Files\Java\jre\lib\ext необхідно скопіювати всі файли з директорії Axis lib (яку можна взяти на офіційному сайті Apache Axis <http://apache.cp.if.ua/axis/axis/java/1.4/>), а також файли tools.jar (з папки C:\Program Files\Java\lib).

Примітка: приклади коду в цих методичних вказівках продемонстровані для канонічної версії Apache Axis1. В даний час використовується Apache Axis2, який експлуатує ту ж ідею Web-сервісів на основі SOAP, що і Apache Axis1. Ви можете використовувати як першу, так і другу версію Axis для того, щоб зрозуміти ідею SOAP.

3. Для створення Web-служби, яка буде працювати під управлінням Axis, досить написати класи Java, які реалізують Web-службу, дати файлу, що містить вихідні тексти цих класів, розширення ".jws", і покласти цей файл в каталог axis сервера застосунків (наприклад, C:\axis\webapps\axis). Папку webapps візьміть з того самого архіву <http://apache.cp.if.ua/axis/axis/java/1.4/>. Крім того, в цю папку треба також прописати змінну середовища свого користувача path.

4. Створимо Web-службу MathService, що надає одну Web-послугу calc, що повертає клієнту добуток двох чисел, що передаються при SOAP-запиті клієнта.

Клас MathService, що містить метод calc, наведено в лістингу:

```
public class MathService
{
    public int calc(int a, int b)
    {
        int prod=a*b;
        return prod;
    }
}
```

Назвемо файл з вмістом лістингу MathService.jws і покладемо його в каталог axis сервера застосунків (наприклад, C:\axis\webapps\axis). Після цього Web-служба готова для використання. Не потрібно компілювати клас MathService, Axis зробить це сам при першому запиті до Web-служби.

5. Створимо клієнта Web-служби. У лістингу наведено код клієнта Web-служби MathService:

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import java.net.*;
public class MathServiceClient
{
    public static void main(String[] args)
        throws Exception
    {
        Service service = new Service();
        Call call = (Call)service.createCall();
        String endpoint =
            "http://localhost:8080/MathService.jws";
        call.setTargetEndpointAddress(
            new URL(endpoint));
        call.setOperationName("calc");
        int a=0; int b=0;
        try{
            a = Integer.parseInt(args[0]);
            b = Integer.parseInt(args[1]);
        }
        catch(Exception e)
        {System.out.println("Error: "+e);}
```

```

int prod =
(Integer)call.invoke(new Object[]{a,b});
System.out.println("Prod "+a+" * "+ b+ " = "
                    + sum);
}
}

```

У лістингу використано інтерфейси `service` і `call` пакета JAX-RPC. Вони реалізовані класами, що входять в Axis. Основну роль в отриманні Web-послуги грає об'єкт типу `call`. Він містить адресу Web-служби `endpoint` і назву Web-послуги `calc`. У нього заносяться аргументи Web-послуги у вигляді масиву об'єктів. У нашому простому прикладі масив аргументів складається тільки з двох елементів – двох цілих чисел `a` і `b`. Об'єкт `call` методом `invoke` повертає об'єкт, який містить результат виконання Web-послуги.

6. Створіть Java проект, в якому розмістіть файл `MathServiceClient.java`. Підключіть до проекту бібліотеки з папки `lib` серверу Axis.

7. Запуск застосунку. В папці з Web-службою `MathService.jws` запустіть `SimpleAxisServer`, виконавши в консолі (терміналі) команду:

```
java org.apache.axis.transport.http.SimpleAxisServer
```

2. В окремій консолі запустіть клієнта Web-служби, виконавши команду:

```
java MathServiceClient 10 100
```

Клієнта Web-служби можна запускати прямо з середовища розробки. В результаті на консоль клієнта буде виведено повідомлення:

```
Prod 10 * 100 = 1000
```

8. Моніторинг даних, якими обмінюються клієнт та сервер

Axis поставляється з корисною утилітою `tcpmon` (<http://archive.apache.org/dist/ws/tcpmon/1.0/>). Ця утиліта на основі пакету `Swing`, може використовуватися для установки TCP-тунелю і перегляду XML і HTML-документів, якими обмінюються клієнт і сервер розподіленого додатку.

Використання TCP - монітора продемонструємо на прикладі роботи Web-служби `MathService`. Завантажте TCP-монітор та розпакуйте папку `build`. Для запуску монітора запустіть `bat`-файл (або `sh`-файл під системою Linux).

Далі відкриється вікно, в якому треба ввести який порт треба прослуховувати та на який порт переправляти дані. Наприклад, сервер розміщено на порті 8080 (Target port). Клієнт підключається до порта 8081 (клієнт думає, що там розміщено сервер – Listen port). TCP-монітор перенаправляє запити з 8081 на 8080 зберігаючи тіло запиту та відповідь сервера. Не використовуйте одні і ті ж значення портів в полях *Listen Port* і *Target Port*, інакше ми увійдемо в нескінченний цикл. Після введення налаштувань натисніть кнопку *Add*. В результаті в основному вікні буде створена нова закладка з назвою *Port 8081*. Натисніть на закладку *Port 8081*, щоб переглянути її вміст. Оскільки ми встановили утиліту *tcpmon* на прослуховування порту 8081, нам потрібно змінити нашого клієнта так, щоб він передавав свої запити на порт 8081. Для цього нам потрібно змінити один рядок в коді нашого клієнта Web-служби *MathService*. З цією метою відкриємо файл *MathServiceClient.java* і замінимо його рядок:

```
String endpoint =  
"http://localhost:8080/MathService.jws";
```

на

```
String endpoint =  
"http://localhost:8081/MathService.jws";
```

Файл *JWS* повинен залишитися незмінним. Запустіть *SimpleAxisServer*. Скомпілюйте і запустіть змінений клієнта *MathServiceClient*. Після виконання клієнтського застосунку вікно утиліти *tcpmon* має містити XML-документи, якими обмінюються клієнт та сервер.

Зміст звіту.

1. Код клієнта та сервера Web-сервісу.
2. Код виконання індивідуального завдання, що видається викладачем.
3. Відповіді на контрольні питання

Контрольні питання

1. Який застосунок називають розподіленим?
2. Які особливості мають «товсті» і «тонкі» компоненти архітектури клієнт-сервер?
3. Що являє собою проміжний (middleware) шар веб-програмного забезпечення?
4. Дайте визначення Web Services.
5. Що являє собою SOAP?

3 ЛАБОРАТОРНА РОБОТА № 3. РОЗРОБКА XML-RPC WEB-SERVICE

Мета роботи: познайомитися зі специфікацією XML-RPC. Навчитися створювати розподілені веб-сервіси за допомогою Java реалізації XML-RPC.

Коротка теорія. Технологія WebServices для створення XML-документів обміну інформацією використовує два стандарти SOAP і XML-RPC. XML-RPC був створений в 1998 р корпорацією Useful спільно з Microsoft і вважається попередником SOAP. Специфікація протоколу XML-RPC проста, вона містить не більше двох десятків елементів XML. Базові типи завжди включені в елемент <value>. Масив визначається з використанням елемента <array>, який містить елемент <data>. Елемент <data> містить список змінних. Подібно до інших типів даних XML-RPC елемент <array> повинен бути укладеним в елемент <value>. Наприклад, наступний масив містить 3 рядки:

```
<value>
  <array>
    <data>
      <value><string>This</string></value>
      <value><string>is</string></value>
      <value><string>XML-RPC</string></value>
    </data>
  </array>
</value>
```

XML-RPC-запит є комбінацією XML-документа і заголовків HTTP. XML-документ визначає зміст XML-RPC-запиту і описує метод, що викликається та його параметри. Заголовки HTTP забезпечують передачу даних запиту по мережі. Кожен XML-RPC-запит містить тільки один XML-документ. Його кореневим елементом є елемент <methodCall>. Елемент <methodCall> в свою чергу містить елементи <methodName> і <params>. Елемент <methodName> ідентифікує ім'я методу, що викликається. Елемент <params> містить список параметрів методу і їх величини (елемент <param> і <value>). Заголовки HTTP відображають параметри відправника і вмісту. Наприклад XML-RPC-запит до типового Web-сервісу виглядає наступним чином:

```
POST /xmlrpc HTTP 1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169
```

```
<?xml version="1.0"?>
<methodCall>
  <methodName>method</methodName>
  <params>
    <param>
      <value><double>3.14</double></value>
    </param>
  </params>
</methodCall>
```

Формат XML-RPC-відповіді подібний до формату XML-RPC-запиту з невеликими змінами. Так в XML-документі відсутні елементи `<methodCall>` і `<methodName>`. При цьому, кореневий елемент `<methodCall>` замінюється елементом `<methodResponse>`. Крім того, XML-RPC-відповідь може містити лише один параметр; цей параметр може бути масивом, тому можлива передача багатьох величин; наявність параметра, що вертається, для XML-RPC-відповіді є необхідною умовою. У той же час, заголовки HTTP відповіді вказують, що для його формування використовується код 200 OK. Крім того, заголовки XML-RPC-відповіді: містять вказівку на те, що підтримується як протокол HTTP 1.0, так і HTTP 1.1; заголовок Content-Type повинен мати значення - text/xml; заголовок Content-Length повинен вказувати довжину відповіді в байтах. Типова XML-RPC-відповідь має наступний вигляд:

```
HTTP/1.1 200 OK
Date: Sat, 15 Jun 2019 10:01:05 GMT
Server: Apache/1.3.12 (Unix)
Connection: close
Content-Type: text/xml
Content-Length: 124
```

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
```

```

        <value><double>1.2</double></value>
    </param>
</params>
</methodResponse>

```

XML-RPC-XML-RPC-винятки генеруються в вигляді XML-документа XML-RPC-відповіді. Вони виникають при наявності помилок під час виконання методів. У цьому випадку елемент `<methodResponse>` містить елемент `<fault>` замість елемента `</params>`.

Порядок виконання роботи.

У даній лабораторній роботі ми будемо використовувати програмний продукт Apache XML-RPC (<http://archive.apache.org/dist/ws/xmlrpc/binaries/>), який дозволяє просто створювати XML-RPC Web-сервіси на мові програмування Java. Створимо Web-службу `JavaWebService`, яка надає одну Web-послугу `sum`, яка повертає клієнту суму двох чисел переданих при XML-RPC-запиті клієнта

Створіть новий проект, підключіть до нього бібліотеку `xmlrpc-2.0.1.jar`. Додатково підключіть бібліотеки `commons-codec.jar`, `commons-codec-javadoc.jar`, які можна завантажити з http://commons.apache.org/proper/commons-codec/download_codec.cgi

У лістингу наведено код клієнта Web-сервісу:

```

import java.util.*;
import org.apache.xmlrpc.*;

public class JavaClient
{
    public static void main (String [] args)
    {
        try {
            XmlRpcClient server = new
                XmlRpcClient("http://localhost:8080");
            Vector params = new Vector();
            int a=0; int b=0;

            try{
                a = Integer.parseInt(args[0]);
                b = Integer.parseInt(args[1]);
            }
        }
    }
}

```

```

        catch(Exception e){System.out.println("Error:
"+e);}
        params.addElement(a);
        params.addElement(b);
        Object result = server.execute("sample.sum",
params);
        int sum = ((Integer) result).intValue();
        System.out.println("The sum is: "+ sum);
    } catch (Exception exception) {
System.err.println("JavaClient: " + exception); }
    }
}

```

Java пакет `org.apache.xmlrpc` містить класи, необхідні для створення XML-RPC-клієнта і сервера. Пакет `java.util` потрібен для створення екземпляра класу `Vector`. Метод `server.execute()` надсилає запит серверу. У лістингу "sample" вказує на заголовок, який визначає Web-служба. Адреса Web-служби міститься в конструкторі класу `XmlRpcClient`.

В окремому проєкті створіть сервер Web-служби:

```

import org.apache.xmlrpc.*;
public class JavaWebService
{
    public int sum(int x, int y)
    {
        return x+y;
    }
    public static void main (String [] args)
    {
        try {
            System.out.println("Attempting to start XML-RPC
Web Service...");
            WebServer server = new WebServer(8080);
            server.addHandler("sample",
                                new JavaWebService());
            server.start();
            System.out.println("Started successfully.");
            System.out.println("Accepting requests");
        }
        catch (Exception exception) { System.err.println("Java
Web Service: " + exception); }
    } }

```

Підключіть до проекту ті самі бібліотеки, що підключали до клієнтського проекту. Для запуску Web-сервісу спочатку запустіть серверний проект, а потім клієнтський. В клієнтському застосунку передайте з консолі два числа, суму яких буде обчислювати сервер. Використовуйте утиліту `tcpmon` для перегляду вмісту повідомлень, якими обмінюються клієнт і сервер.

Зміст звіту.

1. Код клієнта та сервера Web-сервісу.
2. Код виконання індивідуального завдання, що видається викладачем.
3. Відповіді на контрольні питання

Контрольні питання.

1. У чому відмінність технології XML-RPC від SOAP?
2. Які типи даних може використовувати XML-RPC-протокол?
3. Який формат має XML-RPC-запит?
4. Який формат має XML-RPC-відповідь?
5. Які переваги та недоліки технології XML-RPC у порівнянні з програмування розподіленого додатку традиційними сокетамі?

4 ЛАБОРАТОРНА РОБОТА № 4. ОСНОВИ РОБОТИ З БІБЛІОТЕКОЮ REACT

Мета роботи: отримання базових навичок роботи з бібліотекою *React*, призначеної для побудови складних інтерфейсів, що масштабуються та реконфігуруються.

Теорія. Для розуміння ролі бібліотеки *React* згадаємо поширений архітектурний шаблон MVC (Model View Controller). Бібліотеці *React* тут відводиться лише View-складова. Названій бібліотеці дається таке визначення: *React* - це бібліотека інтерфейсу користувача, розроблена компанією *Facebook* для спрощення і автоматизації створення інтерактивних компонентів для користувача інтерфейсу, що зберігають стани (*stateful*), з позиції їх повторного використання. Компоненти – це ключові будівельні блоки названої бібліотеки. Бібліотека *React* використовується в робочому процесі компанії *Facebook*. Зокрема, сервіс *Instagram.com* створений на її основі. На відміну від типового підходу до побудови багатосторінкових веб-застосунків, в *React* використовується односторінкова модель SPA (Single-page Application). При цьому підході немає потреби здійснювати переходи між сторінками або перезавантажувати останні. Навпаки, різні уявлення завантажуються в єдину сторінку по частинах (блоками). Такий підхід суттєво підвищує чуйність і швидкість/зручність роботи програми. Суть підходу представлена на рис. 4.1.



Рисунок 4.1 – Модель односторінкового *ReactJS*-застосунка

При створенні односторінкового React-застосунку істотна частка трудовитрат буде спрямована на підтримку узгодженості даних, якими ми оперуємо, з елементами користувацького інтерфейсу застосунку. Типова структура відповідного рішення представлена на рис. 4.2.

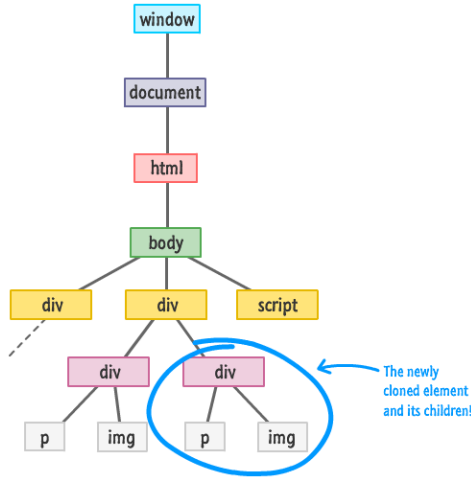


Рисунок 4.2 –Модель структури інтерфейсу React-застосунку

Поняття компонентів в *React*

Складові *React*-застосунків оформляються у вигляді модулів, які називаються *компонентами* – це багаторазово використовувані фрагменти *JavaScript*-коду, результати роботи яких подаються у вигляді *HTML*-елементів. Ці модулі концептуально досить подібні до функцій *JavaScript*. Для роботи з бібліотекою *React* необхідно встановити програмну платформу *Node.js*. Нас, зокрема, цікавить відповідний менеджер пакетів *npm* (*Node Package Manager*) *JavaScript*.

Інструментарій *React* можна охарактеризувати наступним чином:

- декларативний – код більш передбачуваний в розумінні і налагодженні;
- компонентно-орієнтований – компоненти управляють шаблонами даних.

Вже згадана бібліотека базується на концепції *Virtual DOM* (*Document Object Model*), при якій, на підставі зміни станів

компонентів, відображаються тільки необхідні піддерева (окремий випадок графа) дерева *DOM*. Такий підхід дозволяє підтримувати актуальні стани компонентів з найменшими обчислювальними витратами. В *Virtual DOM*, на відміну від *DOM*, маніпулювання проводяться не безпосередньо з об'єктною моделлю документа, а з її поданням до пам'яті – за рахунок цього і відбувається прискорення роботи. Під поточним (актуальним) станом при цьому слід розуміти актуальний знімок значень змінних станів (*JavaScript*-змінних). Чим більше змінних станів, тим більш повну *DOM*-модель суті (сутностей) нашої предметної області ми маємо в своєму розпорядженні, і тим, відповідно, більше маніпуляцій (обчислювальних дій) нам необхідно здійснити, щоб підтримувати актуальними стани компонентів.

Число станів залежить експоненціально від числа змінних станів і від множини (множин) їх допустимих значень. Для спрощення припустимо, що всі змінні станів приймають свої значення з одної безлічі D : $|D| = m$, тобто позначимо через m потужність множини значень змінних станів. Тоді, якщо позначити через n число змінних станів, отримаємо наступну оціночну формулу числа станів компонента $f(m, n)$:

$$f(m, n) = m^n. \quad (4.1)$$

Наприклад, якщо у нас є дві змінні станів, які беруть булеві значення, то, згідно з (1), отримаємо $2^2 = 4$ стани; якщо 3 змінних – $2^3 = 8$. Якщо, наприклад, множина значень не булева, а має вигляд, $\{0, 1, \dots, 9\}$, то при трьох змінних станів отримаємо вже 10^3 . Вид виразу (1) при цьому показує, що тимчасові витрати будуть зростати експоненціально.

Якщо під n розуміти число змінних станів всіх компонентів, то підтримка актуальності компонентів за рахунок використання концепції *Virtual DOM* буде пов'язана з дещо кращою оцінкою складності, ніж показано в (1). В цьому і полягає суть концепції *Virtual DOM* і отримуваний корисний ефект від її використання в рамках бібліотеки *React*.

Життєвий цикл компонентів React

React дозволяє створювати компоненти шляхом виклику методу `React.createClass()`, який передбачає метод рендерингу `render()`, а також обробники подій, що викликають зміни станів.

Життєвий цикл компонентів React можна представити наступними етапами:

- ініціалізація – створення компонента (функцій-обробників подій) і формування складу змінних станів;
- спрацьовування обробників подій, які обумовлюють зміну станів (переходи між станами);
- видалення (unmounting) компонента з DOM.

Установка бібліотеки React

В процесі установки буде використовуватися менеджер `npm`. Нижченаведені команди треба виконувати в консолі. Далі по тексту символ `>` позначає консоль, в якій вводиться наступна команда.

Спочатку треба встановити середовище виконання `Node.js`, що включає менеджер пакетів `npm` та систему контролю версій `Git`.

Перевірити версію встановленого середовища виконання `Node.js`:

```
> node -v
```

В результаті на консолі буде виведено повідомлення, подібне наступного: `V8.9.4`.

Перевірити версію менеджера пакетів:

```
> npm -v
```

В результаті отримаємо, наприклад, наступний рядок: `5.6.0`. Команда при цьому виконується досить повільно.

Перевірити версію системи контролю версій `Git` (в `Git Bash`):

```
> git --version
```

В результаті отримаємо, наприклад, наступний рядок: `git version 2.16.1.windows.1`.

Подальші дії ґрунтуються на двох альтернативних сценаріях розвитку подій:

- додавання підтримки React в новий проект;
- додавання підтримки React в існуючий проект.

У даній роботі розглянемо перший сценарій.

Створення нового проекту з підтримкою React

Для створення нового проекту з підтримкою React скористаємося такою командою:

```
> npx create-react-app my-app
```

Примітка: команда `npx`, а не `npm`, використовується в разі версії встановленого менеджера пакетів 5.2 і вище. Згаданий `npx` – це засіб, що спрощує використання інструментів консолі при роботі з менеджером пакетів `npm`.

У наведеному рядку `my-app` - назва створюваного проекту. Виконання команди займе кілька хвилин, поки всі необхідні пакети не будуть встановлені. Отримана в результаті ієрархія файлів і каталогів буде займати на накопичувачі сотні мегабайт. Інформація про статус процесу установки буде виводитися на консолі.

Тепер перейдемо в каталог створеного застосунку:

```
> cd my-app
```

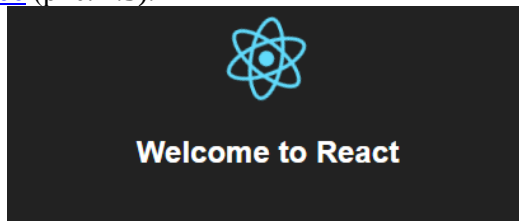
Після появи на консолі повідомлення про успіх створення ієрархії пакетів застосунку, в рамках директорії "`my-app`" будуть доступні наступні команди:

- `npm start` – запустити сервер розробки на виконання;
- `npm run build` – побудова застосунку;
- `npm test` – тестування застосунку;
- `npm run eject` – видалення інструментів розробки і копіювання залежностей, конфігураційних файлів і скриптів в директорію програми. Цей крок є незворотнім, після нього вже неможливо повернутися назад в середовище розробки.

Тепер можемо запустити сервер розробки на виконання:

```
> npm start
```

В результаті застосунок буде відкрито в браузері за адресою <http://localhost:3000> (рис. 4.3).



To get started, edit `src/App.js` and save to reload.

Рисунок 4.3 – Стартова сторінка заготовки створюваного застосунку

Короткий звіт по виконаним діям:

```
> npx create-react-app my-app
> cd my-app
> npm start
> http://localhost:3000 (в браузері)
```

З рис. 4.3 видно, що наступні наші дії будуться на роботі з вмістом каталогу `src`. Запущений в результаті виконання команди `npm start` сервер буде функціонувати до тих пір, поки не буде закрита консоль, в якій ця команда була виконана.

Для продовження проходження методичних вказівок необхідно відкрити нову консоль, не закриваючи попередню, оскільки для виконання наступних дій нам знадобиться функціонуючий сервер розробки.

Ієрархія файлів і папок створеної заготовки застосунку матиме вигляд, наведений на рис. 4.4.

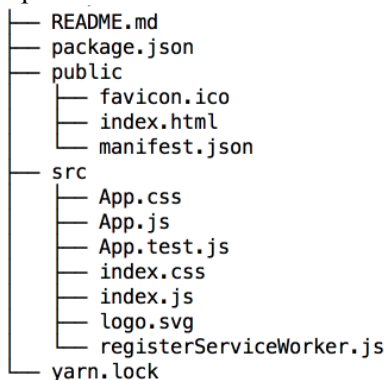


Рисунок 4.4 – Деревоподібна структура створюваного застосунку

У структурі дерева рис. 4.4 фігурують, зокрема, такі файли:

- `README.md` - файл, що містить інформаційне навантаження, призначене для користувача програми. У цей файл розробником вноситься опис проекту, з викладенням специфіки його використання;

- `package.json` – використовується менеджером пакетів `npm` для читання списку програмних пакетів, необхідних для проекту. Створюється як частина інфраструктури проекту; JSON (JavaScript Object Notation) – універсальний текстовий формат обміну даними, заснований на *JavaScript*;

- `src/api.js` – для виклику *backend API*;
- `src/index.js` – для ініціалізації програми – стартова сторінка;

Призначення файлу `package.json` полягає в наступному:

- зберігання списку залежностей - списку пакетів, від яких залежить створюваний проект;
- фіксація версій збірок;
- підвищення відтворюваності проекту.

Менеджер пакетів `npm` використовує файл `package.json` для читання списку програмних пакетів, необхідних для проекту. Файл `package.json` створюється як частина інфраструктури проекту. Каталог, що містить названий файл, інтерпретується як React-каталог.

Вміст файлу `package.json`, отриманого в нашому випадку при виконанні команди `npm create-react-app my-app`, наведено в лістингу 4.1.

Лістинг 4.1 - Вміст конфігураційного файлу `package.json` для випадку додавання підтримки ReactJS в новий проект

```
{
  "name": "my-app",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "react": "^16.2.0",
    "react-dom": "^16.2.0",
    "react-scripts": "1.1.1"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  }
}
```

Вміст лістингу 4.1 було згенеровано автоматично, в ньому фігурують, зокрема, такі параметри:

- `"name"` – назва пакета. Прогалини в назві заборонені;
- `"version"` – версія збірки - оформлюється відповідно до стандарту *Semantic Versioning (semver)* (табл. 1.1);

- "private" – при встановленні в значення true, забороняє публікацію приватних репозитаріїв;
- "dependencies" – список залежностей, в якому назву пакета відображається в його допустиму версію (діапазон версій);
- "scripts" – список допустимих команд, які виконуються в той чи інший час протягом життєвого циклу проекту.

Таблиця 4.1 - Правила зазначення версії збірки

№ з/п	Дії над кодом	Етап	Коментар	Приклад
1	Перший реліз	Початок	Почати з 1.0.0	1.0.0
2	Виправлення помилок і внесення незначних правок	Випуск заplatки	Інкрементувати третю цифру	1.0.1
3	Додавання нових можливостей (features), які, при цьому, не замінюють собою вже реалізовані можливості	Несуттєвий реліз (Minor Release)	Інкрементувати другу цифру	1.1.0
4	Внесення модифікацій, що усувають зворотну сумісність поточної і попередньої збірок	Суттєвий реліз (Major Release)	Інкрементувати першу цифру	2.0.0

Створення найпростішого React-застосунку

В даному випадку всю роботу будемо проводити в файлі index.html. При збереженні зміненого файлу відображення результату роботи програми в браузері (за адресою <http://localhost:3000>) автоматично змінюється.

Побудова каркасу застосунку

Візьмемо за основу HTML-код, наведений у лістингу 4.2, і будемо його поступово доповнювати.

Лістинг 4.2 - Каркас файлу *index.html*

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>First React Application!</title>
</head>
<body>
<script></script>
</body>
</html>

```

Використовуючи теги `<script>`, підключимо бібліотеку React в головній частині HTML-документа (лістинг 4.3, де виділені модифікації, що вносяться).

Лістинг 4.3 - Підключення бібліотеки *React*

```

<head>
  <meta charset="utf-8">
  <title>First React Application!</title>
  <script crossorigin
    src="https://unpkg.com/react@16/umd/react.developme
nt.js">
  </script>
  <script crossorigin
    src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js">
  </script>
</head>

```

Підключимо JavaScript-компілятор Babel. Це дозволить нам в рамках тега `<script>` писати більш компактний і наочний HTML-код, який потім буде перетворюватися компілятором в JavaScript-код. Для підключення компілятора заголовочну частину файлу необхідно доповнити ще одним `<script>` тегом (лістинг 4.4).

Лістинг 4.4 - Підключення компілятора *Babel*

```

<head>
...

```

```
<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
</head>
```

Згідно з підходом, наведеним на рис. 4.2, в межах "тіла" HTML-документа, обрамленого тегами `<body> </body>`, створимо блок, в рамках якого будемо виводити інформацію (лістинг 4.5).

Лістинг 4.5 - Формування блоку виведення інформації

```
<body>
<div id = "block1"></div>
<script></script>
</body>
```

У лістингу 4.5 ми створили блок і привласнили йому ідентифікатор – `block1`. Цей ідентифікатор необхідний для прив'язки до блоку, зокрема, для завдання стилів.

Рендерінг елемента

Модифікуємо елемент `<script>` – для відображення корисного навантаження в межах створеного блоку (лістинг 4.6).

Лістинг 4.6 - Відображення (рендерінг) елементів

```
<body>
<div id = "block1"></div>
<script type = "text/babel">
    var dst = document.querySelector("#block1");
    ReactDOM.render (
        <h1>React</h1>
        /* React.createElement (
            "h1",
            null,
            "React"
        ) */
        ,
        dst
    );
</script>
</body>
```

У лістингу 4.6 виконується наступне:

- вказується тип скрипта – з підтримкою JSX і відповідного компілятора Babel (**type = "text/babel"**);

- викликається метод **render**, який містить два аргументи: 1-й аргумент - "що відображаємо" – **<h1>React</h1>**; 2-й аргумент - "куди відображаємо" – **dst**. У нашому випадку **dst** – це допоміжна змінна, яка посилається на раніше створений блок з ідентифікатором **block1**. Для цього використовується метод **querySelector**;

- завдяки залученню виразних можливостей JSX і компілятора Babel, немає потреби використовувати закоментований блок коду. Замість цього достатньо лише HTML-подібного JSX-коду – **<h1>React</h1>**;

- якби ми не використали JSX-синтаксис, тоді від підключення бібліотеки babel.min.js варто було б позбутися (лістинг 4.4), при цьому використавши закоментований блок коду замість рядка **<h1> React </h1>** (лістинг 4.6). Крім того, відкриваючий тег **<script type = "text / babel">** слід при цьому використовувати без вказівки типу – просто **<script>**.

Завдання стилю блоку

Для завдання стилю блоку скористаємося тегам **<style>** **</style>** (лістинг 4.7). Відповідним кодом доповнимо заголовну частину HTML-документа.

Лістинг 4.7 - Параметри відображення

```
<head>
...
<style>
  #block1 {
    padding: 70px;
    background-color: #11aaff;
  }
  #block1 h1 {
    font-size: 100px;
    font-family: sans-serif;
    color: #0050aa;
  }
</style>
</head>
```

У лістингу 4.7 в якості ключового слова, що ідентифікує прив'язку стилів до елементів, виступає раніше введений ідентифікатор `#block1`. Підсумковий результат внесення всіх модифікацій наведено на рис. 4.5.



Рисунок 4.5 – Результат роботи застосунку

Подальша робота буде вестися з файлом `App.js`, розташованим в каталозі `my-app/src`. Цей файл містить клас `App`, що розширює клас `Component`. Подальші дії будемо проводити з цим файлом.

Створення елемента керування.

Виведемо елемент керування "Кнопка". Для цього вміст файлу `App.js` слід привести у відповідність з вмістом лістингу 4.8.

Лістинг 4.8 - Вміст файлу *App.js*

```
import React, { Component } from 'react';
import './App.css';

class App extends Component
{
  render()
  {
    return (<div className = 'button_container'>
      <button className = 'button'>
        push me
      </button>
    </div>)
  }
}
export default App;
```

Завдання стилів класів

Тепер займемося питаннями стилістичного оформлення нашого Frontend-застосунок. Для цього внесемо зміни в файл `App.css` відповідно до вмісту лістингу 4.9, попередньо видаливши з нього шаблонний вміст.

Лістинг 4.9 - Вміст файлу `App.css`

```
.button_container {  
margin-top: 200px;  
text-align: center;  
}  
  
.button {  
background-color: cyan;  
border: none;  
color: black;  
font-size: 24px;  
height: 80px;  
width: 120px;  
}
```

Поекспериментуйте з форматуванням стилів класів і поспостерігайте, як автоматично змінюється результат в браузері. У нашому випадку результат буде таким, як показано на рис. 4.6.

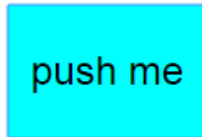


Рисунок 4.6 –Результат, що виводиться

Введення подій і обробників подій

Створимо подію і вкажемо її обробник в файлі `App.js`. Для цього код зазначеного файлу необхідно модифікувати відповідно до вмісту лістингу 4.10.

Примітка: спрацьовування обробників подій (функцій) переводить компонент в новий стан.

Лістинг 4.10 - Зазначення подій і функцій-обробників

```

<button      className      =      'button'      onClick      =
{this.handleClick}>
  push me
</button>

```

onClick – подія (натискання клавіші миші); handleClick – обробник події (функція); this. Вказує контекст (область дії) функції, тобто приналежність до контексту даного компонента. Контекст – це сукупність змінних і їх значень зараз; це по суті відображає поточний стан компонента.

Визначення елемента управління і зв'язування його з контекстом.

Для визначення елемента управління, в класі App (перед визначенням методу render ()) необхідно визначити відповідну однойменну функцію handleClick (лістинг 4.11). Для зв'язування елемента управління з контекстом компонента необхідно визначити відповідний конструктор. Код, яким необхідно модифікувати вміст файлу App.js (перед визначенням методу render ()), приведений в лістингу 4.11.

Лістинг 4.11 - Визначення елемента управління і зв'язування його з контекстом

```

handleClick() {
  console.log('done!');
}
constructor() {
  super();
  this.handleClick = this.handleClick.bind(this);
}

```

Перевірка отриманих результатів здійснюється в браузері в режимі розробника (Developer Mode). У разі використання браузера Opera в названий режим можна перейти шляхом натискання комбінації клавіш Ctrl + Shift + I.

У нашому випадку, при натисканні на елемент управління, в консолі браузера в режимі розробника будемо отримувати відповідні повідомлення "done!" (Лістинг 1.11, рис. 1.7).

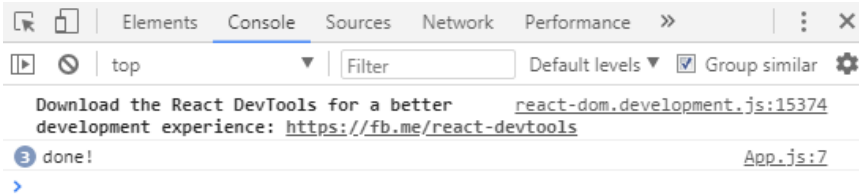


Рисунок 4.7 – Режим розробника

Створення HTTP-запитів.

Для спрощення роботи з HTTP-запитами скористаємося бібліотекою Axios. Для цього необхідно завершити роботу сервера розробки, а потім встановити бібліотеку:

```
> npm install axios
```

Потім слід внести модифікації в файл App.js, а саме імпортувати встановлену бібліотеку і скористатися функціями цієї бібліотеки для формування HTTP-запиту (лістинг 4.12).

Лістинг 4.12 - Формування HTTP-запиту

```
...
import axios from 'axios';
...
handleClick() {
  //console.log('done!');

  axios.get('https://api.github.com/users/maecapozzi')
    .then(response => console.log(response));
}
...
```

У лістингу 4.12 за вказаною в якості аргументу методу get () адресою розташовані дані, представлені в форматі JSON, фрагмент яких приведений в лістингу 4.13.

Лістинг 4.13 - Фрагмент даних, представлених в форматі JSON

```
{
  "login": "maecapozzi",
  "id": 11462208,
  ...
}
```

Якщо тепер зберегти файл App.js з внесеними змінами, і заново перевірити в режимі розробки функціонування обробника подій handleClick (), отриманий результат буде подібний до наведеного на рис. 4.8.

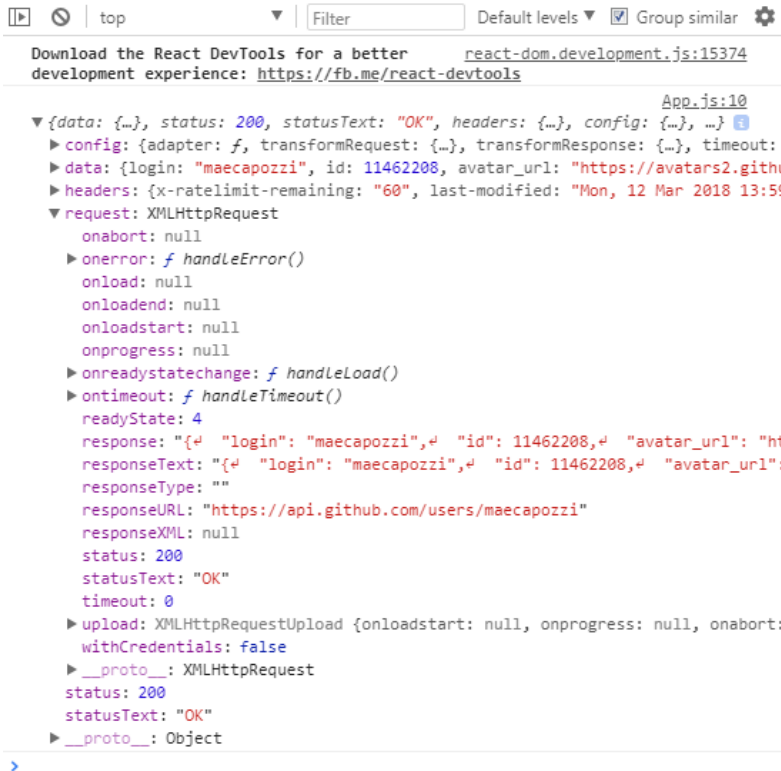


Рисунок 4.8 – Відправка HTTP-запиту і отримання HTTP-відповіді

З рис. 4.8 видно, що в якості відповіді на HTTP-запит ми отримуємо дані, що представлені в форматі JSON (лістинг 4.14).

Тепер сформуємо в конструкторі (файл App.js) множину змінних станів (атрибутів) з одного елемента (лістинг 4.14). Вона згодом може бути розширена.

Лістинг 4.14 - Завдання атрибутів стану

```
...
constructor() {
  super();
  this.state = {
    id: ''
  }
  this.handleClick = this.handleClick.bind(this);
}
...
```

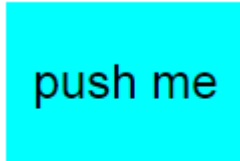
В якості змінної стану введено в обіг атрибут `id`. Тепер виведемо в вікні браузера лише цікаві для нас дані, а саме - представлені атрибутом `id`. Для цього модифікуємо обробник подій `handleClick()` і функцію відображення `render()` лістинг 4.15.

Лістинг 4.15 - Внесення заключних модифікацій

```
...
handleClick() {
  //console.log('done!');

  axios.get('https://api.github.com/users/maecapozzi')
    //.then(response => console.log(response));
    .then(response => this.setState({id:
      response.data.id}))
  }
  ...
  render() {
    return (
      <div className = 'button_container'>
        <button className = 'button' onClick =
{this.handleClick}>
          push me
        </button>
        <p>{this.state.id}</p>
      </div>
    )
  }
  ...
```

Тепер натискання на елемент управління "Кнопка" буде приводити до відображення під ним значення атрибута `id` (лістинг 4.15, рис. 4.9).



11462208

Рисунок 4.9 – Фінальний результат роботи застосунку

Стилізація інтерфейсу та інструментарій bootstrap

Для спрощення процесу оформлення інтерфейсу програми, а також для спрощення подальших модифікацій оформлення, скористаємося інструментарієм Bootstrap. Для цього встановимо відповідний модуль, попередньо зупинивши сервер розробки, а потім знову його запустивши:

```
> npm install --save react-bootstrap
```

Для використання можливостей бібліотеки необхідно виконати наступне:

- підключити бібліотеку `bootstrap.min.css` в заголовок файлу `index.html`, розташованого в директорії `public` проекту (лістинг 4.10);
- імпортувати необхідні компоненти встановленого модуля, вказавши відповідну `import`-директиву в файлі `App.js` (лістинг 4.11).

Лістинг 4.10 - Підключення таблиці стилів

```
<!doctype html>
<html lang="en">
<head>
  ...
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/
3.3.7/css/bootstrap.min.css">
</head>
<body>...
</body>
</html>
```

Листинг 4.11 – Імпортування засобів бібліотеки

```
import React, { Component } from "react";
import { Button, Glyphicon } from 'react-bootstrap';
...
```

Контроль версій проекту

Для контролю версій створеної заготовки проекту скористаємося інструментарієм Git. Виконаємо наступні кроки:

– перейшовши в каталог проекту, проведемо ініціалізацію репозитарія, виконавши наступну команду:

```
> git init
```

– позначимо всі файли проекту як підготовлені до додавання в репозитарій:

```
> git add
```

– додамо в репозитарій підготовлені файли, зафіксувавши зміни, асоційовані з ключовою фразою:

```
> git commit -m "First commit"
```

Усі наступні зміни необхідно супроводжувати новими унікальними ключовими фразами.

Щоб зробити власні нароби за проектом надбанням світу, можна скористатися платформою GitHub, на якій необхідно попередньо зареєструватися та створити віддалений репозитарій. Потім слід скопіювати локальний репозитарій у віддалений (далі фігурує як remote). Для цього можна скористатися такою послідовністю команд:

```
> git remote add origin "remote repository URL"
```

```
> git remote -v
```

```
> git push -f origin master
```

```
> git commit -a -m "2nd commit"
```

```
> git push -f origin master
```

```
> git log
```

Зміст звіту.

1. Код та результати роботи розробленого застосунка.
2. Виконання індивідуального завдання, виданого викладачем.
3. Відповіді на контрольні питання.

Контрольні питання.

1. Призначення бібліотеки React.
2. Роль бібліотеки React в архітектурному шаблоні MVC.
3. Дати визначення компонентів в React.
4. Життєвий цикл компонентів React.
5. Сформулювати визначення стану компонента React.
6. Призначення і коротка характеристика файлу package.json.
7. Призначення бібліотеки Bootstrap.
8. Доцільність використання системи контролю версій Git.
9. Призначення компілятора Babel.
10. Призначення і аргументи методу render.

5 ЛАБОРАТОРНА РОБОТА № 5. АРХІТЕКТУРНИЙ СТИЛЬ REST І ТЕХНОЛОГІЯ GraphQL

Мета роботи: ознайомитися з архітектурним стилем REST і технологією GraphQL. Отримати практичні навички створення GraphQL-запитів.

Теорія. Архітектурний стиль REST

Широко відома клієнт-серверна архітектура в даний час не прийнятна для побудови систем комерційної спрямованості на її основі, зважаючи на значну кількість потенційних користувачів таких систем (інтернет-магазини, портали, пошукові системи і т.д.). Основна причина – незадовільні показники продуктивності систем на основі цієї архітектури через надмірності обчислювального навантаження і потоків трафіку. Рішення полягає в використанні сучасного архітектурного стилю REST (Representational State Transfer). Цей термін був введений в 2000-му році Роем Філдіном одним з творців протоколу HTTP. Системи, побудовані на основі цього архітектурного стилю прийнято називати RESTful-системами.

Архітектурний стиль REST призначений для проектування розподілених систем. Під "розподіленістю" розуміється, перш за все, територіальна розподіленість компонентів системи, а також супутні цьому витрати (тимчасові, обчислювальні). Разом з тим, архітектурний стиль REST також оперує поняттями "клієнта" і "сервера". Відмінні риси REST:

– відсутність станів (statelessness) - ні клієнт не запитує стан сервера, ні сервер не цікавиться станом клієнта. На практиці це означає, що розуміння клієнтом і сервером повідомлень один від одного ніяк не залежить від кількості і складу раніше прийнятих ними повідомлень. Таким чином, в контексті розподілених систем, стан компонента системи визначається кількістю і складом прийнятих ним повідомлень: вхідне повідомлення спонукає виклики відповідних обробників повідомлень, що змінюють стан. Ця властивість досягається завдяки оперуванню поняттям "ресурс". Під "ресурсом" розуміється поняття предметної області Web – деякий об'єкт, який підлягає зберіганню або пересиланню деякого Web-сервісу. Взаємодія RESTful-систем здійснюється за допомогою операцій над ресурсами. З

цієї причини немає потреби в прив'язці до реалізації деякого інтерфейсу. У цьому ключі варто відзначити, що взаємодію компонентів розподіленої системи можна реалізувати двома принципово різними шляхами: шляхом обміну повідомленнями і за допомогою використання поділюваних змінних. У цьому ключі поняття «ресурс» співвідноситься з поділюваною змінною;

- відділення реалізацій клієнта і сервера – реалізація сервера може бути виконана незалежно від клієнтської частини, і навпаки. Це означає, що зміна коду клієнтської частини не впливає на функціонування сервера, і навпаки.

Перераховані вище особливості дозволяють характеризувати RESTful-додатки наступним чином: надійні, продуктивні, масштабовані. Разом з тим, REST-технологія характеризується рядом недоліків, критичних для мобільних додатків. Деякі з цих недоліків наведені нижче:

- при запиті складних об'єктів клієнтським додатком у серверного додатка виникає істотний (найчастіше надлишковий) трафік;

- при зміні вимог до додатка (системи) REST endpoints доповнюються ("обрастають") все новими даними. Для клієнтських застосунків, створених раніше, ці дані є надлишковими, породжують надлишковий трафік і зростання часу відгуку при взаємодії клієнтської і серверної частин застосунку;

- REST endpoints слабо типізовані і характеризуються недоліком метаданих. Це ускладнює процедуру розвитку і супроводу відповідних програмних продуктів.

Наведені недоліки технології REST подолані в рамках прогресивної технології GraphQL, мова про яку піде далі.

Мова запитів GraphQL

GraphQL – це мова запитів, створена компанією Facebook і призначена для отримання даних програми уніфікованим чином (шляхом). Цей засіб було представлено світовій громадськості в 2012 році в рамках технології Relay. GraphQL – це мова запитів (query language) для заданих API, а також серверне середовище виконання для здійснення запитів шляхом використання системи типів, визначеної для даних користувача.

Засоби GraphQL представляються у вигляді бібліотеки: вона не асоціюється з деякою базою даних або системою зберігання, а ґрунтується лише на наявних в розпорядженні коді і даних.

Базові принципи GraphQL:

- ієрархічність – GraphQL-запит – це ієрархічна множина полів; структура запиту відповідає структурі даних, яку ми запитуємо;
- продукто-центровані (мається на увазі цільовий програмний продукт) – характеристики GraphQL визначені вимогами front-end-інженерів з точки зору view-складової схеми MVC (Model-View-Controller);
- обернено сумісний;
- структурований – гранулярність на рівні полів;
- є протоколом прикладного рівня;
- строго типізований;
- інтроспективний – клієнтські та інші програмні компоненти можуть запитувати систему типів безпосередньо на основі синтаксису GraphQL.

Сервіс GraphQL створюється шляхом визначення типів, полів в рамках цих типів, а також функцій для кожного поля заданого типу. Іншими словами, GraphQL – це засіб запитування окремо взятих полів об'єктів.

Розглянемо приклад. Будемо оперувати сутностями «Запит» і «Користувач». Кожну озвучену сутність представимо відповідним типом і наповнимо ці конструкції полями (лістинг 5.1).

Лістинг 5.1 - Визначення типів

```
type Query {
  user: User
}

type User {
  id: ID
  name: String
}
```

У наведеному лістингу 5.1 тип Query містить єдине поле user, тип якого - User. Сміслові навантаження в даному випадку наступне: сервіс "Запит", представлений типом "Запит", призначений для

надання інформації про користувача, який в нашому випадку представлений полем `someone` типу `User`. Далі описуємо тип `User` полями `id` і `name`, які, в свою чергу, мають типи `ID` і `String` відповідно.

Характерна особливість GraphQL - структури запиту і JSON-відповіді (JavaScript Object Notation) збігаються (табл. 5.1).

Таблиця 5.1 - Структури відповіді і запиту GraphQL

Запит	Відповідь
<pre>query { person { name } }</pre>	<pre>{ "data": { "person": { "name": "John" } } }</pre>

У табл. 5.1 для поля `"name"` вертається рядок `"John"`.

Порівняння REST і GraphQL

Відмінні риси технологій REST і GraphQL наведені у вигляді порівняльної таблиці (табл. 5.2).

Таблиця 5.2 - Порівняння технологій REST і GraphQL

№ п/п	Параметри порівняння	Основоположні принципи	
		<i>REST</i>	<i>GraphQL</i>
1	Відмінні риси	Відсутність станів, структурований доступ до ресурсів.	Перехід від концепції множини точок доступу до ресурсів (API Endpoint) до концепції єдиної точки доступу.
2	Переваги	Підвищення продуктивності і спрощення архітектури розподілених систем, в порівнянні з технологією RPC (Remote Procedure	Забезпечує гнучкість і ефективність клієнт-серверної взаємодії, зокрема, за рахунок зниження надмірності інформації, що передається; немає потреби приводити API згідно зі зміною вимог

		Call).	на етапі проектування, що істотно сприяє зручності внесення змін при розробці програми.
3	Недоліки	Орієнтований на статичну природу клієнтської частини.	Порівняно складніше в розумінні, ніж REST.

Використання GraphQL замість REST дозволяє підвищити продуктивність (ефективність) клієнт-серверної взаємодії через те, що в першому випадку у відповідях на HTTP-запити не передається надлишкова інформація. Мова GraphQL може використовуватися з будь-якими backend-інструментами (frameworks) і мовами програмування.

Ключова ідея GraphQL – отримати всю цікаву для нас інформацію одним HTTP-запитом. Даним шляхом вирішуються дві характерні для REST проблеми: Overfetching - завантаження надлишкових даних, що веде до підвищення обсягу трафіку і зниження чутливості застосунку; Underfetching - отримання недостатньої кількості даних, що веде до потреби створення додаткових запитів до серверної частини, а це також загрожує вищенаведеними наслідками.

Таким чином, GraphQL – це наступний крок розвитку технологій реалізації розподілених систем (веб-застосунків), який дозволяє досягти більшої продуктивності і зручності реконфігурування таких рішень, ніж традиційна технологія REST. Розглянемо приклад. Нехай ми хочемо по користувачеві з ідентифікатором "123" отримати інформацію наступного характеру: ім'я користувача, заголовки всіх його публікацій (постів), а також імена останніх трьох його підписників. Тоді HTTP-POST GraphQL-запит і відповідна відповідь можуть мати вигляд, наведений в табл. 5.3.

З табл. 5.3 видно, що структури GraphQL-запиту і JSON-відповіді збігаються.

Таблиця 5.3 - Приклади запиту і відповіді GraphQL

Запит	Відповідь
<pre> query { User(id: "123") { name posts { title } } followers(last: 3) { name } } </pre>	<pre> { "data": { "User": { "name": "John", "posts": [{title: "First"}, {title: "Second"}, ...], }, "followers": [{name: "Will"}, {name: "Smith"}, {name: "Rob"}] } } </pre>

Типи і схеми в GraphQL

Робота з GraphQL полягає в оперуванні поняттями "об'єктів" і "полів". Як приклад розглянемо код, наведений в лівій колонці табл. 5.1. Зовнішні фігурні дужки представляють кореневий (root) об'єкт. Ми запитуємо поле `person` цього об'єкта. На цей запит нам повертається однойменний об'єкт `person`. В цьому об'єкті нас цікавить, в свою чергу, вже поле `name`, і т.д. Для подібних маніпуляцій необхідно знати точний опис даних, які ми запитуємо, а саме склад запитуваних об'єктів і поля цих об'єктів.

Кожен сервіс GraphQL задає множину типів, яка повністю характеризує множину різноманітних даних, які ми запитуємо. Таку множину типів називають схемою, за якою перевіряється валідність (придатність до цільового використання) GraphQL-запитів. Таким чином, схема – це засіб валідації запитів.

Основними компонентами GraphQL-схеми є типи об'єктів, а також поля цих об'єктів, які нас цікавлять. Схема вже фігурувала в лістингу 5.1. Більш комплексний приклад схеми приведений в лістингу 5.2.

Лістинг 5.2 - Приклад GraphQL-схеми

```

type Statement {
  id: ID!
  title: String!
  description: [Bodies]!
}

```

У лістингу 5.1 Statement – тип об'єктів GraphQL, які ми можемо запитувати, а також поля об'єктів цього типу: id, title і description. Наведені поля також характеризуються типами: ID, String і [Bodies] відповідно. Додатковий символ ! служить ознакою, що відповідне поле є non-nullable, тобто GraphQL-сервіс зобов'язується завжди надавати значення запитуваного поля. У разі використання [] ми маємо на увазі, що будемо запитувати масив об'єктів Bodies.

Поля також можуть мати аргументи, які, в свою чергу, наводяться в дужках. Приклад поля з аргументом приведені в лістингу 5.3.

Лістинг 5.3 - Приклад схеми з полем з аргументом

```

type Statement {
  ...
  language(encoding: encType = UTF): String
  ...
}

```

У лістингу 5.3 маємо поле з одним аргументом. Можна відзначити, що формат завдання аргументу поля подібний формату завдання поля. У нашому випадку для поля language типу String маємо єдиний аргумент encoding типу encType. Більш того, аргумент ініціалізований першим значенням за замовчуванням UTF. Це означає, що в разі, якщо аргумент не передається, його значення буде встановлено в UTF. Більш того, бачимо, що символ ! не фігурує, а це значить, що, в разі чого, в якості значення запитуваного поля допускається повернення null.

Таким чином, робота з GraphQL зводиться до створення відповідних схем – специфікування типів об'єктів предметної області, яка нас цікавить, та їх полів. У розглянутих прикладах (лістинги 5.1 і 5.2) фігурували типи об'єктів і полів. Приділимо їм більше уваги.

Схема GraphQL підтримує такі базові типи: String, Int, Float, Boolean, ID. Варто відзначити, що перші чотири наведених типи характерні для більшості сучасних мов програмування. Останній же тип ID дає відсилання на бази даних, що визначається специфікою функціонування розподілених веб-додатків комерційної спрямованості.

Swagger /OpenAPI як аналог схеми GraphQL

Специфікація Swagger / OpenAPI - це формат опису API для REST. Розглянемо складові цього поняття окремо.

Специфікація OpenAPI дозволяє задавати (регламентувати), зокрема, наступне:

- доступні endpoints і операції над ними;
- вхідні та вихідні параметри для кожної операції;
- методи аутентифікації;
- контактну інформацію, ліцензування, умови використання, ін.

Swagger, в свою чергу, – це набір відкрито поширюваних програмних засобів, створених навколо специфікації OpenAPI, покликаних спростити (автоматизувати) процеси проектування, побудови, документування та використання API для REST. Цей набір засобів включає наступні складові:

- Swagger Editor – редактор на основі браузера, призначений для створення специфікацій OpenAPI;
- Swagger UI – засіб подання OpenAPI-специфікації у вигляді інтерактивної документації API;
- Swagger Codegen – засіб генерування серверних заглушок (стаб) і клієнтських бібліотек на основі специфікації OpenAPI.

Характерна особливість OpenAPI – можливість REST API описувати свою власну структуру.

Порядок Виконання роботи.

Створення HTTP-сервера.

Передумови для виконання лабораторної роботи аналогічні таким для попередньої роботи: необхідна наявність встановленої програмної платформи Node.JS.

Скористаємося JavaScript-орієнтованою реалізацією мови GraphQL, а саме - GraphQL.js.

Як було зазначено раніше, GraphQL – це протокол прикладного рівня. З цієї причини виконання роботи почнемо зі створення HTTP-

сервера. Потім перейдемо до визначення GraphQL-схем і формування відповідних запитів.

Створення нового проекту і підключення модуля Express

Використання названого модуля покликане спростити процедуру створення HTTP-сервера. Для цього в новому каталозі, призначеному для проекту, створимо конфігураційний файл `package.json` і підключимо пакет Express:

```
> npm init
> npm install express --save
```

Примітка: параметр `save` зазначає, що інформацію про встановлений модуль слід додати в список залежностей проекту. В іншому випадку вказується параметр `no-save`.

Успішність виконання вищенаведених дій перевіримо, переглянувши вміст файлу `package.json`, отриманого в результаті виконання команди `npm init`: значенням ключа `"dependencies"` буде об'єкт, що містить єдину пару "ключ-значення": `"express": "^4.16.3"`. Це означає, що був встановлений модуль `"express"` версії 4.16.3 і доданий в список залежностей

Підключення компілятора Babel

Babel - це JavaScript-компілятор, що дозволяє писати код згідно актуальної версії стандарту ECMAScript. У нашому випадку це дозволить використовувати синтаксис специфікації ES2015 (ECMAScript 2015). Для підключення відповідного модуля до проекту виконаємо аналогічні дії:

```
> npm install babel-register babel-preset-env --save-dev
```

В результаті в файлі `package.json` виникне ключ `"devDependencies"`, об'єкт-значення якого буде містити пари `"babel-register": "^6.26.0"` і `"babel-preset-env": "^1.7.0"`. Поява нового ключа обумовлено зазначенням при виконанні команди в якості варіанту не `-save`, а `--save-dev`.

Примітки:

- зазначення параметра `--save-dev` замість `--save` сигналізує, що відповідний модуль необхідно додати в список залежностей для розробки, а не для "продакшн";

- установка вищенаведених модулів `babel-register` і `babel-preset-env` покликане привнести підтримку директив `import` стандарту ES6 в Node.JS.

Створення файлу запуску `index.js` - точки входу - для зручності запуску і маніпулювання серверною частиною (лістинг 5.3).

Лістинг 5.3 - Файл запуску HTTP-сервера

```
// підключаємо компілятор Babel
require('babel-register') ({
  presets: [ 'env' ]
})
// підключаємо серверну частину
module.exports = require('./server1.js')
```

Лістинг 5.3 розглядаємо як точку входу, через яку будемо маніпулювати серверною частиною. Бачимо також, що в якості реалізації сервера підключається файл `server1.js`, який ще потрібно створити.

Створення HTTP-сервера засобами інструментарію `express`.

Код серверної частини, що базується на виклику HTTP-методу GET, збережемо у файлі `server1.js` (лістинг 5.4).

Лістинг 5.4 - HTTP-сервер на базі інструментарію `express`.

```
import express from 'express'
const app = express()
app.get('/no-graphql', function (req, res) {
  res.send('get')
})
app.listen(4001, () => console.log('Server 1 is up
and running on port 4001!'))
```

У лістингу 5.4 ми створили HTTP-сервер, що очікує підключення до порту 4001.

Запуск HTTP-сервера.

Для запуску сервера скористаємося раніше створеною точкою входу (лістинг 5.3):

```
> node index.js
```

В результаті в консолі буде виведено наступне повідомлення:
'Server 1 is up and running on port 4001!'

Примітка: якщо б ми не підключили компілятор Babel в файлі index.js, то змушені б були використовувати застарілі виразні можливості, зокрема, використовувати конструкцію `const express = require('express')` замість конструкції `import express from 'express'`.

Скористаємося перевагами (з точки зору гнучкості реконфігурування) з позиції використання зв'язки файлів index.js і server1.js - замінімо сервер, залишивши точку запуску колишньою: створимо альтернативний HTTP-сервер, заснований на використанні методу POST (лістинг 5.5). Новий сервер реалізуємо в файлі server2.js (лістинг 5.5).

Лістинг 5.5 - POST-сервер з альтернативним портом доступу 4002

```
import express from 'express';
let app = express();
let port = 4002;
app.post('/no-graphql', (req, res) => {
  res.send('POST-response');
});
let server = app.listen(port, () => {
  let host = server.address().address;
  let port = server.address().port;
  console.log('No-graphql is listening at
http://%s:%s', host, port);
});
```

Нам залишається лише модифікувати вміст файлу index.js, замінивши рядок, що передається в якості аргументу методу `require` з server1.js на server2.js.

У нашому випадку / no-graphql – це endpoint, з яким будемо взаємодіяти.

Протестуємо новий сервер, відправивши на нього POST-запит. Для цього, при запусненому (що очікує) сервері, в новій консолі скористаємося можливостями утиліти curl, що дозволяє відправляти, зокрема, POST-запити з командного рядка:

```
> curl -X POST http://localhost:4002/no-graphql
```

В результаті отримаємо відповідь від сервера такого вигляду (лістинг 5.5):

```
POST-response.
```

Примітки:

– на даному етапі ми вже отримали функціонуючі HTTP-сервера, однак, все ще не регламентували питання взаємодії з ними, тобто які дані маємо намір передавати і в якому форматі. Для усунення цього упущення введемо підтримку GraphQL;

– також варто відзначити, що в наведених вище лістингах при оголошеннях змінних, замість звичного `var`, фігурували літерали `let` і `const`. Їх підтримка була привнесена стандартом ES-2015.

Додавання підтримки GraphQL і робота зі схемою

Для регламентування формату взаємодії з сервером підключимо модуль GraphQL і згодом визначимо відповідну схему. Для підключення модулів GraphQL виконаємо звичні дії:

```
> npm install graphql --save
```

```
> npm install body-parser --save
```

Модуль `graphql` дозволяє визначити схему і виконувати запити відповідно до цієї схеми. Модуль `body-parser` потрібен для обробки запитів. Схему представимо в окремому файлі - `schema.js` (лістинг 5.6).

Лістинг 5.6 - Визначення схеми

```
import {
  GraphQLObjectType,
  GraphQLSchema,
  GraphQLInt
} from 'graphql/type';
let foo = 1;
let schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'RootType',
    fields: {
      foo: {
        type: GraphQLInt,
        resolve: function() {
          return foo;
        }
      }
    }
  })
});
export default schema;
```

У лістингу 5.6 ми створили екземпляр `schema` типу `GraphQLSchema`. Кореневий об'єкт, який представляє запит типу `GraphQLObjectType`, повертає об'єкт типу `RootType`. Цей об'єкт, в свою чергу, містить єдине поле `foo`, що має тип `GraphQLInt`.

Підключимо схему, наведену в лістингу 5.6 до POST-сервера, представленого файлом `server2.js` (лістинг 5.5). Для цього модифікуємо (доповнимо) вміст лістингу 5.5 кодом, наведеним в лістингу 5.7.

Лістинг 5.7 - Підключення схеми до POST-сервера `server2.js`

```
...
import schema from './schema';
import {graphql} from 'graphql';
import bodyParser from 'body-parser';
...
//використаємо підключений модуль 'body-parser'
app.use(bodyParser.text({type:
'application/graphql'}));
//замінемо попередній виклик методу наступним
app.post('/graphql', (req, res) => {
  graphql(schema, req.body).then(
    (result) => {
      res.send(JSON.stringify(result,null,2));
    }
  );
});
...
```

Запустимо модифікований сервер `server2.js`:

```
> node index.js
```

Виконаємо POST-запит до HTTP-сервера:

```
> curl -X POST -H "Content-Type:application/graphql" -d "query
RootType { foo }" http://localhost:4002/graphql
```

В результаті отримаємо JSON-відповідь згідно створеної схеми (лістинг 5.6):

```
{
  "data": {
    "foo": 1
  }
}
```

Можна також використовувати і скорочений формат виклику, опустивши вказівку кореневого типу:

```
> curl -X POST -H "Content-Type:application/graphql" -d "{ foo }"
http://localhost:4002/graphql
```

Визначимо схему, і сформуємо запит відповідно до цієї схеми (лістинг 58). Для цього здійснимо наступні кроки:

- визначимо схему;
- визначимо функцію отримання значень полів схеми;
- виконаємо GraphQL-запит.

Відповідний файл назвемо requester.js.

Лістинг 5.8 - Визначення схеми і формування запиту

```
import { graphql, buildSchema } from 'graphql';
// 1: Побудова схеми з використанням GraphQL
let schema = buildSchema(`
  type Query {
    hello: String
  }
`);
// 2: Визначення функції отримання значення поля
let root = {
  hello: () => {
    return 'Hello!';
  },
};
// 3: Формування GraphQL-запиту
graphql(schema, '{ hello }', root).then((response)
=> {
  console.log(response);
});
```

Примітка до лістингу 5.8: слід відрізняти символ ```, який використовується при визначенні схеми, від символу `'`, використовуваного при визначенні строкових констант.

Модифікувати файл index.js під виклик створеного скрипта requester.js. Виконати запит відповідно до визначеної схеми (лістинг 5.8):

```
> node index.js
```

В результаті отримаємо консольне виведення такого вигляду:

```
{ data: { hello: 'Hello!' } }
```

Здійснити спробу запуску створеного скрипта безпосередньо. Пояснити отримані при цьому помилки:

```
> node requester.js
```

Реалізуємо отримані напрацювання (requester.js) в рамках HTTP-сервера. Для цього перейменуємо файл requester.js в server3.js. При цьому при виконанні функції use() бібліотеки express застосуємо підхід, що відрізняється від раніше розглянутого. Для цього також підключимо модуль express-graphql. Функціонал відповідної бібліотеки дозволяє монтувати GraphQL API на HTTP endpoint:

```
> npm install express-graphql --save
```

2.2.3.6 Модифікуємо вміст файлу server3.js (лістинг 5.8) згідно вмісту лістингу 5.9.

Лістинг 5.9 - HTTP-сервер на основі попередніх напрацювань

```
import { graphql, buildSchema } from 'graphql';
import express from 'express';
import graphqlHTTP from 'express-graphql';
let schema = buildSchema(`
  type Query {
    hello: String
  }
`);
let root = {
  hello: () => {
    return 'Hello 3!';
  },
};
let app = express();
app.use(
  '/graphql',
  graphqlHTTP({
    schema: schema,
    rootValue: root,
    graphiql: true,
  })
);
app.listen(4003);
console.log('http://localhost:4003/graphql');
```

Перевірка працездатності сервера server3.js шляхом відправки GET-запиту:

```
> curl -X GET -H "Content-Type:application/graphql" -d "{ hello }"
http://localhost:4003/graphql
```

В результаті буде виведено повідомлення наступного виду:

```
{"data":{"hello":"Hello 3!"}}
```

Реалізація клієнтської частини.

Відправлення запитів з панелі розробника браузера. Однією з варіацій перевірки працездатності серверної складової є використання утиліти curl, що неодноразово виконувалося раніше. Альтернативний підхід – використання панелі розробника браузера.

Для цього необхідно виконати наступне:

- звернутися до HTTP endpoint в адресному рядку браузера. У нашому випадку це <http://localhost:4003/graphql>. В результаті буде виведено повідомлення наступного виду: Cannot GET /. Для отримання відповіді від сервера сформуємо запит вручну згідно з нижченаведеними рекомендаціями;

- перебуваючи на зазначеному endpoint, перейти до панелі розробника. Наприклад, в разі використання браузера Opera, це можна здійснити комбінацією клавіш Ctrl+Shift+I;

- викликати функцію fetch () (лістинг 5.10).

Лістинг 5.10 - Використання функції fetch ()

```
fetch('/graphql', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
  body: JSON.stringify({query: "{ hello }"})
})
  .then(r => r.json())
  .then(data => console.log('data returned:',
data));
```

Виконання скрипта, наведеного в лістингу 5.10, в консолі панелі розробника виведе результат, отриманий при виконанні попереднього завдання і використанні утиліти curl.

Замінити значення 'POST' ключа method при виконанні функції fetch () на 'GET'. Прокоментувати при цьому помилки, які виникають.

Зміст звіту.

1. Код та результати роботи розробленого застосунка.
2. Виконання індивідуального завдання, виданого викладачем
3. Відповіді на контрольні питання.

Контрольні питання

1. Призначення і особливості архітектурного стилю REST.
2. Відмінні риси REST.
3. Охарактеризувати поняття "стан" компонента системи.
4. Дати визначення GraphQL.
5. Складові GraphQL-сервісу.
6. Охарактеризувати переваги використання GraphQL в порівнянні з REST.
7. Призначення схеми GraphQL.
8. Базові типи GraphQL.
9. Призначення утиліти curl.

ЛІТЕРАТУРА

1. Роббинс Дженнифер. HTML5, CSS3 и JavaScript. Искрывающее руководство / Дженнифер Роббинс. – 4-е издание. – Эксмо, 2014. – 516 с.
2. Фримен, А. Angular для профессионалов / А. Фримен. – СПб. : Питер, 2018. – 800 с.
3. Бранденбау Д. JavaScript: сборник рецептов / Д. Бранденбау - СПб.: Питер, - 2000, - 416 с.
4. Хабибулин И.Ш. Создание Web-служб средствами Java. – СПб.: БХВ-Петербург, 2003. – 400 с.
5. Хабибулин И.Ш. Создание распределенных приложений на Java 2. – СПб.: БХВ-Петербург, 2002. – 704 с.
6. Casteleyn S., Daniel F., Dolog P., Matera M. Engineering Web Applications. – Berlin: Springer-Verlag, 2009. – 363p.
7. Learning React.js: Getting Started and Concepts : <https://scotch.io/tutorials/learning-react-getting-started-and-concepts>.
8. Node.js: <https://nodejs.org/en/>
9. React js, учебник, документация: <http://thewebland.net/react-js/>.
10. A re-introduction to JavaScript (JS tutorial): https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript
11. Understanding REST: <https://spring.io/understanding/REST>
12. What is REST?: <https://www.codecademy.com/articles/what-is-rest>.
13. Introducing Relay and GraphQL: <https://reactjs.org/blog/2015/02/20/introducing-relay-and-graphql.html>