React Quickly

React Quickly SECOND EDITION MORTEN BARKLUND AZAT MARDAN ©2023 by Manning Publications Co

Розділ 1

Зустрічаємо React

Цей розділ охоплює

- Розуміння того, що таке React
- Розв’язування проблем за допомогою React
- Встановлення React у ваші веб-додатки
- Написання вашої першої веб-програми React: Hello World

React — це новаторський інструмент, про який веб-розробники навіть не підозрюють, що він їм потрібен, але не можуть відмовитися від нього, спробувавши. Це точно стосується двох авторів цієї книги, а також багатьох інших веб-розробників-ентузіастів. React надзвичайно популярний — і не дарма.

Якщо ви займалися веб-розробкою на початку 2000-х, все, що вам потрібно, це трохи HTML і мова сервера, наприклад Perl або PHP. Ах, старі добрі часи розміщення вікон alert() лише для налагодження коду інтерфейсу. Відтоді Інтернет значно розвинувся, і складність створення веб-сайтів різко зросла. Веб-сайти стали веб-додатками зі складними інтерфейсами користувача (UI), бізнес-логікою та рівнями даних, які потребують змін і оновлень з часом — і часто в режимі реального часу.

Багато бібліотек шаблонів JavaScript були написані, щоб спробувати вирішити проблеми складних інтерфейсів користувача. Але вони все ще вимагають від розробників дотримуватись старого розподілу завдань, який розділяє стиль (каскадні таблиці стилів [CSS]), дані та структуру (HTML) і динамічні взаємодії (JavaScript), і вони не відповідають сучасним потребам. (пам'ятаєте DHTML?).

Навпаки, React пропонує новий підхід, який за правильного використання оптимізує інтерфейс веб-розробки. React — це потужна

бібліотека інтерфейсу користувача, яка пропонує альтернативу, яку прийняли багато великих компаній, як-от Facebook, Netflix і Airbnb, і розглядають її як шлях вперед. Замість того, щоб визначати одноразовий шаблон для ваших інтерфейсів користувача, React дозволяє вам створювати повторно використовувані компоненти інтерфейсу користувача в JavaScript, які ви можете використовувати знову і знову на своїх сайтах.

Вам потрібен контроль captcha або вибір дати? Використовуйте React, щоб визначити компонент <Captcha /> або <DatePicker />, який ви можете додати до своєї форми: простий вставний компонент із усіма функціями та логікою для зв'язку з серверною частиною. Вам потрібне вікно автозаповнення, яке асинхронно запитує базу даних, коли користувач вводить чотири або більше літер? Визначте компонент <Autocomplete charNum="4"/>, щоб зробити цей асинхронний запит. Ви можете вибрати, чи має він інтерфейс текстового поля, чи не має інтерфейсу і замість цього використовує інший настроюваний елемент форми, наприклад, <Autocomplete textbox="..." />.

Цей підхід не новий. Створення створюваних інтерфейсів користувача існує вже давно, але React є першим, хто використовує чистий JavaScript без шаблонів, щоб зробити це можливим. І цей підхід виявився легшим для підтримки, повторного використання та розширення.

React — чудова бібліотека для створення інтерфейсів користувача, і вона має бути частиною вашого веб-інструменту для зовнішнього інтерфейсу, але це не повне рішення для всіх інтерфейсних веб-розробок. Частину цього розділу ми приділимо розгляду переваг і недоліків використання React у ваших програмах і тому, як React може вписатися у ваш існуючий стек веб-розробки.

У цій книзі ми розглянемо основи React і не більше, надаючи читачам міцну основу в основних концепціях і принципах бібліотеки React, не заглиблюючись у будь-які зовнішні або складні теми. Зосередившись виключно на React, читачі отримають повне розуміння його можливостей і будуть добре підготовлені для застосування своїх знань у широкому діапазоні проектів веб-розробки.

ПРИМІТКА. Вихідний код для прикладу в цій главі доступний за адресою https://rq2e.com/ch01.

1.1 Переваги використання React

Кожна нова бібліотека чи фреймворк у певному відношенні претендує на те, щоб бути кращими за своїх попередників. На початку у нас був jQuery, і він був неабияк кращим для написання кросбраузерного коду на рідному JavaScript. Якщо ви пам'ятаєте JavaScript зі старих часів, один запит до сервера займав би багато рядків коду, оскільки він мав враховувати Internet Explorer і подібні до WebKit браузери. З jQuery для цього знадобився лише один рядок: $.ajax(), наприклад. Колись jQuery певною мірою був відомий як фреймворк, але тепер ні! Тепер фреймворк – це щось більше та потужніше.

Так само з Backbone, а потім Angular, кожне нове покоління фреймворків JavaScript привнесло щось нове. React не унікальний у цьому. Новим є те, що React кидає виклик деяким основним концепціям, які використовуються в більшості популярних інтерфейсних фреймворків, наприклад, ідеї про необхідність мати шаблони.

Наступний список висвітлює деякі переваги React порівняно з іншими бібліотеками та фреймворками, які існували на момент появи React:

- Простіші веб-програми — React використовує компонентну архітектуру (CBA) із чистим JavaScript; декларативний стиль; потужні, зручні для розробників абстракції Document Object Model (DOM) (і не лише DOM, але й iOS, Android тощо).

- Швидкі інтерфейси користувача — React забезпечує надзвичайну продуктивність завдяки своїй віртуальній DOM і інтелектуальному алгоритму узгодження, який, як побічну перевагу, дає змогу виконувати тестування без розкручування (запуску) безголового браузера.

- Менше коду для написання — чудова спільнота React і обширна екосистема компонентів надають розробникам різноманітні бібліотеки та компоненти. Це важливо, коли ви обмірковуєте, яку структуру використовувати для розробки.

Багато функцій зробили роботу з React простішою, ніж з більшістю інших інтерфейсних фреймворків, доступних у зародковому стані. Однак після появи React з'явилося багато нових фреймворків. Частково завдяки популярності React деякі з цих нових фреймворків були розроблені з

подібними перевагами чи ідеями, кожну дещо змінено різними способами. Деякі інші фреймворки можуть просто бути натхненні загальною ідеєю, але працювати зовсім по-іншому, тоді як інші дуже схожі на React, лише з меншим набором функцій, які інколи вимагають від вас писати більше коду, але в інших випадках кодова база програми набагато менша. .

Ми розглянемо переваги, які роблять React популярним. Це основні переваги React, і вони зробили фреймворк унікальним на момент його появи, хоча інші сучасні фреймворки мають подібні переваги сьогодні. Давайте почнемо розкривати ці переваги одну за одною, починаючи з того, наскільки React надзвичайно простий у використанні.

1.1.1 Простота

Концепція простоти в інформатиці високо цінується розробниками та користувачами, але вона не дорівнює простоті використання. Щось просте може бути важко реалізувати, але в кінцевому підсумку воно буде більш елегантним і ефективним. І часто легка річ стає складною. Простота тісно пов'язана з принципом KISS (keep it simple, stupid). Суть полягає в тому, що простіші системи працюють краще.

Підхід React дозволяє створювати простіші рішення завдяки значно кращому досвіду веб-розробки для розробників програмного забезпечення. Коли ми почали працювати з React, це був значний зсув у позитивному напрямку, який нагадав нам про перехід від використання простого JavaScript без фреймворку до jQuery.

У React ця простота досягається за допомогою таких функцій:

 Декларативний стиль замість імперативного — React використовує декларативний стиль замість імперативного, автоматично оновлюючи представлення.

 СВА з використанням чистого JavaScript — React не використовує доменно-спеціальні мови (DSL) для своїх компонентів, лише чистий JavaScript. І немає розділення під час роботи над тією самою функціональністю.

 Потужні абстракції — React має спрощений спосіб взаємодії з DOM, що дозволяє вам нормалізувати обробку подій та інші інтерфейси, які однаково працюють у різних браузерах

Розглянемо ці функції одну за одною.

ДЕКЛАРАТИВНИЙ СТИЛЬ НАД НАКАЗОВИЙ

Декларативний стиль означає, що розробники пишуть, як це має бути, а не що робити, крок за кроком (імперативний). Але чому декларативний стиль є кращим вибором? Перевага полягає в тому, що декларативний стиль зменшує складність і полегшує читання та розуміння коду.

Різниця між імперативним і декларативним стилями кодування може швидко стати певною мірою академічною. Якщо довести до крайності, декларативне програмування може стати дуже складним для читання, якщо ви добре не розумієте деякі досить складні поняття, такі як монади та функтори. Ось кілька різних способів описати різницю між двома стилями:

- ***Інструкції проти виразів***. Програмування в імперативному стилі часто працює з незалежними операторами, які індивідуально передають стан програми, тоді як декларативне програмування використовує вирази, які будуються одне на одному, щоб просувати потік логіки.

- ***Використання зарезервованого с***лова — програмування в імперативному стилі часто використовує багато зарезервованих слів, таких як for, while, switch, if та else, тоді як програмування в декларативному стилі використовує методи масиву, функції зі стрілками, доступ до об'єктів, логічні вирази та тернарні оператори для досягнення однакових результатів.

- ***Композиція функцій*** — програмування в імперативному стилі часто використовує незалежні виклики функцій і виклики методів, у той час як програмування в декларативному стилі використовує композицію функцій для побудови попереднього виразу та створення невеликих узагальнених фрагментів логіки, які, складені, досягають бажаного результату.

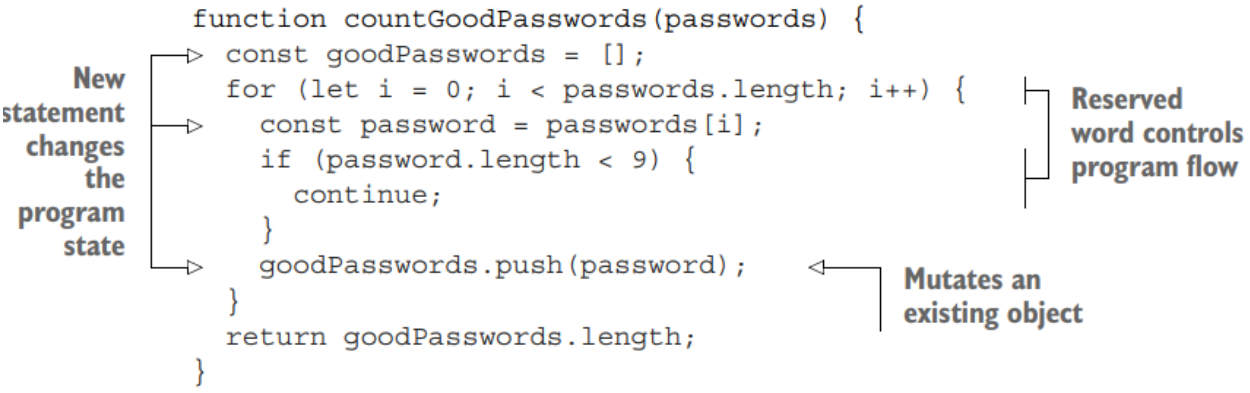
- ***Змінність*** — програмування в імперативному стилі часто використовує змінні об'єкти та маніпулює існуючими структурами, тоді як програмування в декларативному стилі використовує незмінні дані та створює нові структури зі старих, а не редагує існуючі.

Давайте створимо простий приклад, щоб проілюструвати ці різні моменти. Метою цього завдання є створення функції countGoodPasswords, яка, маючи список паролів, повертатиме, скільки паролів є правильними. Тут ми

визначимо хороший пароль як будь-який пароль довжиною щонайменше дев'ять символів.

Це чудове просте завдання, яке можна вирішити на будь-якій мові програмування багатьма способами. Деякі мови програмування за своєю суттю роблять один стиль більш природним для досягнення, але JavaScript є дещо особливим, оскільки він є членом обох світів. Цю задачу можна вирішити як імперативно, так і декларативно.

Почнемо з (дуже) наївного імперативного рішення:



Це, звичайно, частково доведено до крайності, і навіть за повністю імперативної парадигми програмування це може бути набагато коротшим.

Давайте реалізуємо цей самий приклад за допомогою мислення декларативного програмування:

```
function countGoodPasswords(passwords) {
  return passwords.filter(p => p.length >= 9).length;
}
```

Ми досягаємо мети безпосередньо в одному операторі, маніпулюючи об'єктом у кілька кроків, використовуючи композицію функції, щоб досягти мети. Ми фільтруємо вихідний масив, щоб отримати тимчасове значення, яке є масивом лише хороших паролів. Однак ми ніколи ніде не зберігаємо цей масив; ми переходимо безпосередньо до наступного кроку визначення довжини цього масиву.

Це був лише загальний код JavaScript. Як це пов'язано з React? React використовує той самий декларативний підхід, коли ви створюєте UI. По-перше, розробники React описують елементи UI у декларативному стилі.

Потім, коли відбуваються зміни у представленнях, згенерованих цими елементами інтерфейсу, React піклується про оновлення. ура!

Зручність декларативного стилю React сяє в повній мірі, коли вам потрібно внести зміни у представлення. Це так звані зміни внутрішнього стану. Коли стан змінюється, React оновлює перегляд відповідно.

ПРИМІТКА Ми розглянемо, як працюють стани, у розділі 5.

КОМПОНЕНТНА АРХІТЕКТУРА З ВИКОРИСТАННЯМ ЧИСТОГО JAVASCRIPT

CBA (Component-Based Architecture) існувала до появи React. Поділ проблем, слабкий зв'язок і повторне використання коду є основою цього підходу, оскільки він дає багато переваг; Інженери програмного забезпечення, включаючи веб-розробників, люблять CBA. Будівельним блоком CBA в React є клас компонентів. Як і в інших CBA, він має багато переваг, головною з яких є повторне використання коду (ви можете писати менше коду!).

До React не вистачало чистої реалізації цієї архітектури на JavaScript. Коли ви працюєте з Angular, Backbone, Ember або більшістю інших фреймворків, подібних до Model-View-Controller (MVC), у вас є один файл для JavaScript, а інший — для шаблону. (Angular використовує термін *директиви* для компонентів.)

Існує кілька проблем із наявністю двох мов (і двох чи більше файлів) для одного компонента. Розділення HTML і JavaScript добре спрацювало, коли вам потрібно було відобразити HTML на сервері, а JavaScript використовувався лише для того, щоб ваш текст миготів. Тепер односторінкові програми (SPA) обробляють складний ввід користувача та виконують рендеринг у браузері. Це означає, що HTML і JavaScript тісно пов'язані функціонально. Для розробників має сенс не вимагати розділення HTML і JavaScript під час роботи над частиною проекту (компонентом).

Під капотом React використовує віртуальний DOM, щоб знаходити відмінності (дельта) між тим, що вже є в браузері, і новим переглядом. Цей процес називається розрізненням DOM або узгодженням стану та вигляду (*diffing* or *reconciliation of state and view* - повернення їх до подібності). Це означає, що розробникам не потрібно турбуватися про явну зміну перегляду;

все, що їм потрібно зробити, це оновити стан, і перегляд буде оновлено автоматично за потреби. Ви побачите, як ми неявно використовуємо цю концепцію знову і знову в книзі. Ми ніколи не маніпулюємо DOM безпосередньо; ми дозволили React зробити цю роботу за нас.

І навпаки, з jQuery вам потрібно обов'язково впроваджувати оновлення. Маніпулюючи DOM, розробники можуть програмно змінювати частини веб-сторінки без повторного рендерингу всієї сторінки. Маніпуляції з DOM – це те, що ви робите, коли викликаєте методи jQuery.

Подумайте про допомогу, яку надає базова структура, у шкалі, як показано на малюнку 1.1. На одному кінці шкали у вас є «framework», який насправді вам зовсім не допомагає. Якби ви створили свою програму на простому JavaScript, ви б опинилися в цій крайності. Використання jQuery полегшило б маніпулювання DOM, але ви все одно не отримували б допомоги від фреймворку, коли щось оновлюється. Вам доведеться вручну переконатися, що ваші перегляди jQuery оновлюються, коли оновлюються дані jQuery.

Малюнок 1.1 Наскільки вам допомагає фреймворк? jQuery нічого не робить; Angular робить все. Для деяких React є найкращим місцем між ними.

На іншому кінці шкали ми маємо такі фреймворки, як Angular, який є ще одним дуже популярним фреймворком, який у всіх відношеннях можна порівняти з React. Однак Angular працює принципово по-іншому, оскільки за лаштунками відбувається набагато більше «магії». Ви часто просто описували, як ваші компоненти підходять один до одного, а Angular намагатиметься правильно з'єднати речі за лаштунками. Проблема з Angular полягає в тому, що ви часто втрачаєте бажаний тонкий контроль, якщо щось працює некоректно. Багато речей приховано від вас, що робить речі непотрібно складними.

React вражає тим щасливим середовищем, де фреймворк допомагає вам виконувати багато виснажливої роботи з підключення різних речей за лаштунками, але не позбавляючи вас тонкого контролю, необхідного для

створення складних веб-додатків. Очевидно, це суб'єктивна думка, але ми не єдині, хто так вважає.

ПОТУЖНІ АБСТРАКЦІЇ

React поставляється з такими чудовими абстракціями, які полегшують життя розробника React:

- Синтетичні події, що абстрагують відмінності веб-переглядача у нативних подіях
- JavaScript XML (JSX) абстрагування JS DOM
- Незалежність від браузера, що дозволяє рендеринг у небраузерних середовищах (наприклад, на сервері)

React має потужну абстракцію моделі подій браузера. Іншими словами, він приховує базові інтерфейси та надає нормалізовані/синтезовані методи та властивості. Наприклад, коли ви створюєте подію onClick у React, замість того, щоб обробник події отримував нативний об'єкт події браузера, він отримує синтетичний об'єкт події, який є оболонкою навколо нативних об'єктів події. Ви можете очікувати такої самої поведінки синтетичних подій незалежно від браузера, у якому ви запускаєте код. React також має набір синтетичних подій для сенсорних подій, які чудово підходять для створення веб-додатків для мобільних пристроїв.

Крім того, є JSX, який є одним із найбільш суперечливих елементів React. Для деяких абстракція JSX є вагомим аргументом на користь використання React, тоді як для інших JSX був каменем спотикання або навіть стримуючим фактором.

Якщо ви знайомі з Angular, то вам уже доводилося писати багато JavaScript у коді свого шаблону, тому що в сучасній веб-розробці звичайний HTML є надто статичним і навряд чи використовується сам по собі. Наша порада полягає в тому, щоб не сумніватися в React і дати JSX чесну роботу.

JSX — це трохи синтаксичного цукру на основі JavaScript для запису елементів React у JavaScript, використовуючи HTML-подібну нотацію з <>. React чудово поєднується з JSX, оскільки розробники можуть краще реалізувати та читати код. Подумайте про JSX як про міні-мову, скомпільовану в нативний JavaScript. Отже, JSX не запускається в браузері, а

використовується як вихідний код для компіляції. Ось компактний фрагмент, написаний на JSX:

```
if (user.session) {
    return <a href="/logout">Logout</a>;
} else {
    return <a href="/login">Login</a>;
}
```

Навіть якщо ви завантажуєте файл JSX у свій браузер із бібліотекою трансформатора часу виконання, яка під час виконання компілює JSX у рідний JavaScript, ви все одно не запустите JSX; замість цього ви запускаєте JavaScript. У цьому сенсі JSX схожий на CoffeeScript. Ви компілюєте ці мови в нативний JavaScript, щоб отримати кращий синтаксис і функції, ніж ті, які надає звичайний JavaScript.

Ми знаємо, що декому з вас виглядає дивно мати HTML, вставлений у код JavaScript. Кожному новому розробнику React (включно з нами) потрібен час, щоб адаптуватися, тому що ми очікуємо лавину повідомлень про синтаксичні помилки. І так, використання JSX необов'язкове. З цих двох причин ми не розглядатимемо JSX до третього розділу. Однак повірте нам — він дуже потужний і навіть викликає звикання, коли ви з ним ознайомитеся.

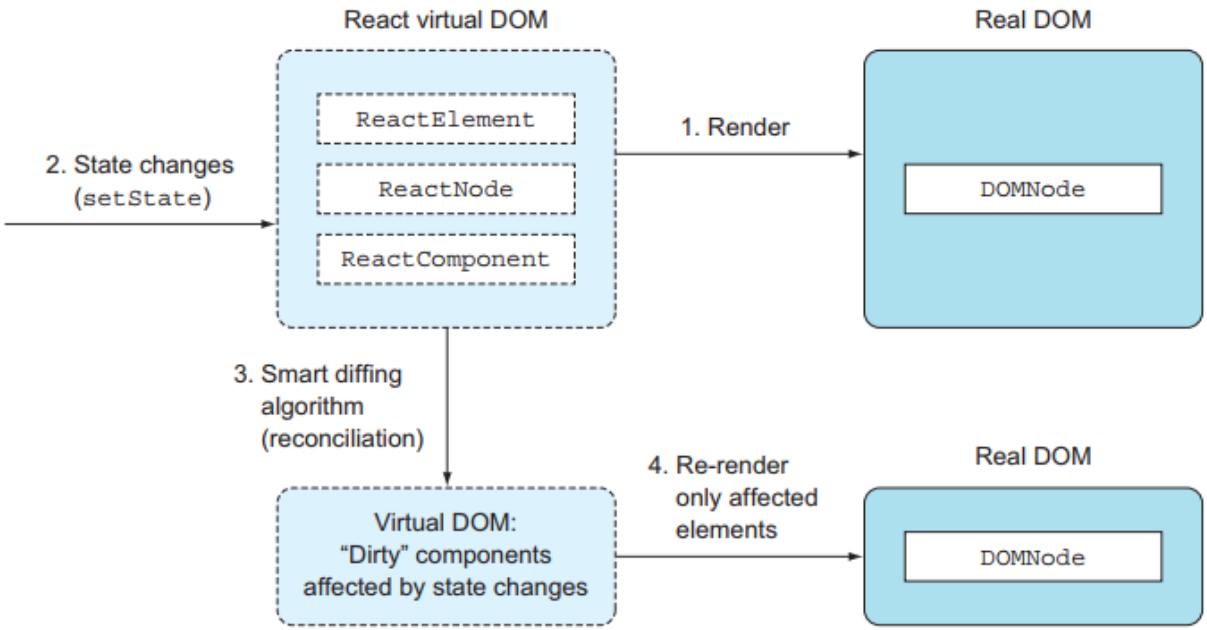
Іншим прикладом абстракції DOM React є те, що ви можете рендерити елементи React на сервері. Це може бути зручно для кращої оптимізації пошукових систем (SEO) і підвищення продуктивності.

Є багато варіантів, коли справа доходить до рендерингу компонентів React як у рядках DOM, так і в HTML на сервері. Ви навіть можете використовувати гібридні підходи, коли ваші шаблони рендеряться з певним вмістом на сервері, а потім доповнюються живими даними в браузері. Ми поговоримо про це докладніше в розділі 1.3. І, якщо говорити про DOM, однією з найбільш затребуваних переваг React є його чудова продуктивність.

1.1.2 Швидкість і можливість тестування

Окрім необхідних оновлень DOM, ваша структура може виконувати непотрібні оновлення, що ще погіршує продуктивність складних інтерфейсів користувача. Це стає особливо помітним і болючим для користувачів, коли на вашій веб-сторінці багато динамічних елементів інтерфейсу.

З іншого боку, віртуальний DOM React існує лише в пам’яті JavaScript. Щоразу, коли відбувається зміна даних, React спочатку порівнює відмінності за допомогою віртуальної DOM; лише коли бібліотека дізнається, що відбулися зміни у візуалізації, вона оновить фактичний DOM. На малюнку 1.2 показаний огляд високого рівня того, як працює віртуальний DOM React, коли є зміни даних.



Малюнок 1.2 Коли компонент відтворюється, якщо його стан змінюється, він порівнюється з віртуальною DOM у пам’яті та повторно відтворюється, якщо необхідно.

Зрештою, React оновлює лише ті частини, які необхідні, щоб внутрішній стан (віртуальний DOM) і вигляд (справжній DOM) були однаковими. Наприклад, якщо є елемент <p> і ви доповнюєте текст через стан компонента, оновлюватиметься лише текст (тобто внутрішній HTML), а не сам елемент. Це призводить до підвищення продуктивності порівняно з повторним відтворенням цілих наборів елементів або, навіть більше, цілих сторінок (відтворення на стороні сервера).

Приголомшливі подробиці Reconciliation

Якщо ви любите розбиратися в алгоритмах і нотації Big O, ці дві статті чудово пояснюють, як команді React вдалося перетворити проблему O(n3) на O(n):

 «Reconciliation» на веб-сайті React (http://mng.bz/PQ9X)

 «React’s Diff Algorithm» Крістофера Шедо (http://mng.bz/68L4)

Додаткова перевага віртуального DOM полягає в тому, що ви можете проводити модульне тестування без headless браузерів, таких як PhantomJS (http://phantomjs.org). Існує кілька бібліотек, у тому числі Jest і React Testing Library, які дозволяють тестувати ваші компоненти безпосередньо з командного рядка. У наступних розділах ми детальніше зупинимося на модульному тестуванні компонентів і хуків React.

1.1.3 Екосистема та спільнота

І останнє, але не менш важливе, React підтримується розробниками веб-додатку Juggernaut під назвою Facebook, а також їхніми колегами в Instagram. Як і у випадку з Angular та деякими іншими бібліотеками, наявність великої компанії, що стоїть за цією технологією, забезпечує надійний полігон (її розгортають у мільйонах браузерів), впевненість у майбутньому та збільшення швидкості внеску. Звичайно, це також ризик, тому що якщо Facebook раптом захоче розгорнути React у новому напрямку, ви можете застрягти, якщо вам не сподобається цей напрямок, тому ретельно зважте свої варіанти.

Уже існує багато чудового вмісту, створеного спільнотою для React. Ви побачите, що коли вам потрібен якийсь компонент чи інтерфейс, ви можете просто шукати в Інтернеті «react [назва-компонента]», і більш ніж у 95% випадків ви знайдете щось варте уваги.

Історія програмного забезпечення з відкритим кодом чітко показує, що маркетинг проектів з відкритим кодом так само важливий для його широкого впровадження та успіху, як і сам код. Під цим ми маємо на увазі, що якщо проект має поганий веб-сайт, бракує документації та прикладів або має потворний логотип, більшість розробників не сприймуть це серйозно, особливо зараз, коли є так багато бібліотек JavaScript. Розробники вибагливі, і вони не будуть використовувати бібліотеку гидкого каченяти.

Як говориться: «Не судіть про книгу за обкладинкою». Це може здатися суперечливим, але, на жаль, більшість людей, включаючи інженерів програмного забезпечення, схильні до упереджень, таких як хороший брендинг. На щастя, React має чудову інженерну репутацію. І, говорячи про

обкладинки книжок, ми сподіваємося, що ви купили цю книгу не лише заради обкладинки!

1.2 Недоліки React

Звісно, майже все має свої недоліки. Це справедливо для React, але повний список недоліків залежить від того, кого ви запитуєте. Деякі з відмінностей, наприклад, декларативність проти імперативності, є дуже суб'єктивними. Вони можуть бути як плюсами, так і мінусами залежно від ваших особистих уподобань. Ось наш список недоліків React (як і будь-який подібний список, він може бути необ'єктивним):

 React не є повноцінним фреймворком типу швейцарського армійського ножа. Розробникам потрібно поєднати його з бібліотекою, такою як Redux або XState, щоб досягти функціональності, порівнянної з Angular або Ember. Це також може бути перевагою, якщо вам потрібна мінімалістична бібліотека інтерфейсу користувача для інтеграції з наявним стеком.

 Стеки React потребують обслуговування та постійного керування пакетами. Оскільки ви ніколи не використовуєте React окремо, а майже завжди поєднуєте його з кількома іншими пакетами, вам потрібно постійно підтримувати свої залежності та переконатися, що ви використовуєте правильні версії різних пакетів. У великих проектах це може стати значним джерелом сторонніх завдань.

 React використовує дещо новий підхід до веб-розробки, а JSX і функціональне програмування можуть налякати новачків. Особливо на початку не вистачало кращих практик, хороших книг, курсів і ресурсів, доступних для освоєння React і подібних фреймворків. Ми обговоримо JSX більш детально в розділі 3.

 React має лише односторонню прив'язку. Хоча одностороннє зв'язування краще для складних веб-додатків і усуває багато складності, деякі розробники (особливо розробники Angular), які звикли до двостороннього зв'язування, будуть писати трохи більше коду. Ми пояснимо, як працює одностороннє зв'язування React у порівнянні з двостороннім зв'язуванням Angular у розділі 9, який охоплює роботу з даними форми.

 React не є реактивним (як у реактивному програмуванні та архітектурі, які є більш керованими подіями, стійкими та чуйними) із коробки. Розробникам потрібно використовувати інші бібліотеки, такі як

бібліотека React Query, щоб їхні програми легко і швидко реагували на зовнішній вміст. Це також вимагає від розробників використовувати інше мислення під час розробки додатків React, інакше додатки будуть жахливо закодовані в результаті спроби змусити круглий React перетворити на квадратну архітектуру.

Щоб продовжити знайомство з React, давайте подивимося, як це вписується у веб-додаток.

1.3 Як React може розміститися на вашому веб-сайті

Веб-сайти мають багато варіантів, і React можна використовувати для створення інтерактивного вмісту на багатьох типах веб-сайтів, або як заміну іншим технологіям, або як спосіб додати нову функціональність до вашого веб-сайту. React можна використовувати як на «класичних» веб-сайтах, які здебільшого відображаються сервером, так і на клієнтських веб-додатках, також відомих як односторінкові додатки (SPA), як згадувалося раніше.

Основна бібліотека React — це перш за все бібліотека інтерфейсу користувача. Одну лише базову бібліотеку можна порівняти з іншими бібліотеками інтерфейсу користувача, але не можна порівняти безпосередньо з більш повноцінними фреймворками веб-додатків, такими як Angular. Однак у поєднанні з іншими бібліотеками, розробленими командою React або іншими сторонами (наприклад, React Router і Redux), React може бути повним конкурентом будь-якій структурі веб-додатків.

Якщо ви використовуєте інший фреймворк SPA (наприклад, Angular, Vue, Ember, Backbone тощо) для візуалізації свого веб-додатку сьогодні, вам, ймовірно, доведеться повністю замінити його на стек на основі React. Дуже важко і майже неможливо створити гібридний SPA, де одні частини відрендерені, наприклад, Angular, а інші React.

Ви можете використовувати React лише для частини свого інтерфейсу, якщо у вас є веб-сайт із меншими інтерактивними елементами інтерфейсу (або віджетами). У такому випадку ви можете замінити свої віджети один за одним невеликими програмами React, не змінюючи все інше. Ці існуючі віджети можуть бути написані на простому JavaScript, jQuery або навіть Angular чи подібних фреймворках. Перетворюючи віджети на React, ви можете оцінити, що найкраще підходить для вашої організації.

React є агностиком бекенда для розробки інтерфейсу. Іншими словами, вам не потрібно покладатися на серверну частину на основі JavaScript (Node або Deno), щоб використовувати React. Добре використовувати React з будь-якою іншою технологією серверної частини, такою як Java, Ruby, Go або Python. Зрештою, React — це бібліотека інтерфейсу користувача. Ви можете інтегрувати його з будь-якою серверною частиною та будь-якою бібліотекою даних зовнішньої частини (Backbone, Angular, Meteor тощо).

Іншим популярним варіантом використання React є генератори статичних сайтів. У такому налаштуванні React використовується для визначення вашого веб-сайту локально у вашому середовищі, але коли його розгортають на живому сервері, він відображається «вниз» до простого веб-сайту HTML із JavaScript, який виконує лише мінімальну роботу для додавання інтерактивності. Усі ваші шаблони тощо буде вирішено. Спочатку це було в основному популярно для невеликих веб-сайтів, таких як блоги, які не надто часто оновлюються.

Нещодавні досягнення в рендерингу React на стороні сервера зробили цей підхід попереднього рендерингу дедалі популярнішим навіть для великих SPA, які часто оновлюються. Ви можете зробити це за допомогою популярних фреймворків, створених на основі React, таких як Next.js або Remix. Вони вважаються частково відтвореними сервером веб-додатками, де ваш код React виконується як на сервері, так і в клієнті. Ви можете, наприклад, попередньо відобразити список на сервері та додати параметри інтерактивного фільтрування та сортування в клієнті. Це може здатися трохи страшним, але новіші фреймворки, такі як Next.js і Remix, роблять це відносно легким.

Щоб підсумувати, як React вписується у веб-сайт, він найчастіше використовується в таких сценаріях:

- Як бібліотека інтерфейсу користувача в SPA, наприклад React+React Router+Redux
- Як додатковий віджет у будь-якому інтерфейсному стеку, як-от компонент введення автозавершення React на веб-сайті, створеному з використанням будь-якої іншої комбінації технологій
- Як статичний веб-сайт, який відображається під час розгортання для надання рідко оновлюваного вмісту

 Як веб-сайт або SPA, які частково відображаються на стороні сервера, створені на основі більш потужної системи, вміст якої потенційно надсилається зовнішньою CMS, такою як WordPress або Contentful

 Як бібліотека інтерфейсу користувача в мобільних програмах, що використовують React Native, або настільних програмах, які використовують Electron

React чудово працює з деякими зовнішніми технологіями, але в основному він використовується як частина SPA. У наступному розділі ми розглянемо, як React вписується в SPA.

1.3.1 Односторінкові програми та React

SPA – це загалом підмножина веб-сайтів. Веб-сайт вважається SPA, якщо він має багато функцій, доступних безпосередньо в браузері, а не лише інформацію. Приклади включають Facebook, Google Docs, Gmail тощо.

SPA створюються з використанням безлічі технологій, з яких React є лише однією потенційною частиною стеку. Ви навіть не можете використовувати React самостійно; принаймні кілька інших технологій необхідні, щоб React можна було використовувати як окрему програму. У цьому розділі ми визначимо, що таке SPA загалом, а потім вкажемо, як React вписується в цю структуру.

SPA також відомі як товсті клієнти, оскільки браузер, будучи клієнтом, містить більше логіки та виконує такі функції, як рендеринг HTML, перевірка, зміни інтерфейсу користувача тощо. Порівняйте це з тонким клієнтом, де клієнт браузера використовується лише для відображення інформації, попередньо відтвореної сервером. У тонкому клієнті браузер виконує дуже мало роботи.

На малюнку 1.3 наведено приклад загального SPA на дуже високому рівні незалежно від використовуваної технології. Він показує типову архітектуру з висоти пташиного польоту з користувачем, браузером і сервером. На малюнку зображено користувача, який робить запит, і дії введення, такі як натискання кнопки, перетягування, наведення миші тощо.

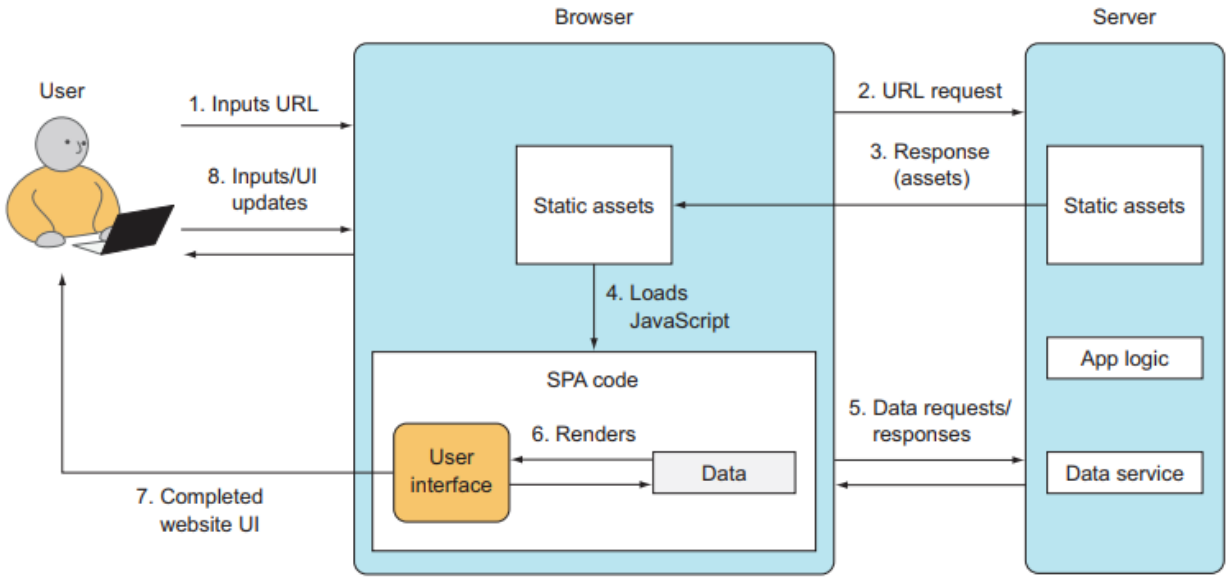


Рисунок 1.3 Загальна архітектура SPA

Давайте пройдемося по цьому типовому наскрізному процесу, дотримуючись пронумерованих кроків на малюнку 1.3:

1 Користувач вводить URL-адресу в браузері, щоб відкрити нову сторінку.

2 Браузер надсилає URL-запит на сервер.

3 Сервер відповідає статичними ресурсами, такими як HTML, CSS і JavaScript. У більшості випадків HTML є простим; тобто він має лише скелет веб-сторінки. Зазвичай є повідомлення «Завантаження . . . ” повідомлення та/або GIF-зображення, що обертається.

4 Статичні ресурси включають код JavaScript для програми. Під час завантаження цей код робить додаткові запити даних.

5 Дані повертаються у форматі JSON, XML або будь-якому іншому.

6 Після того, як програма отримає дані, вона може відобразити відсутній HTML (блок інтерфейсу користувача на малюнку). Інакше кажучи, процес рендерингу інтерфейсу користувача відбувається в браузері, коли програма вводить дані в попередньо відрендерені шаблони, також відомий як гідратація.

7 Після завершення візуалізації веб-переглядач оновлює вміст, що відображається, і користувач може працювати з програмою.

8 Користувач бачить красиву веб-сторінку. Користувач може взаємодіяти зі сторінкою (вхідні дані на малюнку), запускаючи нові запити від програми до сервера, і цикл кроків 2–6 продовжується. На цьому етапі маршрутизація браузера може відбутися, якщо програма реалізує її, тобто перехід до нової URL-адреси ініціюватиме не перезавантаження нової сторінки з сервера, а радше повторне відтворення програми в браузері.

Підводячи підсумок, у SPA більшість візуалізації інтерфейсів користувача відбувається в браузері. У браузер і з нього передаються лише дані. Порівняйте це з «класичним» веб-сайтом, який не є SPA, де весь рендеринг відбувається на сервері. React вписується в цю архітектуру SPA на етапах 6 і 8, відтворюючи вміст на основі даних, а також обробляючи введені користувачем дані та оновлюючи вміст на основі оновлених даних, які є результатом цих введених даних.

1.3.2 Стек React

React не є повноцінним інтерфейсним фреймворком JavaScript SPA. React є мінімалістичним у тому сенсі, що він виконує лише одну роботу (відтворення реактивних інтерфейсів користувача) і намагається робити це дуже добре. Він не нав'язує певний спосіб виконання таких речей, як моделювання даних, стилізація чи маршрутизація (це не самовпевненість). Через це розробникам часто доводиться поєднувати React з бібліотекою маршрутизації та/або даних.

Хоча ви можете використовувати React як меншу частину стеку, розробники найчастіше вибирають стек, орієнтований на React, який складається з самого ядра React, а також бібліотек даних, маршрутизації та стилів, створених спеціально для використання з React, такі як:

- ***Бібліотеки моделей даних і серверні модулі*** — наприклад, TanStack Query (https://tanstack.com/query/latest), Redux (http://redux.js.org), Recoil.js (https://recoiljs.org/), XState (https://xstate.js.org /) і Apollo (www.apollographql.com/)

- ***Бібліотека маршрутизації*** — часто React Router (https://github.com/remix-run/react-router) або подібний маршрутизатор, реалізований у багатьох фреймворках

 Бібліотеки стилів — або попередньо визначений набір стилізованих компонентів, таких як Material UI (https://mui.com/) або Bootstrap (https://react-bootstrap.github.io/), або бібліотека для легкої роботи з CSS усередині Компоненти React, такі як Styled-Components (https://styled-components.com/), Vanilla Extract (https://vanilla-extract.style/) або навіть Tailwind CSS (https://tailwindcss.com/)

Екосистема бібліотек для React зростає щодня. Крім того, здатність React описувати складові компоненти (самодостатні фрагменти інтерфейсу користувача) дозволяє повторно використовувати код. Багато компонентів упаковані як модулі npm.

Чудовий (підібраний) список різноманітних компонентів React для багатьох цілей можна знайти тут: https://github.com/brillout/awesome-react-components . У цьому списку є все: від компонентів інтерфейсу користувача (включно з безліччю елементів форми) до повних інтерфейсів інтерфейсу користувача до утиліт розробки та інструментів тестування.

Фреймворки React

Інша категорія фреймворків React — це повномасштабний серверний фреймворк, який подбає про все за вас. Такі фреймворки бувають двох варіантів, але іноді фреймворк може працювати в будь-який спосіб:

 Генератори статичних сайтів (Static site generators, SSG)

 Динамічний серверний React (Dynamic server-rendered React, SSR)

SSG — це саме ті фреймворки, які створять для вас повністю статичний веб-сайт, повністю готовий до розгортання на будь-якому хості статичного веб-сайту, що вимагає дуже мало роботи з вашого боку та жодного дорогого хостингу. Це особливо популярно для невеликих персональних веб-сайтів, таких як блоги, але також може використовуватися для невеликих підприємств і навіть веб-сайтів електронної комерції (які не потребують надто частого оновлення).

Фреймворки SSR є складнішими та подбають про попередню візуалізацію вашого додатка React на сервері перед тим, як надсилати HTML по мережі до браузерів ваших відвідувачів. Це означає, що це добре для SEO, забезпечує можливість спільного використання та має багато інших переваг.

Тут ми перерахуємо три такі фреймворки:

 Gatsby — ця дуже популярна платформа для ведення блогів також корисна для багатьох інших типів статичних веб-сайтів.

 Next.js — як, мабуть, найпопулярніший фреймворк для веб-сайтів React, він корисний як для невеликих статичних веб-сайтів, так і для величезних динамічних гігантів.

 Remix— цей досить новачок у блоці дуже швидко набирає обертів і популярності, обслуговуючи надшвидкі динамічні веб-сайти React.

Усі ці фреймворки — і багато, багато іншого — є різними розширеннями React, кожний з яких функціонує за власною парадигмою. Усі вони додають додаткову функціональність на додаток до React, а іноді також постачаються з набором компонентів React, які допомагають вам створити свій веб-сайт, щоб максимально використовувати фреймворк.

Наразі ви маєте зрозуміти, що таке React, його стек, його місце у веб-додатках вищого рівня та як ви можете використовувати інструменти, створені на основі React, для створення складних веб-сайтів. Настав час забруднити руки і написати свій перший додаток React.

1.4 Ваш перший додаток React: Hello World

Давайте дослідимо вашу першу програму React, реалізувавши програму Hello World — квінтесенцію прикладу, який використовується для вивчення мов програмування (див. рис. 1.4). Якщо ми цього не зробимо, боги програмування можуть покарати нас.

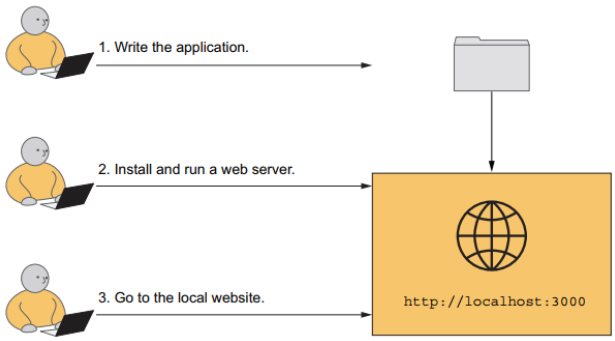


Рисунок 1.4 Процес створення вашої першої програми React складається лише з трьох простих кроків.

Перш ніж почати, вам знадобиться кілька речей. На щастя, оскільки ми розробляємо програму, яка працює в браузері, вам не потрібні всілякі компілятори чи бібліотеки. Ось короткий список речей, які вам знадобляться, перш ніж ви зможете почати:

 Текстовий редактор.

 Знання того, як використовувати термінал у вашій системі.

 Встановіть npm версії 5.2 або новішої (враховуючи, що версія 5.2 існує з липня 2017 року, велика ймовірність того, що ваша версія npm буде достатньо хорошою, якщо вона у вас є).

 Встановіть сучасний браузер (працює будь-яка остання версія Edge, Firefox, Chrome або Safari)

І це приблизно все. Якщо ви можете взяти позначку з цього списку, ви можете вибрати цей перший приклад. Коли ми перейдемо до інших прикладів у наступних розділах, вам не знадобиться набагато більше, ніж те, що є в цьому списку.

1.4.1 Результат

Проект надрукує «Hello world!!!» заголовок (<h1>) на веб-сторінці. На малюнку 1.5 показано, як це виглядатиме, коли ви закінчите (якщо ви не дуже захоплені і віддаєте перевагу лише одному знаку оклику

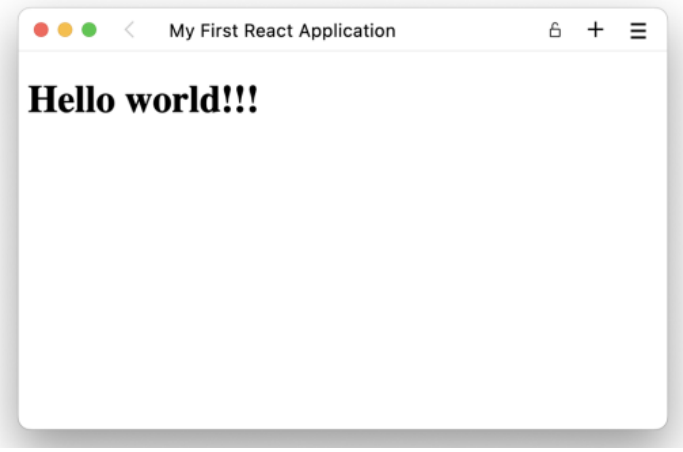


Рисунок 1.5 Додаток Hello World

Ви ще не будете використовувати JSX, а лише звичайний JavaScript (насправді ми не почнемо використовувати JSX до глави 3 і далі).

1.4.2 Написання додатку

Цей проект настільки простий, що він складатиметься лише з одного файлу HTML. Цей файл міститиме посилання на найновіші версії React 18 (найстабільніша версія на момент написання) бібліотек React Core і ReactDOM. Він також, звичайно, включатиме крихітний код JavaScript, необхідний для відтворення дуже простого додатка, який ми створюємо.

Код для файлу HTML простий і починається з включення бібліотек у <head>. В елементі <body> ви створите контейнер із кореневим ідентифікатором і елементом <script> (куди пізніше буде розміщено код програми), як показано в наступному лістингу.

Лістинг 1.1 Завантаження бібліотек і коду React

```
<!DOCTYPE html>
<html>
 <head>
  <title>My First React Application</title>
  <script
     src="//unpkg.com/react@18/umd/react.development.js">
  </script>
  <script src="//unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
 </head>
 <body>
  <div id="root"></div>
  <script type="text/javascript">
  ...
  </script>
 </body>
</html>
```

Просто введіть цей код за допомогою текстового редактора та збережіть його як файл з іменем index.html у папці на вашому комп’ютері.

Можливо, вам цікаво, чому ми повинні створити вузол для відтворення вмісту, а не відображати елемент React безпосередньо в елементі <body>. Відповідь полягає в тому, що це може призвести до конфлікту з іншими бібліотеками та розширеннями браузера, які маніпулюють тілом документа. Якщо ви спробуєте приєднати елемент безпосередньо до тіла, ви отримаєте консольну помилку:

Включивши бібліотеки у файл HTML, ви отримуєте доступ до глобальних об’єктів React і ReactDOM: window.React і window.ReactDOM. Вам знадобляться два методи з цих об’єктів: один для створення елемента (React), а інший для відтворення його в контейнері (ReactDOM), як показано в лістингу 1.2. Щоб створити елемент React, все, що вам потрібно зробити, це викликати React.createElement(elementName, data, children) з трьома аргументами, які мають такі значення:

 elementName—тег HTML у вигляді рядка (наприклад, 'h1') або власний клас компонента як об'єкт. У нас поки що немає спеціальних компонентів, але ми почнемо створювати їх у розділі 2.

 data— об’єкт даних, що містить атрибути та властивості елемента. Зараз нам не потрібні жодні властивості, тому ми просто передаємо null. Ми повернемося до використання властивостей у розділі 2.

 children – дочірні елементи або внутрішній HTML/текстовий вміст. У цьому прикладі це просто «Hello world!!!».

Лістинг 1.2 Створення та рендеринг елемента h1

```
const reactElement = React.createElement(
 'h1',
 null,
 'Hello world!!!'
);
const domNode = document.getElementById('root');
const root = ReactDOM.createRoot(domNode);
root.render(reactElement);
```

Код у лістингу 1.2 вставляється в тег <script> у файлі HTML, який ви створили раніше, замість ..., який ми спочатку розмістили там як заповнювач. Цей лістинг отримує елемент React і зберігає посилання на цей об’єкт у змінній reactElement. Змінна reactElement не є справжнім вузлом DOM; скоріше, це екземпляр компонента (елемента) React h1. Ви можете назвати його як завгодно, наприклад, helloWorldHeading. Іншими словами, React забезпечує абстракцію над DOM.

Після того, як елемент створено та збережено в змінній, ви створюєте власника програми React (називається root) з елемента DOM за допомогою методу ReactDOM.createRoot(). Нарешті, ви відтворюєте елемент React у root за допомогою методу root.render(), показаного в лістингу 1.2.

За бажанням ви можете перемістити всі кроки в один виклик. Результат той самий, за винятком того, що ви не використовуєте три додаткові змінні, як ми зробили в наступному списку.

Лістинг 1.3 Одиночний оператор

```
ReactDOM
.createRoot(document.getElementById('root'))
.render(React.createElement('h1', null, 'Hello world!'));
```

У лістингу 1.2 ми будемо використовувати більш чітку версію, тому повний HTML-файл має виглядати так, як наведено нижче.

Лістинг 1.4 Створення та рендеринг елемента h1

```
<!DOCTYPE html>
<html>
 <head>
  <title>My First React Application</title>
  <script
src="//unpkg.com/react@18/umd/react.development.js"></script>
  <script src="//unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
 </head>
 <body>
  <div id="root"></div>
  <script type="text/javascript">
   const reactElement = React.createElement(
    "h1",
    null,
    "Hello world!!!"
   );
   const domNode = document.getElementById("root");
   const root = ReactDOM.createRoot(domNode);
   root.render(reactElement);
  </script>
 </body>
</html>
```

Коли файл HTML завершено, тепер нам потрібно побачити це в дії, надаючи вміст у наш браузер.

1.4.3 Встановлення та запуск веб-сервера

Тепер настає наступний крок — надання HTML-сторінки браузеру. Чому ми повинні обслуговувати контент? Хіба ми не можемо просто відкрити файл HTML безпосередньо в браузері? Через перехресні обмеження ви не можете відкрити файл, розташований на локальному жорсткому диску, у браузері та отримати доступ до вмісту в інших доменах (наприклад, бібліотеки React, завантажені з https://unpkg.com). Браузери просто не дозволяють цього. Ви можете спробувати відкрити файл безпосередньо у своєму браузері, двічі клацнувши його, але відобразиться лише порожня біла сторінка. Тож це не добре.

Натомість нам потрібно обслуговувати вміст за допомогою локального веб-сервера розробки. Це може здатися надзвичайно складним, але сьогодні це напрочуд просто зробити.

Якщо у вас налаштовано Node, як рекомендовано у вступі, цього буде достатньо, щоб почати роботу. Просто введіть таку команду в папку, де ви зберегли файл index.html:

```
$ npx serve
```

Це воно. Вас можуть попросити встановити пакет (якщо ви раніше не використовували цю команду, просто натисніть Enter для підтвердження), але через кілька секунд, коли інструмент повідомить, що все працює, ваш веб-сервер запущено.

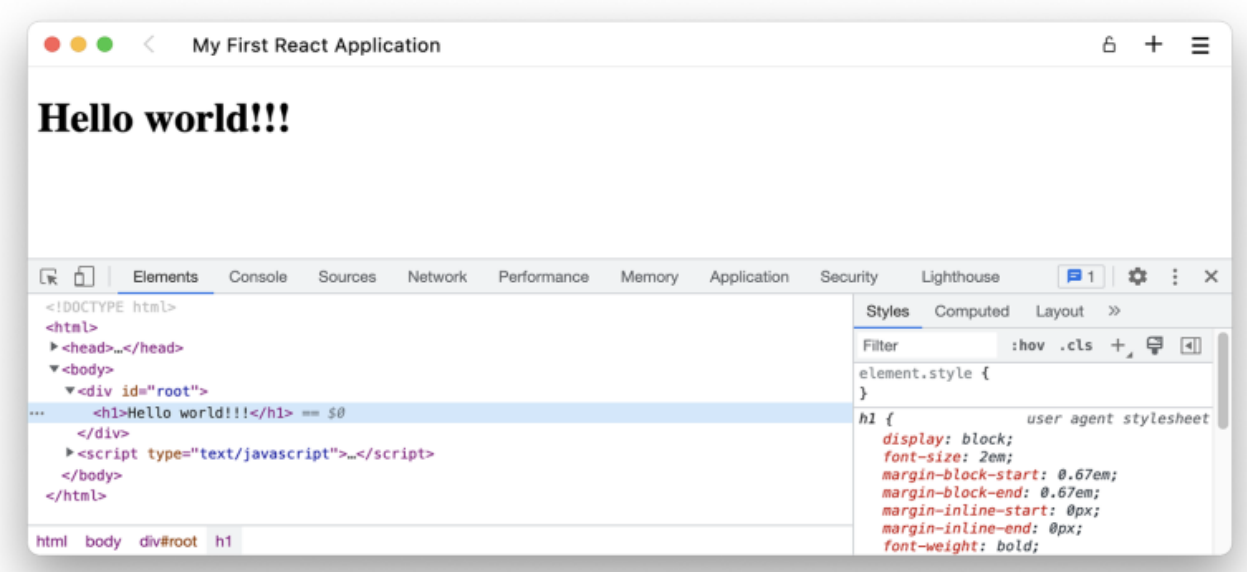
1.4.4 Перехід на локальний веб-сайт

Коли веб-сервер працює, тепер ви можете використовувати свій браузер і перейти на цей сайт:

```
http:/ /localhost:3000
```

Тут ви зможете побачити свою програму в дії, і вона має виглядати майже як малюнок 1.5 на початку цього розділу.

На малюнку 1.6 показано вкладку «Елементи» в інструментах розробника браузера з вибраним елементом <h1>. Ви знаєте, що React, мабуть, щось тут зробив, тому що у вашому вихідному HTML-файлі в кореневому вузлі немає елемента <h1> — він був порожнім. Щиро вітаю! Ви щойно запровадили свою першу програму React!



Малюнок 1.6 Перевірка веб-програми Hello World, відтвореної React

Починаючи з наступного розділу, ми не будемо створювати такі наші React-додатки. Ми будемо використовувати невеликий інструмент для швидкого створення та налаштування основної програми React для нас, що зробить увесь цей процес набагато зручнішим. Він також подбає про обслуговування нашого вмісту, тому вам більше не доведеться турбуватися про веб-сервери.

Резюме

 React для Інтернету складається з бібліотек React Core і ReactDOM. ReactCore — це бібліотека, призначена для створення та спільного використання складових компонентів інтерфейсу користувача за допомогою JavaScript і (опціонально) JSX універсальним способом. З іншого боку, для роботи з React у браузері ви можете використовувати бібліотеку ReactDOM, яка має методи для відтворення DOM, а також для відтворення на стороні сервера.

 React є декларативним; це лише перегляд або рівень інтерфейсу користувача.

 React використовує компоненти, які ви створюєте за допомогою ReactDOM.createRoot().

 Ви використовуєте чистий JavaScript для розробки та створення інтерфейсів користувача в React.

 Вам не обов’язково використовувати JSX (синтаксис, схожий на HTML для об’єктів React) під час розробки з React, але це потрібно всім.

 React може вписуватися у ваш веб-стек багатьма способами, від просто маленького віджета на деякій сторінці до основи всього вашого веб-сайту.

 React — це не швейцарський армійський ніж, а радше рівень інтерфейсу користувача веб-додатку, який також складається з багатьох інших частин. React часто використовується разом із бібліотеками даних, такими як Redux або XState.

Розділ 2

Дитячі кроки з React

Цей розділ охоплює

 Створення нового проекту React

 Елементи вкладеності

 Створення класу компонентів

 Робота з властивостями

Цей розділ навчить вас, як створити новий проект React і як створити власні компоненти для відтворення HTML. Обидві ці концепції слугуватимуть основою для всіх наступних розділів.

Спочатку ми розглянемо, як створити новий проект React. Роблячи це, ми навчимо вас як запускати власні проекти React, так і як використовувати систему шаблонів React для швидкого створення прикладів і проектів, над якими ми працюватимемо в цій книзі. Надзвичайно чарівно, як одним рядком ви можете отримати код, завантажений і готовий до роботи з усім налаштованим для вас!

Починаючи наш перший проект React, ми познайомимося з кількома фундаментальними концепціями React, якими ви будете часто користуватися, включаючи елементи, компоненти та властивості. У двох словах, елементи — це екземпляри компонентів, яким можна передати властивості. Які варіанти їх використання та чому ви їх використовуєте? Чекайте цієї інформації до

розділу 2.3, тому що зараз ми обговоримо, як створити нову веб-програму React.

ПРИМІТКА. Вихідний код для прикладів у цій главі доступний за адресою https://rq2e.com/ch02 . Однак у розділі 2.2 ви дізнаєтесь, що вам не потрібно нічого завантажувати вручну. Ви можете створити екземпляри всіх прикладів із цього та наступних розділів безпосередньо з командного рядка за допомогою однієї команди.

2.1 Створення нового додатка React

У цьому розділі ми познайомимо вас із чарівною програмою командного рядка, за допомогою якої всі ваші налаштування React пройдуть гладко. Лише за три короткі команди та пару хвилин ви завантажите повноцінну фіктивну веб-програму React, скомпілюєте її, запустите через веб-сервер і побачите у своєму браузері (перегляньте огляд на малюнку 2.1).

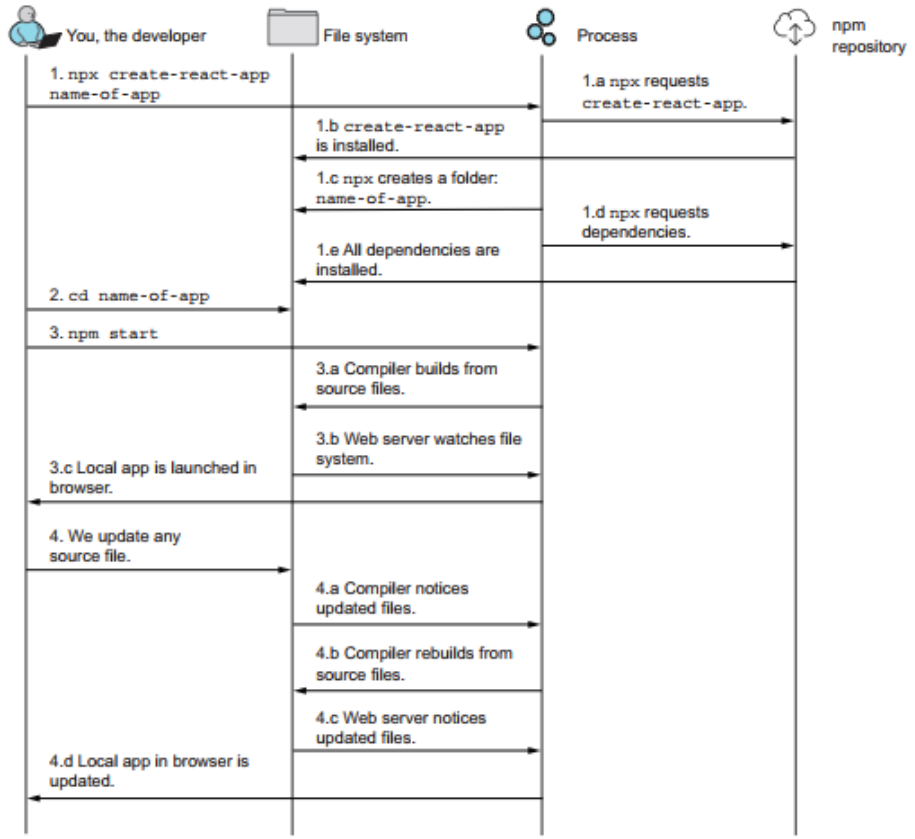


Рисунок 2.1 Три команди, які переведуть вас із нічого до робочої програми React. Звідти ви можете оновити вихідні файли, і система автоматично перекомпілює та оновить вашу програму в браузері.

Якщо у вас встановлено сучасну версію Node (і npm), як зазначено у вступі, ви зможете написати таку команду у своєму терміналі:

```
$ npx create-react-app name-of-app
```

ПРИМІТКА. npx не є помилкою. npx — це інструмент запуску пакетів, який постачається з npm. Це дозволяє нам запускати команди, використовуючи пакети, наявні лише в цій папці проекту, і/або запускати команди, які за потреби завантажуватимуться динамічно.

Виконайте цю команду, і новий додаток React буде створено для вас! Під час першого запуску цієї команди npm запитає підтвердження для завантаження утиліти create-react-app (просто натисніть Enter, щоб підтвердити це). Це відноситься до кроків 1.a і 1.b на малюнку 2.1. Кожного разу, коли ви використовуєте цю команду після цього, жодних запитань не буде.

ПРИМІТКА У наступних розділах ми називатимемо інструмент create-react-app CRA.

Команда створить нову папку з переданим іменем, яким у попередньому випадку є ім’я програми. У цій папці утиліта ініціалізує новий проект Git, завантажить необхідні ресурси для програми, а потім завантажить і локально встановить усі залежності, необхідні для проекту.

Команда виконуватиметься протягом короткого часу, ймовірно, близько 1–3 хвилин залежно від складності проекту та умов мережі. Після завершення команди ви побачите щось на зразок цього:

```
Success! Created name-of-app at <folder>
Inside that directory, you can run several commands:

npm start
Starts the development server.

npm run build
Bundles the app into static files for production.


npm test
Starts the test runner.


npm run eject
```

```
Removes this tool and copies build dependencies, configuration
files and scripts into the app directory. If you do this, you
can't go back!
We suggest that you begin by typing:


cd name-of-app
npm start


Happy hacking!
```

Ой, захоплююче. Зауважте, що якщо у ваших результатах згадується команда під назвою yarn, а не npm, не хвилюйтеся. Перегляньте пояснення на бічній панелі.

Альтернативи npm

Є кілька популярних менеджерів пакунків для проектів JavaScript, які працюють з тим же сховищем пакетів і структурою, але з дещо різними командами. Найбільш використовуваним менеджером є npm, але альтернативи включають Yarn і pnpm. Популярний вибір, npm, поставляється з попередньо встановленим Node і є менеджером за замовчуванням, яким користуються багато людей.

Однак ви можете вибрати інший менеджер, який матиме дещо простішу структуру команд. Для цілей цієї книги немає ніякої різниці між використанням npm чи альтернативи, окрім дещо іншого синтаксису під час введення команд. Якщо у вас є Yarn або pnpm, ви, ймовірно, також маєте встановлений npm, тому, швидше за все, це завжди працюватиме для вас.

Якщо ви хочете використовувати один із цих менеджерів пакунків, перегляньте документацію про те, як запускати команди:

- Yarn: https://classic.yarnpkg.com/lang/en/docs/cli/run/
- pnpm: https://pnpm.io/cli/run

Тепер давайте виконаємо пропозицію в попередньому фрагменті коду та запустимо ці дві команди:

```
$ cd name-of-app
$ npm start
```

Тепер відбувається третя частина магії. Сервер розробки React запускається, компілюючи всі використані файли та ресурси (дія 3.a на малюнку 2.1) і запускає локальний веб-сервер розробки (дія 3.b на малюнку 2.1). Через кілька секунд командний рядок скаже приблизно так:

```
Compiled successfully!
You can now view name-of-app in the browser.
  Local: http:/ /localhost:3000
```

Крім того, програма вже буде запущена у вашому браузері, оскільки команда також запускає вікно браузера за правильною URL-адресою (дія 3.c на малюнку 2.1). Якщо ні, просто відкрийте localhost:3000 у своєму браузері, щоб побачити програму. У цьому вікні браузера буде показано програму React (як показано на малюнку 2.2), створену для вас за допомогою шаблону. Це типовий шаблон, який використовується для нових додатків React, які не вказують конкретний шаблон для використання. Ми обговоримо шаблони трохи пізніше в розділі 2.1.3.

Зауважте, що ця остання команда, npm start, є постійно запущеною командою, яка залишається активною в терміналі. Він спостерігатиме за вашими вихідними файлами, перекомпілює всю програму, коли будь-який вихідний файл змінюється, і навіть перезавантажуватиме браузер із оновленою програмою (дії 4–4.d на малюнку 2.1)! Тепер це чиста магія.

Рисунок 2.2 Додаток React за замовчуванням, запущений новим проектом React. Ваша програма, швидше за все, працюватиме в темному режимі з білим текстом на темному тлі. Яскравість на цьому знімку екрана інвертовано для кращих результатів друку.

Якщо і коли ви захочете скасувати цю команду, просто натисніть Ctrl-C у вашому терміналі, і ви повернетеся до звичайного терміналу. Однак ваша програма більше не працює, оскільки ви також зупинили локальний веб-сервер розробки.

Можливо, ви помітили, що в попередніх результатах створення нашої програми містилася не лише команда start, яку ми щойно використали, а й три інші команди: build, test і eject. Ми розглянемо всі ці чотири команди більш детально в наступному підрозділі.

2.1.1 Команди проекту React

Тепер, коли у вашій системі є вихідний код програми React, ви, ймовірно, захочете взаємодіяти з ним кількома способами. Дві головні речі, які ви хочете, це бачити, що ви розробляєте під час розробки, і розгорнути свою програму на веб-сервері. Ви також можете запустити всі тести у своїй програмі, щоб переконатися, що все досі працює, як задумано. Нарешті, ви можете уникнути обмежень CRA, щоб повозитися з двигуном під ним. CRA абстрагує деякі речі, про які вам не потрібно турбуватися спочатку, але коли програми стануть більш просунутими, ви можете отримати доступ до внутрішньої частини конфігурації програми. Для цих чотирьох цілей нова програма React, створена за допомогою CRA, має такі чотири команди:

 start — запустить локальний веб-сервер розробки та постійно компілює проект у міру його змін, надсилаючи його в будь-який локальний браузер.

 build — скомпілює всі ресурси в пакет, готовий до виробництва, який можна розгорнути на потрібному веб-хосту.

 test — запустить програму виконання тестів, яка виконуватиме всі модульні тести, визначені у вашому проекті.

 eject — розкриє внутрішню роботу проекту та зробіть його повністю конфігурованим.

2.1.2 Структура файлів

Коли ви створюєте проект за допомогою CRA, він майже завжди дотримується тієї самої файлової структури. Користувацькі шаблони можуть робити щось по-іншому, але рідко. Структура включає такі важливі елементи:

```
/
  public/
    index.html
  src/
    index.js
    App.js
  package.json
```

Усього дві папки та чотири файли.

Спільна папка призначена для файлів, які обслуговуватимуться безпосередньо через веб-сервер. Це включає файл index.html, який обслуговує всю вашу програму, а також двійкові файли, які ви не бажаєте об'єднувати у свою програму, як-от вміст, потрібний безпосередньо файлу index.html (наприклад, favicon, каскадні таблиці стилів [CSS).], шрифти або зображення для спільного використання) і великі файли (наприклад, відео та зображення).

Вихідна папка (src) – це те, куди буде поміщено весь ваш пакетний JavaScript, а також будь-який інший вміст, який ви хочете об'єднати як єдиний пакет. Це здебільшого лише JavaScript, але потенційно також може включати CSS, значки, маленькі зображення, файли JSON тощо. Групування починається з файлу index.js у вихідній папці. Зазвичай основна програма розміщується у файлі з назвою App.js або app.js, залежно від особистих уподобань, але в іншому випадку ви можете бути гнучкими. Деякі шаблони структурують вміст папки src у підпапках, що необхідно для структурування великих проектів.

Основним конфігураційним файлом для вашого проекту є package.json, як того вимагає npm і Yarn. Це початковий файл для вашого проекту, який визначає залежності, а також команди, які ви можете виконувати, як описано в розділі 2.1.1.

Коренева папка часто містить масу інших конфігураційних файлів, необхідних для різних бібліотек, включених до проекту. Це не рідкість

побачити спеціальні шаблони з 20+ файлами конфігурації в корені проекту. Тепер давайте розглянемо, що таке спеціальні шаблони та як вони вам допомагають.

2.1.3 Шаблони

Хоча типова програма, яку ми бачили на малюнку 2.2, досить гарна, вона не завжди корисна. Програма за замовчуванням налаштовує вас на створення простої веб-програми в тому самому стилі, що й ця веб-програма, але це може бути не те, що ви шукаєте. Якщо ви хочете створити веб-додаток, використовуючи певний стек технологій або використовуючи React певним чином, ви, ймовірно, захочете використати інший стартовий шаблон, щоб правильно налаштувати себе.

Коли ви використовуєте CRA, ви можете вказати шаблон для використання. Шаблон за замовчуванням — це той, який ви бачили раніше з (крутим) логотипом React. Якщо ви хочете вказати інший шаблон, ви можете зробити це як аргумент:

```
$ npx create-react-app name-of-app --template name-of-template
```

Ви можете використовувати лише назву шаблону, який уже існує; якщо його не існує, програма буде перервана. Часто люди взагалі не обирають шаблон і просто працюють із стандартним. Але якщо ви знаєте, що вам потрібна конкретна настройка або хочете запустити свою кодову базу в певному стані, ви можете скористатися шаблоном, який налаштує вас саме на це. Деякі шаблони, які часто використовуються, включають:

- Мінімальні шаблони з навіть меншою кількістю функцій, ніж стандартний, наприклад, --template minimal. Він поставляється без зображень, CSS, тестів, веб-показників та інших дрібниць, які використовуються в шаблоні за замовчуванням.

- Варіанти типового або мінімального шаблону з використанням TypeScript, наприклад, --template typescript або --template minimal-typescript. Це корисно для запуску нового проекту за допомогою TypeScript.

- Складні шаблонні налаштування, створені іншими розробниками, де у вас є стек певних залежностей, уже вбудованих у вашу нову програму, наприклад, --template redux-typescript, який постачається попередньо

запакованим із Redux і TypeScript, або --template rb, який є популярний шаблон React (звідси rb), який постачається з безліччю авторитетних бібліотек, включаючи Redux із Redux-Saga, стилізовані компоненти, ESLint, husky та багато іншого.

Однією з дуже корисних особливостей системи шаблонів для CRA є те, що вона повністю децентралізована. Кожен може опублікувати пакет у npm і структурувати його таким чином, щоб ви могли використовувати його як основу для власних програм. Це, звісно, теж один із мінусів. Якщо ви знайшли шаблон на npm, не можна сказати, чи він хороший і чи виконує те, що написано. Тут вам, ймовірно, варто довіритися мудрості натовпу — якщо вона популярна, то, ймовірно, добре.

Однією з переваг дозволу майже будь-якому випадковому розробнику публікувати шаблон на npm є те, що це включає нас, авторів цієї книги. Ми будемо використовувати власні шаблони React для всіх прикладів і проектів у цій книзі. Ми повернемося до цього за секунду. Спочатку ми обговоримо переваги та недоліки використання CRA.

2.1.4 Плюси і мінуси

Використання CRA для створення нової програми React має багато переваг, але, як завжди, такі переваги мають наслідки. Ми вже обговорювали багато переваг, але все ж перерахуємо їх тут:

 Простота — Вам менше потрібно турбуватися під час налаштування нової програми. Ви отримуєте транспіляцію, групування, тестування, автоматичне перезавантаження та багато іншого JavaScript XML (JSX) безкоштовно, не маючи жодних взаємозалежностей.

 Можливість оновлення — Ви можете легко оновити React до новіших версій і всіх інших використовуваних бібліотек. Ми не обговорювали, як це зробити, але це напрочуд просто. Просто запустіть npm install --exact react-scripts@VERSION, щоб оновити весь проект до певної версії сценаріїв React. Подробиці перевірте в журналі змін для сценаріїв реакції.

 Спільнота. Завдяки величезній кількості доступних шаблонів CRA та легкому шляху створення нових ви завжди зможете знайти готовий шаблон із правильною комбінацією інструментів, щоб вам не доводилося мати справу з їх правильним змішуванням.

 Налаштування — окрім різноманітних шаблонів, ви все ще маєте можливість додавати всі інші плагіни та бібліотеки, необхідні для вашого проекту. Чи працює ваш проект, наприклад, із Google Maps і Amazon Web Services (AWS)? Просто додайте їхні бібліотеки, і все буде готово.

Однак є й деякі недоліки. Деякі з них можна проігнорувати або замовчувати, але в деяких ситуаціях вам доведеться шукати інші налаштування, крім того, що може надати CRA. Ми також розглянемо деякі з цих ситуацій тут:

 Розуміння — не створивши весь проект з нуля, ви не знатимете всього, що входить у таку роботу. Якщо ви опинитеся в ситуації, коли вам потрібна унікальна установка, але ви завжди покладалися на CRA, ви можете швидко опинитись у безвиході, оскільки ніколи не звертали на це уваги. Але це подвійність усіх абстракцій: ви отримуєте перевагу від того, що не хвилюєтеся ціною незнання того, що відбувається під ними.

 Контроль — ви втрачаєте контроль над тим, які бібліотеки використовуються. Зараз CRA використовує webpack і BabelJS для об'єднання та транспіляції JSX, але вони аж ніяк не єдині гравці. Нещодавно з'явилися такі інструменти, як esbuild, Bun, SWC і Rome, які частково охоплюють ту саму основу, але ви не можете легко перейти до одного з них. Ви застрягли на наборі технологій, які наразі вибрали для вас CRA. З іншого боку, це також є перевагою, оскільки коли інший інструмент стане стандартним і, можливо, навіть перевершить Babel, CRA адаптує та використовуватиме його замість цього, і вам не доведеться про це турбуватися. У випадках, коли ви наполягаєте на використанні певного стека, вам доведеться налаштувати свій проект з нуля. Іншим варіантом є вилучення програми, як описано в розділі 2.1.1, що дає вам додаткові можливості конфігурації та контролю ціною втрати можливості оновлення.

 Інтеграція. Якщо ви хочете інтегрувати свою програму в налаштування на стороні сервера, CRA наразі не може вам допомогти. Для проектів, заснованих на фреймворках веб-сайтів, як описано в першому розділі, ви повинні використовувати налаштування, надані цими фреймворками, а не CRA.

Зваживши щойно перелічені плюси та мінуси, ми дійшли висновку, що CRA ідеально підходить для нових розробників. Ви отримуєте багато простоти та менше турбот. Отримавши більше досвіду, ви можете почати

експериментувати за межами CRA. Ось чому ми використали CRA для прикладів і проектів у цій книзі.

2.2 Примітка щодо прикладів у цій книзі

Як згадувалося, ми будемо використовувати CRA для всіх проектів і майже всіх прикладів у цій книзі. Єдиним винятком є перший приклад, який ви закінчили в першому розділі.

Усі шаблони, які ми створили для цієї книги, матимуть назви відповідно до цієї структури:

```
rqXX-NAME
```

Звичайно, rq означає React Quickly. XX буде замінено номером розділу, а останній біт буде спеціальною короткою назвою для кожного прикладу. Для кожного прикладу та проекту з використанням CRA ви побачите назву шаблону та способи його використання на бічній панелі, як показано нижче.

Repository: rq02-nesting

Цей приклад можна побачити в сховищі rq02-nesting. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

```
$ npx create-react-app rq02-nesting --template rq02-nesting
```

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip:

https://rq2e.com/rq02-nesting

Іноді приклади містять кілька варіантів вихідного коду, і в таких випадках кожен варіант матиме власний шаблон, як щойно показано. Є також приклади, які містять пропозиції щодо додаткових домашніх завдань. У таких випадках шаблон буде вказано як відправну точку для цього додаткового домашнього завдання, а інший шаблон міститиме одне можливе рішення. Ви можете використовувати шаблон рішення як натхнення або для порівняння зі своїм власним рішенням. Усі такі домашні завдання можуть мати нескінченну кількість рішень, тож те, що ваша робота не відповідає шаблону, це не означає, що вона неправильна — вона просто інша.

Для зручності використання назву шаблону також можна використовувати як назву програми. Отже, припустімо, ви хочете почати працювати над наступним прикладом у цій книзі. Ім'я шаблону — rq02-nesting, тому давайте також використаємо його як назву веб-програми:

```
$ npx create-react-app rq02-nesting --template rq02-nesting
```

Просто введіть це у своїй консолі, і ви вже запущені та готові взятися за приклад, щоб працювати над проблемою разом з нами, якщо ви того бажаєте. Ви також можете просто прочитати розділ і переглянути код у списках у книзі. Якщо вам здаються деякі речі дивними або вам потрібно вникнути в код, щоб спробувати щось, ви можете створити екземпляри шаблонів і побачити приклади в дії. Тепер давайте перейдемо до цього прикладу, який, здається, стосується вкладення чогось.

2.3 Елементи вкладеності

Повертаючись до створення додатків React, які ми збиралися робити в цій книзі, давайте почнемо робити речі трохи складнішими, ніж той повчальний, але надто спрощений приклад, який ми розглянули в розділі 1. У цьому розділі ви навчилися створювати один елемент React. Нагадуємо, що ви використовуєте метод React.createElement(). Наприклад, ви можете створити такий елемент посилання:

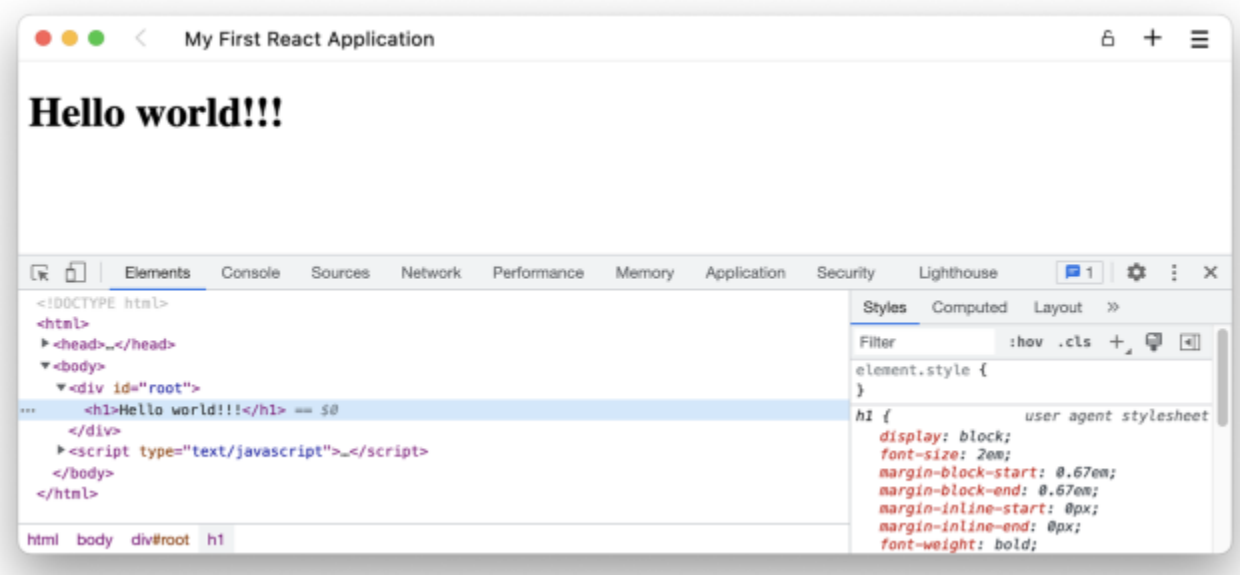
```
const reactLinkElement = React.createElement("a",
  { href: "http:/ /react.dev" },
  "React Website"
)
```

Це добре, якщо ми створюємо лише один елемент. Проблема полягає в тому, що кожен веб-сайт має більше ніж один елемент; інакше, як би ви мали будь-яку іншу інформацію, крім одного абзацу?

Рішення для створення багатоелементних структур в ієрархічній манері - це вкладені елементи. У попередньому розділі ви реалізували свій перший код React, створивши один елемент React і відобразивши його в DOM за допомогою ReactDOM.createRoot().render():

```
const title = React.createElement("h1", null, "Hello world!");
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(title);
```

Важливо зауважити, що ReactDOM.createRoot().render() може приймати лише один (кореневий) елемент React як аргумент, яким у цьому прикладі є reactElement. Отриманий додаток показано на малюнку 2.3.



Малюнок 2.3 Ваш браузер відображає один елемент заголовка. Тут ми відкрили інструменти розробника, щоб показати вам базову структуру HTML. Зверніться до свого браузера, щоб дізнатися, як відкрити інструменти розробника, але ймовірно, що Ctrl-Alt-I/Cmd-Opt-I може допомогти.

Repository: rq02-nesting

Цей приклад можна побачити в сховищі rq02-nesting. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

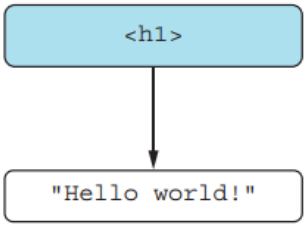
```
$ npx create-react-app rq02-nesting --template rq02-nesting
```

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip:

https://rq2e.com/rq02-nesting

Коли ви перевіряєте шаблон rq02-nesting, у вас буде попередня програма, але цього разу з використанням CRA замість ручного додавання бібліотек і написання HTML, як ми робили в розділі 1.

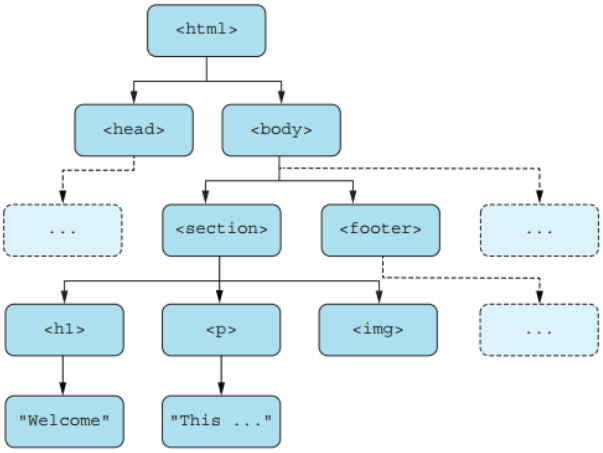
Пам’ятайте, що коли ви використовуєте createElement, третій аргумент є дочірнім елементом. У цьому випадку ми просто надаємо простий текст як дочірній елемент. Але цей текст насправді є іншим елементом — принаймні в отриманому DOM. У React він не має певного типу елемента, але все ще певною мірою функціонує як елемент. Ми можемо показати цю залежність на дуже простій діаграмі, як показано на малюнку 2.4.



Малюнок 2.4 Сірий вузол є справжнім елементом React, тоді як білий елемент є лише текстовим елементом.

2.3.1 Ієрархія вузлів

Перш ніж ми подивимося, як ми можемо створювати складні HTML-структури, нам спочатку потрібно трохи базової термінології. HTML-документ часто представляється як перевернуте дерево, як показано на малюнку 2.5. Вузли в дереві зазвичай описуються у вигляді сімейства (батьківський, дочірній тощо).



Малюнок 2.5 Перевернута деревовидна структура HTML-документа, де кожен вузол пов’язаний з іншими в сімейній ролі, наприклад, батьківський, дочірній і рідний.

Наступна термінологія стосується деревовидної структури:

 Node — будь-який член дерева є вузлом, включаючи елементи HTML і текстові вузли. Усі поля на малюнку 2.5 є вузлами. Два нижні поля є текстовими вузлами, а всі інші є вузлами елементів.

 Root — перший (верхній) вузол є коренем дерева. На малюнку 2.5 вузол <html> є кореневим вузлом.

 Parent — вузол, розташований безпосередньо над заданим вузлом, є його батьківським. Кожен вузол у дереві має лише одного батька. Вузол вище можна назвати дідусем і так далі. На малюнку 2.5 батьківським вузлом <body> є вузол <html>. Кореневий вузол не має батька, і це єдиний вузол у дереві без батька.

 Child — будь-який вузол, що знаходиться безпосередньо під заданим вузлом, є дочірнім для цього вузла. Вузол може мати кілька дочірніх елементів. Дочірніми елементами вузла <section> є вузли <h1>, <p> і <img>. Не всі вузли мають дітей. Елемент <img> не має дітей. Текстові вузли ніколи не мають дітей.

 Sibling — два вузли, які мають одного і того ж батьківського, вважаються однорідними вузлами. Вузол <p> має два вузли-сестри: вузли <h1> і <img>.

 Ascendants (Висхідні вузли) — батьківський вузол, його батьківський вузол, батьківський вузол його батьківського вузла і так далі — аж до кореня — називаються висхідними вузлами. Вузол <h1> має три висхідні: вузли <section>, <body> і <html>.

 Descendants (Нащадки) — нащадки вузла, усі їхні нащадки, усі нащадки їхніх дітей і так далі називаються нащадками вузла. Вузол <section> має п'ять нащадків: його три прямі дочірні елементи, а також два текстові вузли, які є онуками перших двох дочірніх елементів.

 Nesting — Вкладення — це процес організації вузлів у дереві та визначення того, які вузли будуть дочірніми для яких інших вузлів, таким чином створюючи дерево документів. На малюнку 2.5 ми вирішили вкласти вузли <h1>, <p> і <img> у вузол <section>.

2.3.2 Просте вкладення

Припустімо, ви хочете відобразити слово world курсивом у рядку «Hello world!» але все одно помістіть все це в елемент h1. Як показано на малюнку 2.6, ви створюєте елемент em із рядком «world» у якості дочірнього елемента та інший елемент h1 із трьома дочірніми елементами:

- Рядок «Hello» (зверніть увагу на пробіл у кінці)
- em елемент раніше
- Рядок "!"

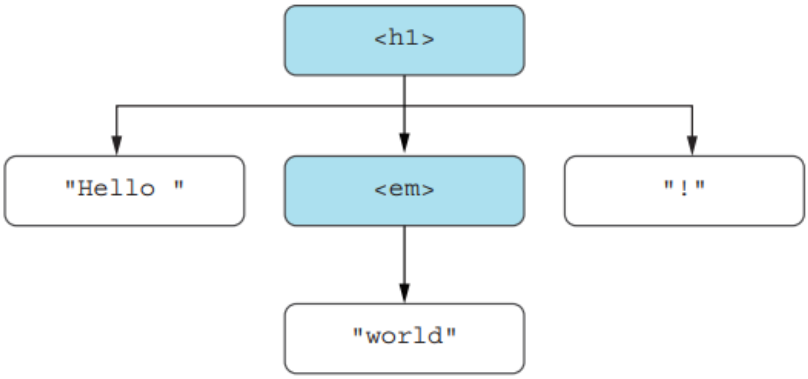


Рисунок 2.6 Два елементи React і три текстові елементи, необхідні для відтворення нашого злегка підкресленого вітального повідомлення

Використовуючи React.createElement, це стає наступним:

```
const world = React.createElement("em", null, "world");
  const title = React.createElement(
  "h1", null, "Hello ", world, "!"
)
```

Як ви бачите тут, зараз ми передаємо п’ять аргументів createElement: спочатку тип елемента, потім властивості і, нарешті, дочірні елементи елемента. Ви можете передати елементу скільки завгодно аргументів як дітей. Ви також можете передати дочірні елементи як масив:

```
const title = React.createElement("h1", null, ["Hello ", world, "!"])
```

У цьому випадку немає сенсу поміщати елементи в масив перед передачею їх як аргумент, але якщо у нас уже є масив елементів, ми могли б

просто передати його як аргумент сам по собі. Зібравши все це разом (без використання масиву), весь сценарій стає наступним лістингом.

Лістинг 2.1 Яскраве вітання світу

```
import React from "react";
import ReactDOM from "react-dom/client";
const world = React.createElement("em", null, "world");
const title = React.createElement("h1", null, "Hello ", world, "!");
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(title);
```

Якщо ми застосуємо це в дію, наша програма буде виглядати у браузері так, як показано на малюнку 2.7.

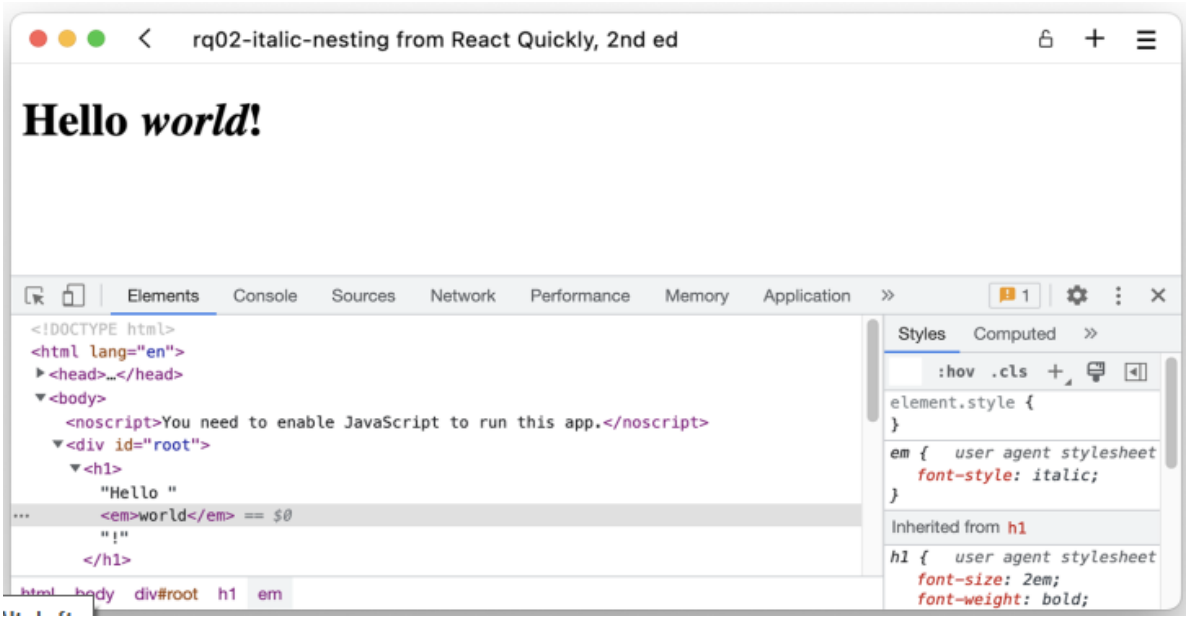


Рисунок 2.7 Виділене привітання в браузері. Зверніть увагу на базову структуру HTML в інструментах розробника.

Репозиторій: rq02-nesting-italic

Цей приклад можна побачити в сховищі rq02-nesting-italic. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

$ npx create-react-app rq02-nesting-italic --template rq02-nesting-italic

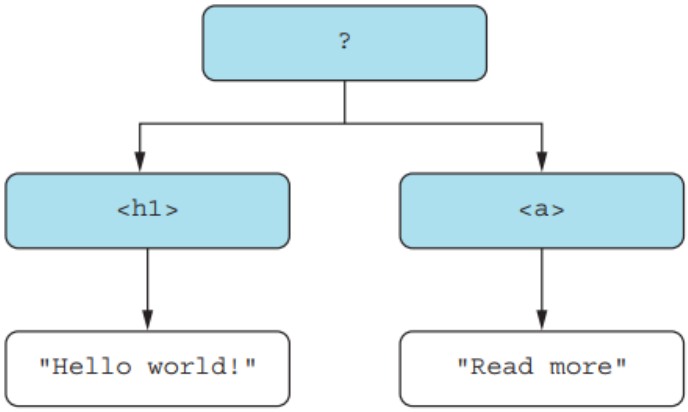
Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip: https://rq2e.com/rq02-nesting-italic .

Але що, якщо ви хочете розмістити елемент після h1, а не просто всередині нього? Ми розглянемо однотипні елементи в наступному розділі.

2.3.3 Брати та сестри

У багатьох випадках ви можете використовувати лише один елемент React на верхньому рівні. Це стосується методу ReactDOM.createRoot().render() — лише один елемент може бути відтворений у DOM як кореневий елемент. Трохи пізніше ви також побачите, як спеціальні компоненти можуть повертати лише один елемент.

Але що, якщо ви хочете показати заголовок, а потім посилання після нього в нашому попередньому прикладі (див. рис. 2.8)? Це були б два різні елементи поруч один з одним, і ви не можете відобразити це безпосередньо за допомогою ReactDOM.createRoot().render().



Малюнок 2.8 Два однотипних React-елементи, які відображаються в корені

Замість цього ви повинні загорнути їх в інший елемент (щось замість ? на малюнку 2.8). У вас є два різні варіанти. Одним із варіантів є використання нейтрального елемента DOM, що легко, але додає «фізичний» елемент до вихідного HTML. Альтернативою є використання елемента React Fragment, який працює як будь-який інший елемент, але сам по собі не призводить до вихідного HTML. Дивіться різницю між цими підходами на малюнку 2.9.

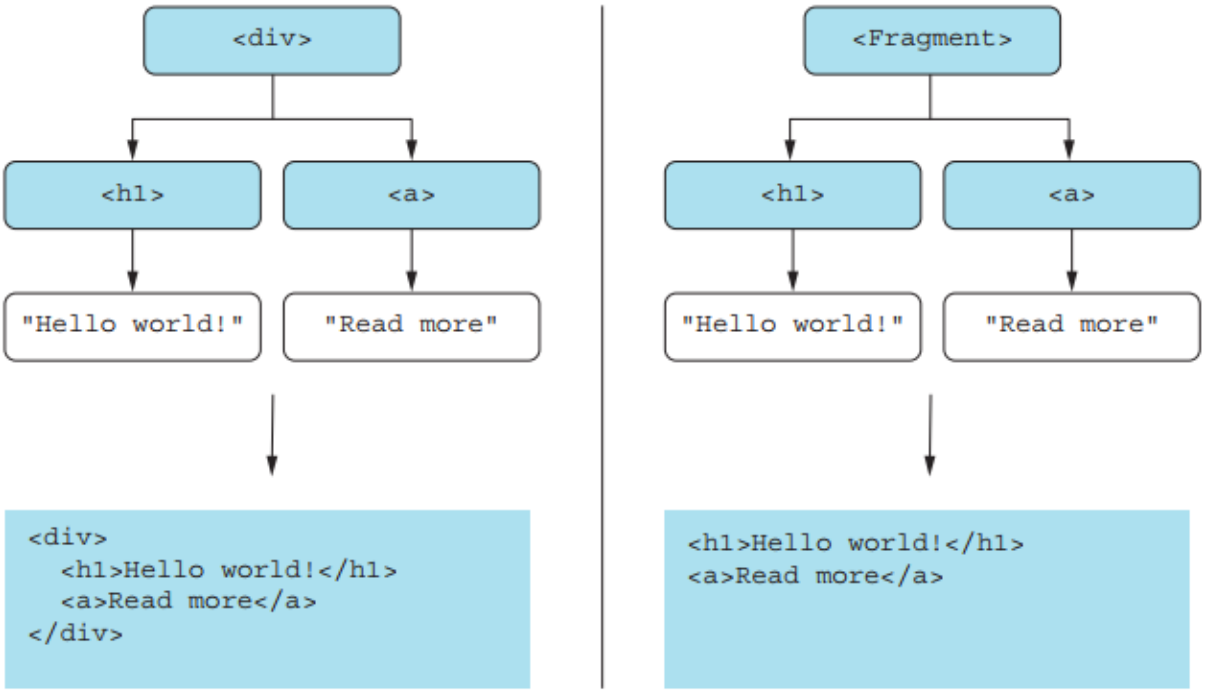


Рисунок 2.9 Два різних підходи до однорідних елементів з різними результатами

Якщо ви хочете використовувати нейтральний елемент DOM, ви можете, наприклад, використати , щоб згрупувати їх, як показано в наступному списку. Це призводить до HTML, який ви бачите на малюнку 2.10.

Лістинг 2.2 Два елементи в контейнері групування

```
import React from "react";
import ReactDOM from "react-dom/client";
const title = React.createElement("h1", null, "Hello world!");
const link = React.createElement("a", { href: "//react.dev" }, "Read more");
const group = React.createElement("div", null, title, link);
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(group);
```

Рисунок 2.10 Заголовок і посилання в елементі групування

Репозиторій: rq02-siblings-div

Цей приклад можна побачити в сховищі rq02-siblings-div. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

$ npx create-react-app rq02-siblings-div --template rq02-siblings-div

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip: https://rq2e.com/rq02-siblings-div

Контейнер зазвичай є хорошим вибором для вмісту на рівні блоку, а використовується для вмісту вбудованого рівня. Але вам не обов'язково використовувати «справжній» елемент. Ви також можете створити порожній елемент React, єдиною метою якого є групування кількох інших елементів і не виводити себе в HTML на сторінці. Це можна зробити за допомогою магічного компонента під назвою React.Fragment, і його можна використовувати як тип елемента групування. Давайте зробимо це в наступному лістингу.

Лістинг 2.3 Два елементи у фрагменті

```
import React from "react";
import ReactDOM from "react-dom/client";
const title = React.createElement("h1", null, "Hello world!");
const link = React.createElement("a", { href: "//react.dev" }, "Read more");
const group = React.createElement(
React.Fragment, null, title, link
);
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(group);
```

Репозиторій: rq02-siblings-fragment

Цей приклад можна побачити в сховищі rq02-siblings-fragment. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

$ npx create-react-app rq02-siblings-frag --template rq02-siblings-fragment

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код у форматі zip-файлу: https://rq2e.com/rq02-siblings-fragment

Вихідні дані цього показані на малюнку 2.11 у браузері.

Ви також можете відобразити весь елемент в одному операторі наступним чином:

```
const group = React.createElement(
 React.Fragment,
 null,
 React.createElement(
  "h1",
  null,
  "Hello world!",
 ),
 React.createElement(
  "a",
  { href: "//react.dev" },
  "Read more",
 ),
);
```

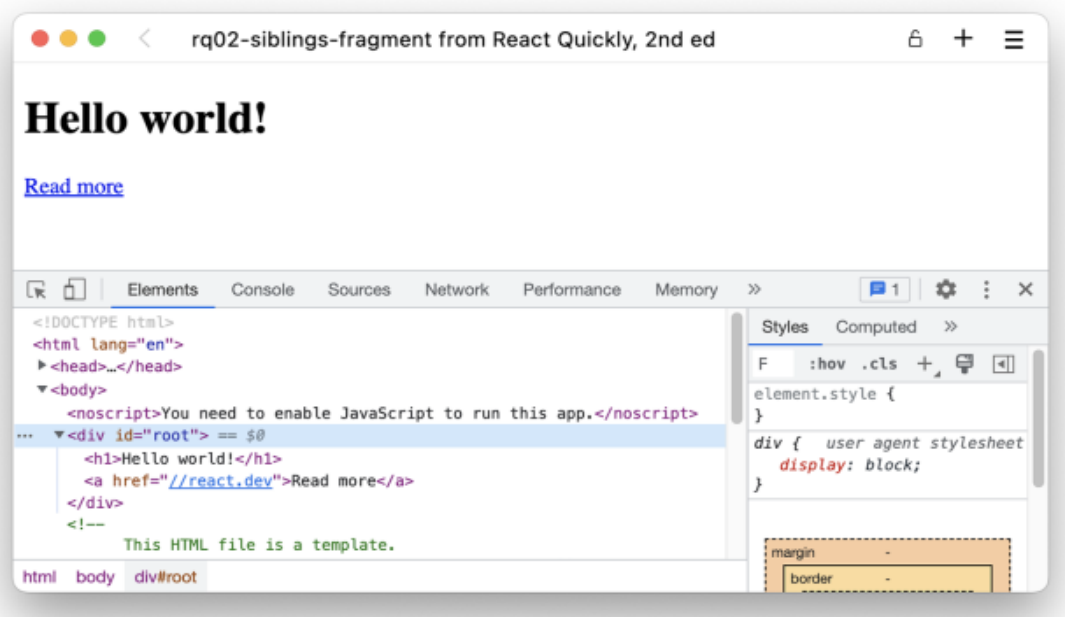


Рисунок 2.11 Заголовок і посилання без елемента групування

Це функціонально еквівалентно попередньому коду; він просто використовує менше змінних. Деякі стверджують, що це стає більш очевидним, тоді як інші скажуть, що це стає менш читабельним.

Поки що ви переважно надавали рядкові значення як перший параметр createElement(). Але перший параметр може мати два типи введення, як ми щойно бачили з фрагментами:

 Стандартний тег HTML у вигляді рядка, наприклад, "h1", "div" або "p" (без кутових дужок). Ім'я пишеться з малої літери.

 Компонент React як посилання (а не рядок). Ім'я зазвичай пишеться з великої літери.

Перший підхід відображає стандартні елементи HTML. Ви можете використовувати будь-який рядок як назву тегу HTML, незалежно від того, чи має він значення у браузері за замовчуванням. Отже, хоча ви переважно використовуєте звичайні елементи HTML, такі як div, main, section тощо, ніщо не заважає вам створити елемент tiny-horse, який відтворюватиметься як <tiny-horse> у браузері . Це не має значення та стилю за замовчуванням, але це буде працювати.

У другому щойно перерахованому підході ми можемо надати компонент React як еталон. Під цим ми маємо на увазі не назву компонента React як рядок,

а пряме посилання на відповідний компонент. Ви вже бачили один випадок цього, використовуючи React.Fragment. Тепер давайте подивимося, як ми можемо створити власні користувацькі компоненти в наступному розділі.

2.4 Створення нестандартних (custom) компонентів

Після вкладення елементів за допомогою React ви незабаром натрапите на наступну проблему: є багато елементів, які часто повторюються. Вам потрібно використовувати архітектуру, засновану на компонентах (CBA), описану в розділі 1, яка дозволяє повторно використовувати код, розділяючи функціональні можливості на слабко пов'язані частини: познайомтеся з класами компонентів або просто компонентами, як їх часто називають для стислості (не плутати з веб-компонентами).

Розглядайте стандартні теги HTML як будівельні блоки. Ви можете використовувати їх для створення власних компонентів React, які можна використовувати для створення користувацьких елементів (примірників компонентів). Використовуючи користувальницькі елементи, ви можете інкапсулювати й абстрагувати логіку в компоненти, які можна багаторазово використовувати. Ця абстракція дозволяє командам повторно використовувати інтерфейси користувача (UI) у великих складних програмах, а також у різних проектах. Приклади включають панелі, входи, кнопки, меню тощо.

Для цього прикладу ми хочемо створити три ідентичні посилання. Немає сенсу створювати ідентичні посилання, але наразі ми не можемо їх налаштувати, тож розглянемо цей сценарій. Ми хочемо створити три посилання, у кожному з яких буде написано «Докладніше про React» і посилання на веб-сайт React за адресою www.react.dev . Ми також хочемо обернути кожне посилання в абзац, щоб вони розташовувалися в окремих рядках.

Є два різних підходи до цього. Ми можемо зробити це простим способом, маючи три ідентичні копії елементів, або ми можемо зробити це розумним шляхом, створивши багаторазово використовуваний компонент посилання, а потім створивши його тричі, як показано на малюнку 2.12.

Давайте спочатку розглянемо попередній підхід, коли ми використовуємо лише один компонент із копіями, створеними вручну. Нам

потрібні три незалежні посилання всередині незалежних абзаців, і ми можемо зробити це досить докладно, як у наступному лістингу.

Лістинг 2.4 Три посилання, по одному разу кожне

```
import React from "react";
import ReactDOM from "react-dom/client";
const link1 = React.createElement(
  "a", { href: "//react.dev" }, "Read more about React"
);
const p1 = React.createElement("p", null, link1);
const link2 = React.createElement(
  "a", { href: "//react.dev" }, "Read more about React"
);
const p2 = React.createElement("p", null, link2);
const link3 = React.createElement(
  "a", { href: "//react.dev" }, "Read more about React"
);
const p3 = React.createElement("p", null, link3);
const group = React.createElement(React.Fragment, null, p1, p2, p3);
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(group);
```

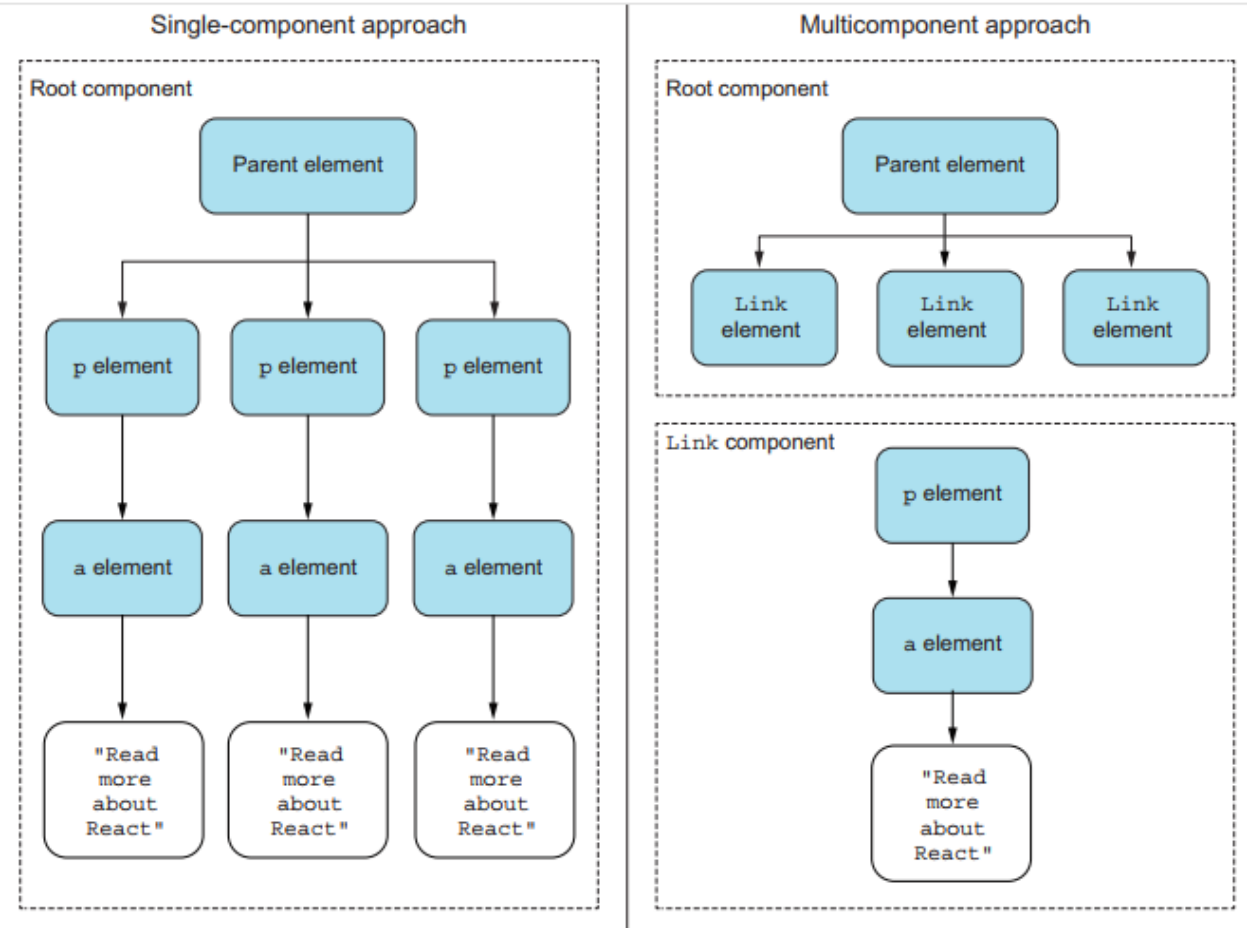


Рисунок 2.12 Два підходи до створення повторюваних елементів

Якщо ми відкриємо це в браузері, ми отримаємо результат, показаний на малюнку 2.13, і це саме те, що ми хотіли.

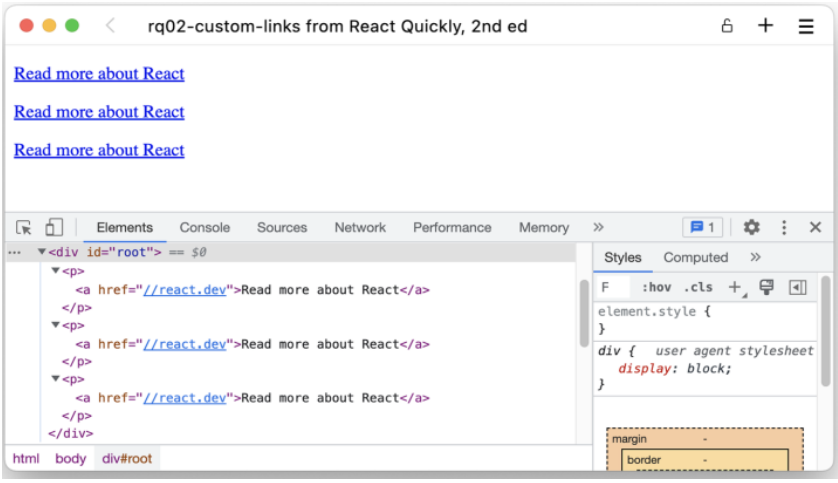


Рисунок 2.13 Три однакових посилання в нашому додатку

Але ми часто повторюємося в лістингу 2.4, що, звичайно, не бажано. Суть React і подібних фреймворків полягає в тому, щоб взагалі припинити повторюватися. Це вимагає спеціального компонента!

Спеціальний (custom) компонент — це іменований об’єкт, який містить інші елементи та екземпляри компонентів. Отже, у цьому випадку ми могли б створити єдиний компонент Link, який відображав би потрібне нам посилання правильним чином, і тоді ми б включили три екземпляри компонента Link, а не «необроблені» <p> і <a> елементи з усіма їхніми властивостями.

Ви створюєте клас компонента React, розширюючи клас React.Component за допомогою класу CHILD розширює синтаксис PARENT ES6. Давайте створимо спеціальний клас компонента Link за допомогою класу Link extends React.Component.

Єдиною обов’язковою річчю, яку ви повинні реалізувати для цього нового класу, є метод render(). Цей метод має повертати єдиний кореневий елемент, створений за допомогою createElement(), який створюється з іншого класу спеціального компонента або тегу HTML. Будь-який може мати вкладені елементи, якщо ви цього бажаєте, якщо є лише один кореневий елемент.

Лістинг 2.5 Створення та рендеринг класу компонентів React

```
import React from "react";
import ReactDOM from "react-dom/client";
class Link extends React.Component {
  render() {
    return React.createElement(
      "p",
      null,
      React.createElement(
        "a",
        { href: "//react.dev" },
        "Read more about React"
      )
    );
  }
}
const link1 = React.createElement(Link);
const link2 = React.createElement(Link);
const link3 = React.createElement(Link);
const group = React.createElement(
  React.Fragment, null, link1, link2, link3
);
```

```
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(group);
```

Репозиторій: rq02-custom-links

Цей приклад можна побачити в репозиторії rq02-custom-links. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

$ npx create-react-app rq02-custom-links --template rq02-custom-links

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip: https://rq2e.com/rq02-custom-links

За домовленістю назви змінних, що містять компоненти React, пишуться з великої літери. У звичайному JavaScript це не потрібно. Ви можете використовувати назву класу someLink у нижньому регістрі в попередньому коді замість Link, і це все одно працюватиме. Але оскільки це необхідно для JSX (про що ми розглянемо в наступному розділі), ми також застосовуємо цю угоду тут.

Подібно до ReactDOM.createRoot().render(), метод render() у компоненті класу може повертати лише один елемент. Якщо вам потрібно повернути кілька елементів одного рівня, оберніть їх у компонент-контейнер — елемент HTML або фрагмент React. Якщо ми зараз запустимо цей код у браузері, ми отримаємо той самий HTML, що й раніше (див. малюнок 2.13).

Цей новий код набагато компактніший. Непотрібні повторення видалено, і ми розділили частину коду, яку можна повторно використовувати скільки завгодно. Це сила багаторазового використання компонентів! Це веде до швидшого розвитку та меншої кількості помилок. Компоненти також мають властивості, події життєвого циклу, стани, події DOM та інші функції, які дозволяють зробити їх інтерактивними та самодостатніми; усі ці теми розглядаються в наступних розділах.

Зараз посилання однакові. Чи не було б чудово, якби ви могли встановлювати атрибути елементів і змінювати їх вміст і/або поведінку окремо? Ви можете зробити саме це за допомогою властивостей, як ми обговоримо далі.

2.5 Робота з властивостями

Властивості є наріжним каменем декларативного стилю, який використовує React. Думайте про властивості як про незмінні значення в елементі. Вони дозволяють елементам мати різні варіації, якщо вони використовуються в представленні, наприклад змінювати URL-адресу посилання шляхом передачі нового значення для властивості:

React.createElement("a", { href: "//react.dev" }, "React");

Слід пам’ятати, що властивості є незмінними в межах своїх компонентів. Батьківський елемент призначає властивості дочірнім елементам під час їх створення. Дочірній елемент не повинен змінювати свої властивості. Наприклад, ви можете передати властивість PROPERTY_NAME зі значенням VALUE компоненту типу Link, ось так:

React.createElement(Link, { PROPERTY_NAME: VALUE});

Властивості дуже схожі на атрибути HTML (як показано за допомогою href у посиланні фрагмента на початку цього розділу). Це одна з їхніх цілей, але вони також мають іншу — ви можете використовувати властивості елемента у своєму коді як завгодно для наступного:

 Щоб відобразити стандартні атрибути HTML елемента: href, title, style, class тощо

 Як настроювані інструкції для компонентів, щоб зробити їх рендеринг окремо

Доступ до об’єкта властивостей можна отримати всередині компонента за допомогою this.props. Цей об’єкт є замороженим (незмінним) об’єктом, з якого можна лише зчитувати значення, а не встановлювати їх.

Заморожені об'єкти в JavaScript

Всередині React використовує Object.freeze(), яка є вбудованою функцією в JavaScript, щоб зробити об’єкт this.props незмінним. Щоб перевірити, чи заморожений об’єкт, можна скористатися методом Object.isFrozen(). Наприклад, ви можете визначити, чи повертатиме цей оператор true:

```
class Test extends React.Component {
   render() {
```

```
        console.log(Object.isFrozen(this.props))
        return React.createElement("div")
    }
}
```

Деталі цього досить складні, але наразі просто знайте, що ви ніколи не намагаєтеся редагувати чи додавати властивості всередині самого компонента. Це те, що ви робите в батьківському контексті.

2.5.1 Єдина властивість

Почнемо з дуже простого прикладу. Ми хочемо, щоб назва фреймворку в посиланнях, які ми створили раніше, була налаштована. Отже, ми можемо сказати «Дізнайтеся більше про React» в одному посиланні, «Дізнайтеся більше про Vue» у другому та «Дізнайтеся більше про Angular» у третьому, як показано на малюнку 2.14.

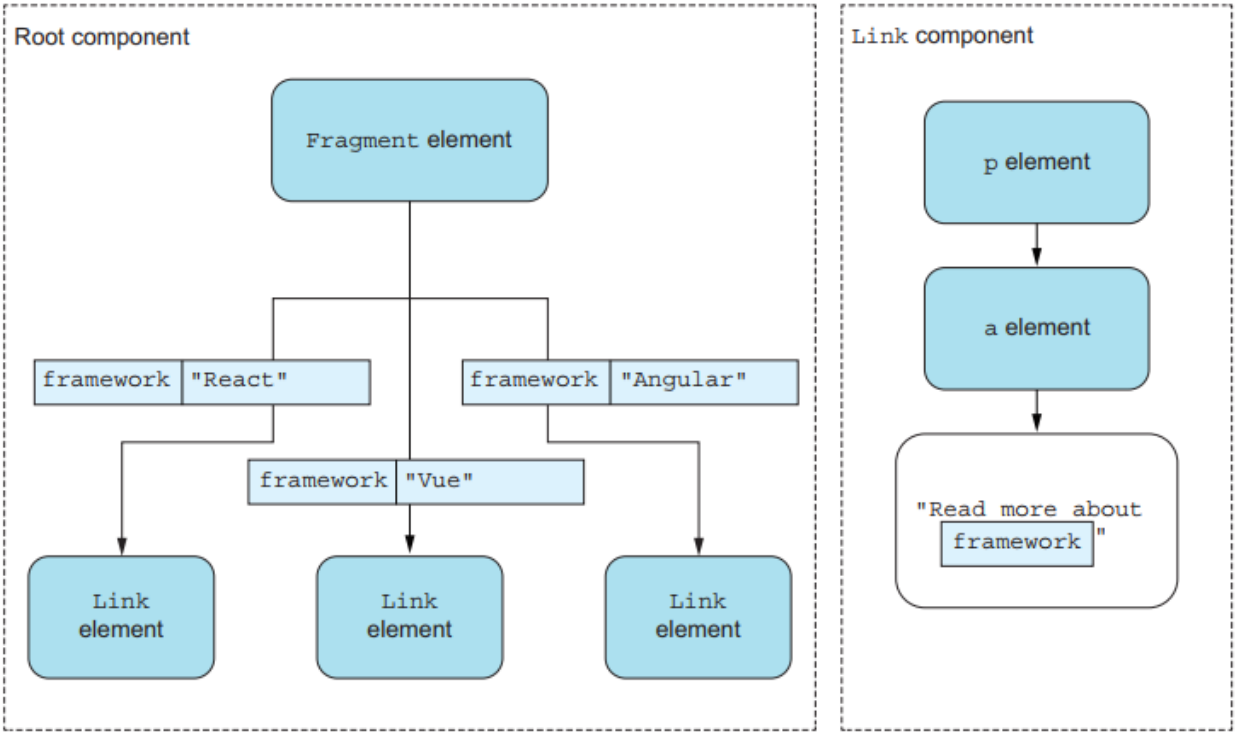


Рисунок 2.14 Передача властивості компонентам і використання властивості всередині компонента

Для цього нам потрібно зробити дві речі:

1 Передати властивість екземплярам наших компонентів.

2 Використати властивість всередині компонента.

По-перше, нам потрібно передати нову властивість екземплярам посилання. Отже, замість того щоб просто використовувати

const link1 = React.createElement(Link);

ми будемо надавати об’єкт як другий аргумент з єдиною властивістю:

const link1 = React.createElement(Link, { framework: "React" });

Тут ми використали структуру імен змінних. Це довільний вибір, який ми можемо зробити як творці компонента. Нам просто потрібно переконатися, що на другому кроці використовується те саме ім’я змінної.

Тепер нам потрібно використати цю передану властивість у нашому класі. Враховуючи те, що ми викликали змінну структуру, ми отримаємо доступ до неї через this.props.framework. У наступному лістингу показано загальний результат коду.

Лістинг 2.6 Посилання на екземпляри з іншим текстом

```
import React from "react";
import ReactDOM from "react-dom/client";
class Link extends React.Component {
  render() {
    return React.createElement(
      "p",
      null,
      React.createElement(
        "a",
        { href: "//react.dev" },
        `Read more about ${this.props.framework}`,
       ),
      );
    }

}
const link1 = React.createElement(Link, {
  framework: "React"
});
const link2 = React.createElement(Link, {
  framework: "Vue"
});
```

```
const link3 = React.createElement(Link, {
  framework: "Angular"
});
const group = React.createElement(
  React.Fragment, null, link1, link2, link3
);
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(group);
```

Ви можете побачити це в дії у браузері на малюнку 2.15.

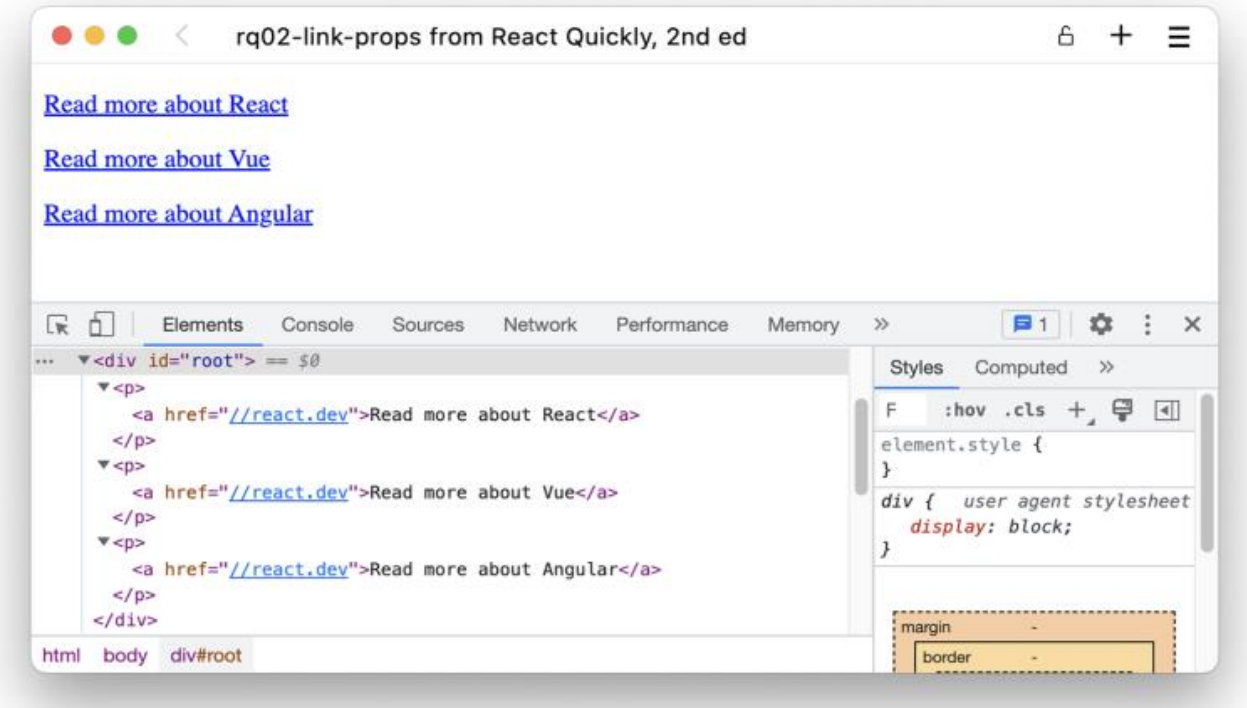


Рисунок 2.15 Три посилання з різним текстом, кожне всередині абзацу

2.5.2 Кілька властивостей

Можливо, ви помітили, що всі посилання все ще вказують на ту саму URL-адресу, яка є веб-сайтом React. Звичайно, це недобре, тому що нам потрібні інші URL-адреси. Використовуючи той самий підхід, ми просто винаходимо нову властивість, url, і використовуємо її всередині компонента, а також у екземплярах компонента. Ви можете бачити, що це показано на діаграмі на малюнку 2.16 і реалізовано в коді в лістингу 2.7.

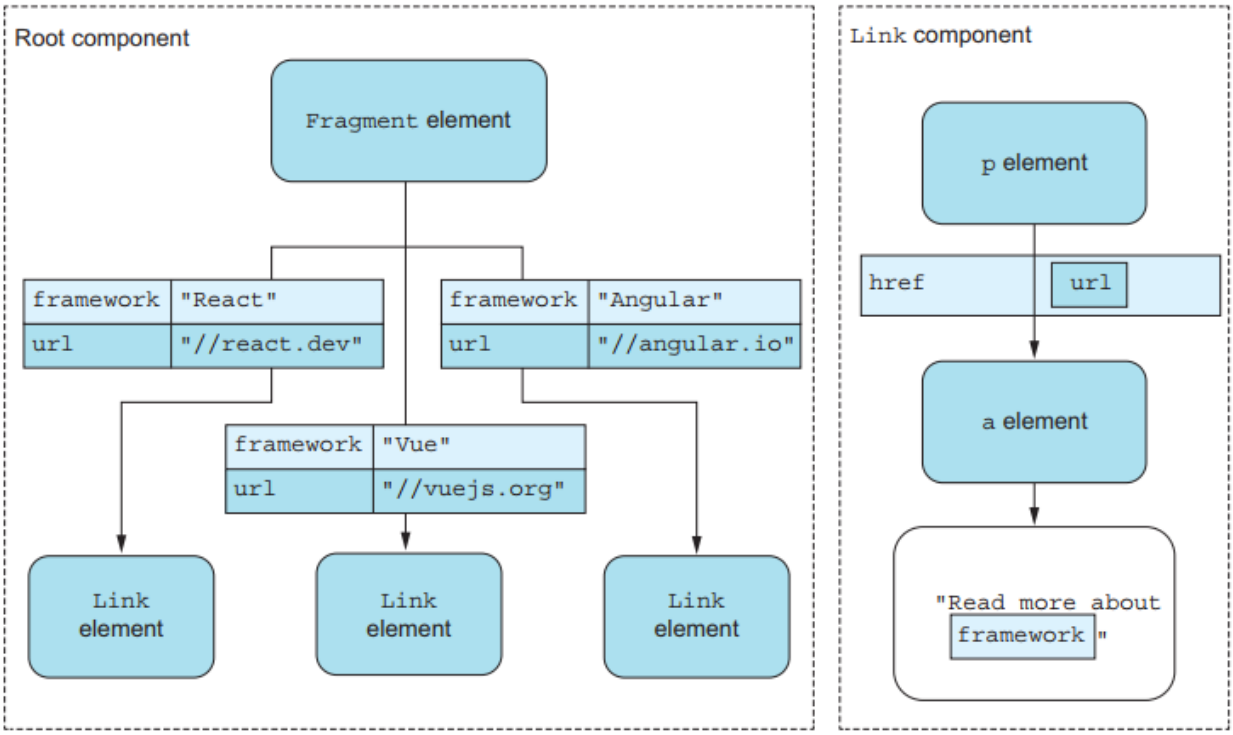


Рисунок 2.16 Дві різні властивості передаються нашим компонентам.

Лістинг 2.7 Посилання на екземпляри з іншим текстом і URL-адресами

```
import React from "react";
import ReactDOM from "react-dom/client";
class Link extends React.Component {
      render() {
        const link = React.createElement(
            "a",
            { href: this.props.url },
            `Read more about ${this.props.framework}`
        );
        return React.createElement("p", null, link);
      }
}
 const link1 = React.createElement(Link, {
     framework: "React",
     url: "//react.dev",
 });
 const link2 =
 React.createElement(Link, {
     framework: "Vue",
     url: "//vuejs.org",
 });
 const link3 =
 React.createElement(Link, {
     framework: "Angular",
     url: "//angular.io",
```

```
});
const group = React.createElement(
   React.Fragment, null, link1, link2, link3
);

const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(group);
```

Репозиторій: rq02-link-props

Цей приклад можна побачити в репозиторії rq02-link-props. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

$ npx create-react-app rq02-link-props --template rq02-link-props

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip: https://rq2e.com/rq02-link-props

Ви можете побачити це в дії у браузері на малюнку 2.17. Як бачите, ми можемо використовувати властивості як користувацьких компонентів (які використовуються всередині компонента для налаштування повернутої структури), так і елементів HTML (які встановлюють атрибути HTML).

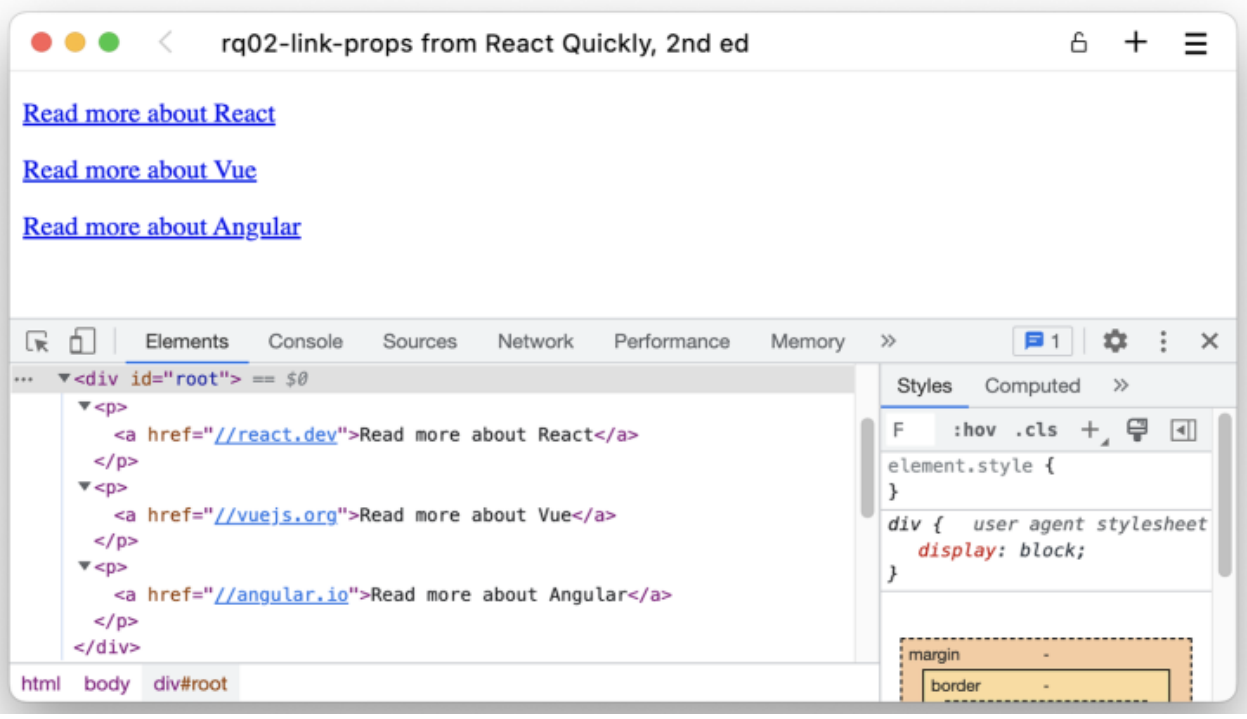


Рисунок 2.17 Три посилання з різним текстом і різними URL-адресами

Що станеться, якщо ви зіпсуєтеся й установите спеціальну властивість для елемента HTML? React все одно відтворить її. До React 16 недійсні властивості відфільтровувалися, але оскільки сучасні веб-додатки часто використовують бібліотеки сторонніх розробників, які можуть покладатися на деякі власні властивості, React 16 і новіші версії дозволять вам використовувати ті властивості, які ви виберете.

Ви можете повністю змінити відтворені елементи на основі значення властивості. Наприклад, ми можемо перевірити властивість framework та повернути величезний заголовок із посиланням у випадку, якщо ім'я фреймворку «React»:

```
class Link extends React.Component {
  render() {
    const link = React.createElement(
      "a",
      { href: this.props.url },
      `Read more about ${this.props.framework}`,
    );
    if (this.props.framework === "React") {
      return React.createElement("h1", null, link);
    }
    return React.createElement("p", null, link);
  }
}
```

Це також чудовий приклад того, що елементи React є звичайним старим JavaScript. Ми можемо створити елемент і зберегти його в змінній, а потім використовувати цю змінну на свій розсуд. Ми також можемо створювати розгалуження за допомогою звичайної функції JavaScript. Якщо ми відтворимо цей новий компонент у браузері, посилання раптом перестануть бути ідентичними, як ви бачите на малюнку 2.18.

Зараз ми розглянули кілька перестановок дуже простого HTML, який сам по собі майже марний. Але починаючи з малого, ми будуємо міцну основу для майбутнього, більш складні теми. Правда полягає в тому, що за допомогою спеціальних компонентів можна досягти багатьох чудових результатів.

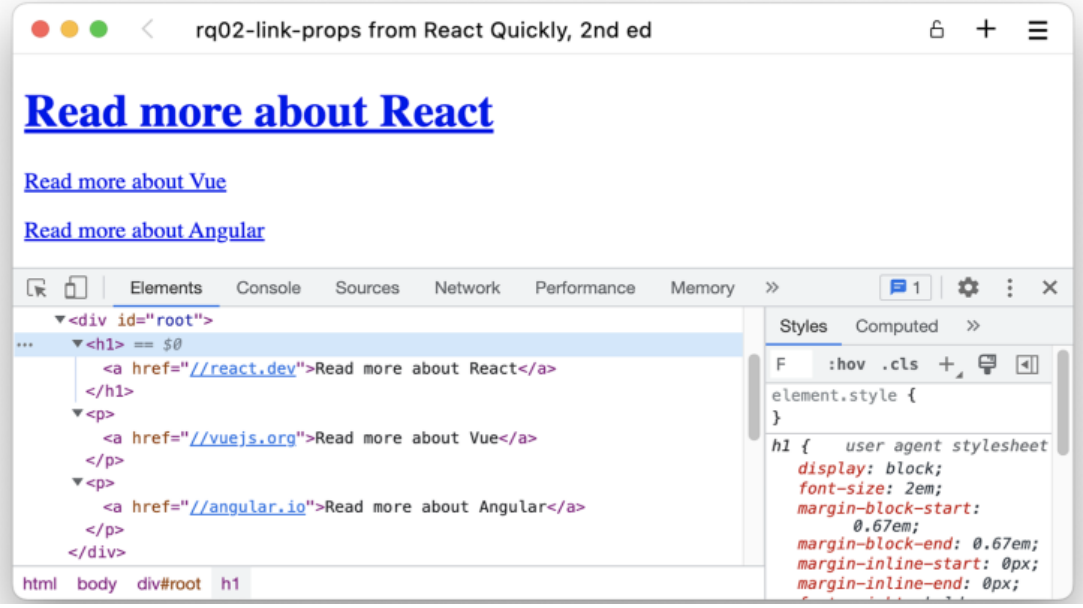
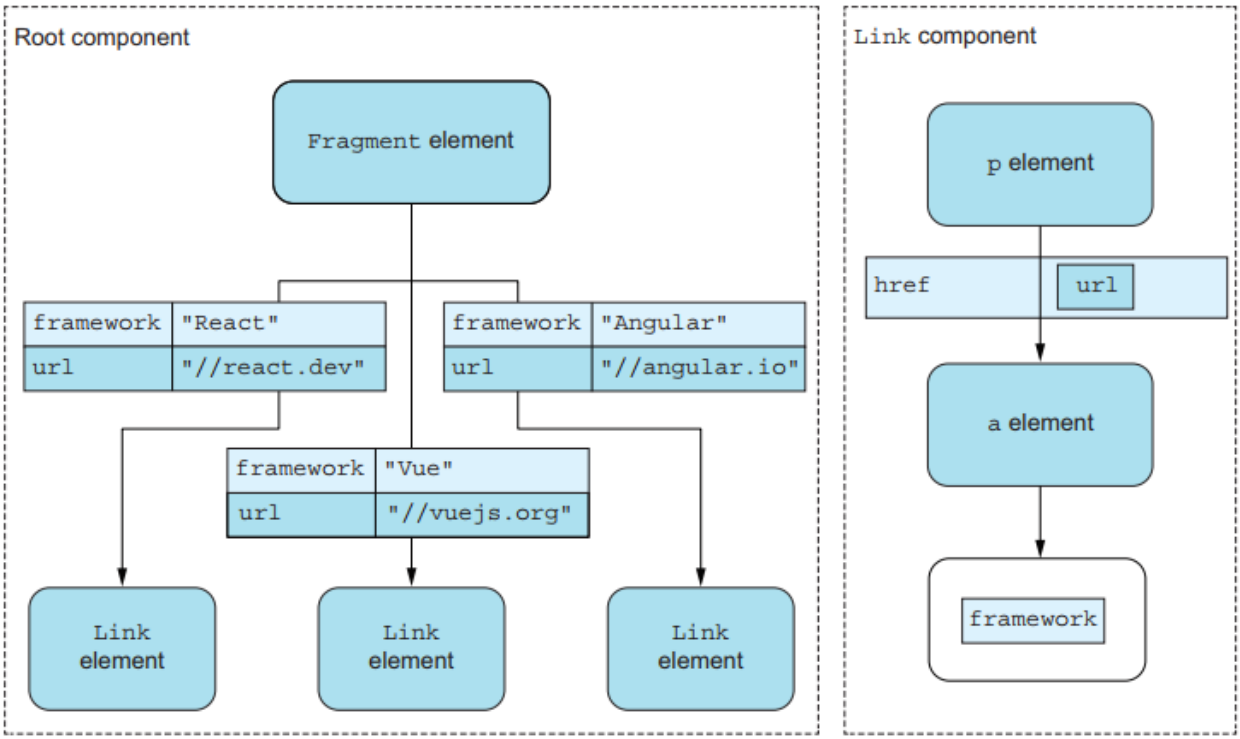


Рисунок 2.18 Показано три посилання, але React виділяється як набагато важливіший.

Дуже важливо знати, як React працює у звичайному JavaScript, якщо ви (як і багато розробників React) плануєте використовувати JSX. Це пояснюється тим, що врешті-решт браузери все одно запускатимуть звичайний JavaScript, і час від часу вам потрібно буде розуміти результати транспіляції JSX-JS. Надалі ми будемо використовувати JSX, який описано в наступному розділі. Але перш ніж перейти до цього, нам потрібно трохи обговорити структуру програми React.

2.5.3 Особлива властивість: children

Елементи React мають особливі властивості, children. Це не властивість, яку ви вказуєте звичайним способом, але ви використовуєте її як будь-яку іншу властивість. Давайте трохи змінимо наш приклад і натомість створимо список посилань, де текст є лише назвою фреймворку без тексту «Докладніше про» перед ним, як показано на малюнку 2.19.



Малюнок 2.19 Наша нова структура з посиланнями, які містять лише назву фреймворку

Тепер зробимо ще один крок далі. Скажімо, ми хочемо, щоб фреймворк для React відображався жирним шрифтом. Ми вже знаємо, як зробити елемент жирним шрифтом — просто оберніть його в елемент таким чином:

React.createElement("strong", null, "React");

Але як ми передамо це як власність? Ми можемо зробити це, створивши вузол для фреймворку React так:

```
const boldReact = React.createElement("strong", null, "React");
const link1 = React.createElement(
    Link,
    { framework: boldReact, url: "//react.dev" }
);
```

Хоча це трохи дивно. Зараз ми створюємо елементи, але передаємо їх як властивості, що зазвичай не робимо. Що, якби замість цього ми могли створити елемент і передати його як дочірній елемент?

Пам’ятаєте, як аргумент третій і далі до React.createElement є дочірніми елементами? Ми не використовували це для спеціальних компонентів, але можемо. Усі вузли, передані як дочірні елементи спеціального елемента,

доступні через this.props.children. Ця властивість є або одним вузлом (якщо передано лише один дочірній елемент), або масивом вузлів (якщо передано кілька дочірніх елементів).

Отже, давайте змінимо наш кореневий компонент, щоб він містив три посилання, де текст посилання передається не як властивість з назвою framework, а як дочірній вузол. Для першого посилання ми все ще хочемо зробити текст жирним, як показано на малюнку 2.20, а потім реалізовано в лістингу 2.8.

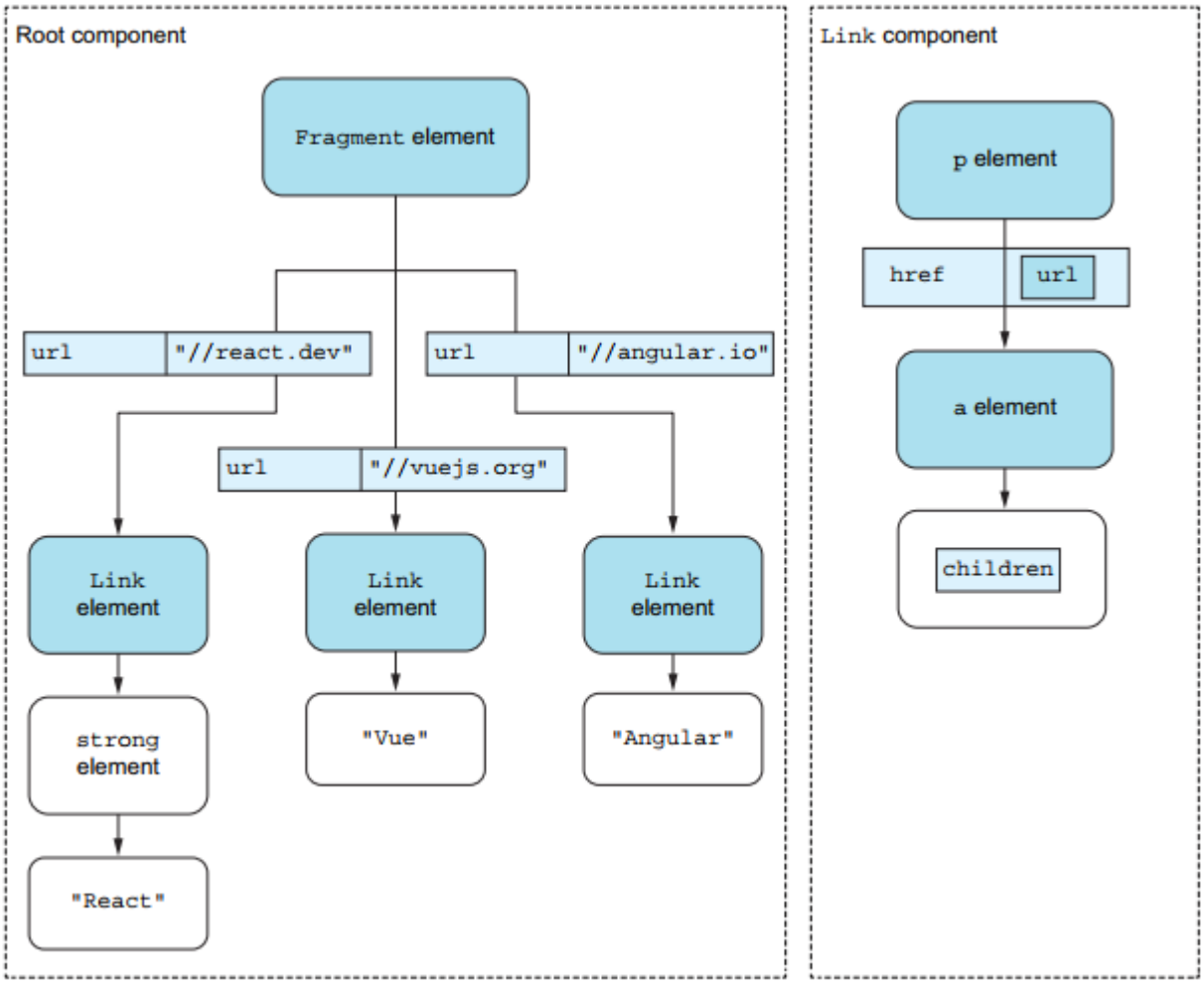


Рисунок 2.20 Дерево компонентів, коли ми передаємо текст посилання як дочірній вузол, а не як звичайну властивість

Лістинг 2.8 Посилання з текстом як дочірні вузли

```
import React from "react";
import ReactDOM from "react-dom/client";
```

```
class Link extends React.Component {
    render() {
        return React.createElement(
          "p",
          null,
          React.createElement(
            "a",
            { href: this.props.url },
            this.props.children
          )
        );
    }
}
const boldReact = React.createElement("strong", null, "React");
const link1 = React.createElement(
    Link,
    { url: "//react.dev" },
    boldReact
);
const link2 = React.createElement(
    Link,
    { url: "//vuejs.org" },
    "Vue"
);
const link3 = React.createElement(
    Link,
    { url: "//angular.io" },
    "Angular"
);
const group = React.createElement(
    React.Fragment, null, link1, link2, link3
);
const domElement = document.getElementById("root");
ReactDOM.createRoot(domElement).render(group);
```

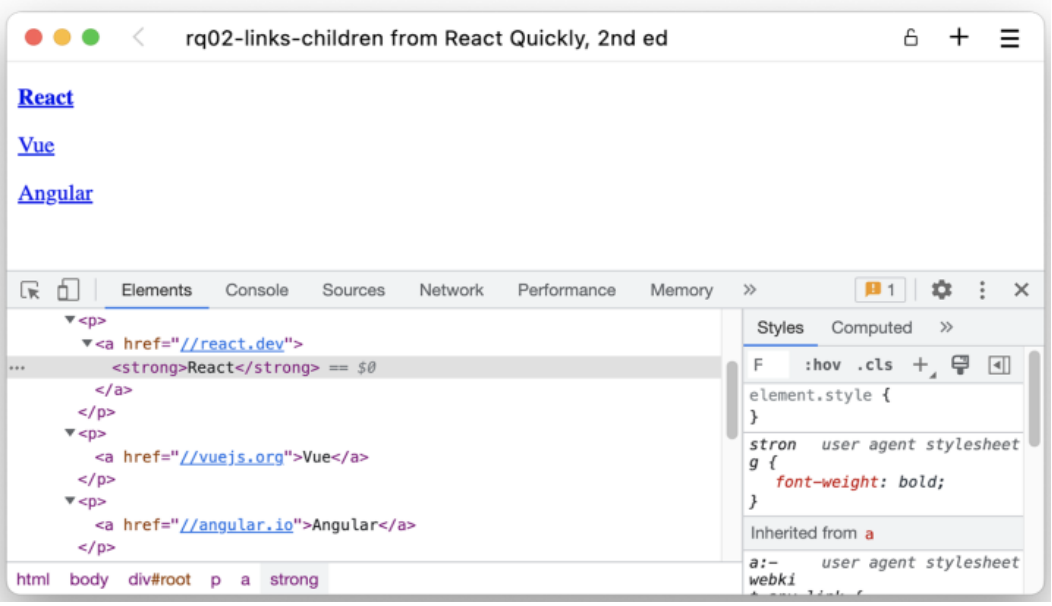
Репозиторій: rq02-links-children

Цей приклад можна побачити в репозиторії rq02-links-children. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

$ npx create-react-app rq02-links-children --template rq02-links-children

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip: https://rq2e.com/rq02-links-children

Якщо ви запустите це в браузері, ви отримаєте результат, показаний на малюнку 2.21. Різниця між використанням звичайної властивості та дочірньої властивості на даний момент може здатися несуттєвою, але в наступному розділі, коли ми почнемо використовувати JSX, ви побачите, як це починає мати великий сенс при правильному використанні.



Малюнок 2.21. Наші компоненти посилання використовують дочірні властивості, де перше посилання виділено жирним шрифтом

2.6 Структура програми

Починаючи з наступного розділу, ми збираємося структурувати наші програми таким же організованим способом зі схожими шаблонами для легкого розпізнавання. Ми також дотримуватимемося стандартної структури, яку надає шаблон CRA за замовчуванням.

У попередніх прикладах у цій главі ми розмістили нашу програму безпосередньо у файлі index.js у вихідній папці. Відтепер ми використовуватимемо custom App component (спеціальний компонент додатка) як кореневий елемент наших додатків і відтворюватимемо його як єдиний дочірній компонент у браузері. Це означає, що нам взагалі не

доведеться знову торкатися src/index.js. Він залишатиметься тим самим файлом для всіх майбутніх програм, які використовують CRA.

З цією метою ми перепишемо нашу програму з трьома посиланнями, що були раніше, у вигляді двох нових компонентів. Один — це коренева програма (root App), а інший — компонент Link. Ми використаємо останнє три рази в першому. Нарешті, ми деструктуруємо деякі властивості з простору імен React, щоб трохи скоротити визначення нашого компонента. Все це ми розмістимо в src/App.js. Дивіться малюнок 2.22 і лістинг 2.9.

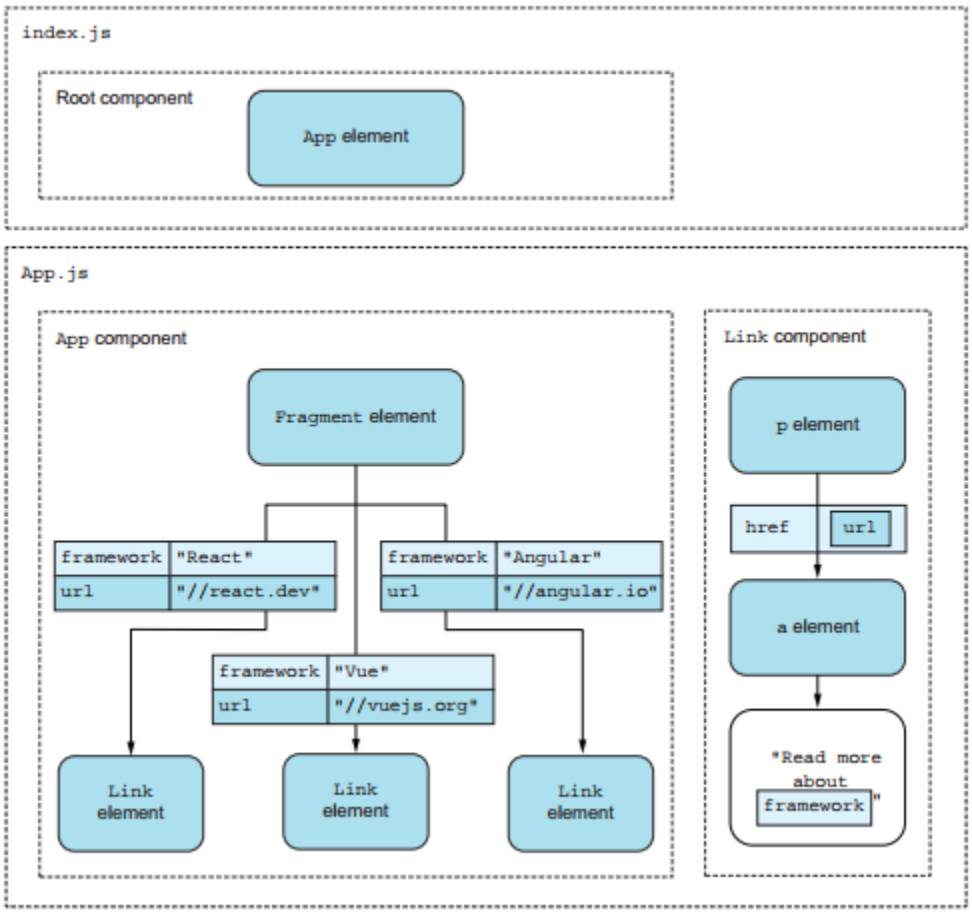


Рисунок 2.22 Наша нова файлова структура з нашими двома компонентами всередині файлу App.js

Лістинг 2.9 Програма, яка міститься в src/App.js

```
import React, { Fragment, Component } from "react";
class Link extends Component {
    render() {
        return React.createElement(
          "p",
```

```
          null,
          React.createElement(
            "a",
            { href: this.props.url },
            `Read more about ${this.props.framework}`
          )
        );
    }
}

class App extends Component {
    render() {
        const link1 = React.createElement(Link, {
            framework: "React",
            url: "//react.dev",
        });
        const link2 = React.createElement(Link, {
            framework: "Vue",
           url: "//vuejs.org",
        });
        const link3 = React.createElement(Link, {
            framework: "Angular",
            url: "//angular.io",
        });
        return React.createElement(
            Fragment, null, link1, link2, link3
        );
    }
}
export default App;
```

Потім ми змінюємо src/index.js, щоб імпортувати нашу програму з App.js і рендерити її в кореневий елемент DOM, як показано в лістингу 2.10.

Лістинг 2.10 src/index.js

```
import React from "react";
import { createRoot } from "react-dom/client";
import App from "./App";
createRoot(document.getElementById("root"))
.render(React.createElement(App));
```

Цей новий файл src/index.js тепер в основному готовий. Нам більше ніколи не потрібно його редагувати; ми лише редагуємо src/App.js для налаштування наших майбутніх програм.

Репозиторій: rq02-links-app

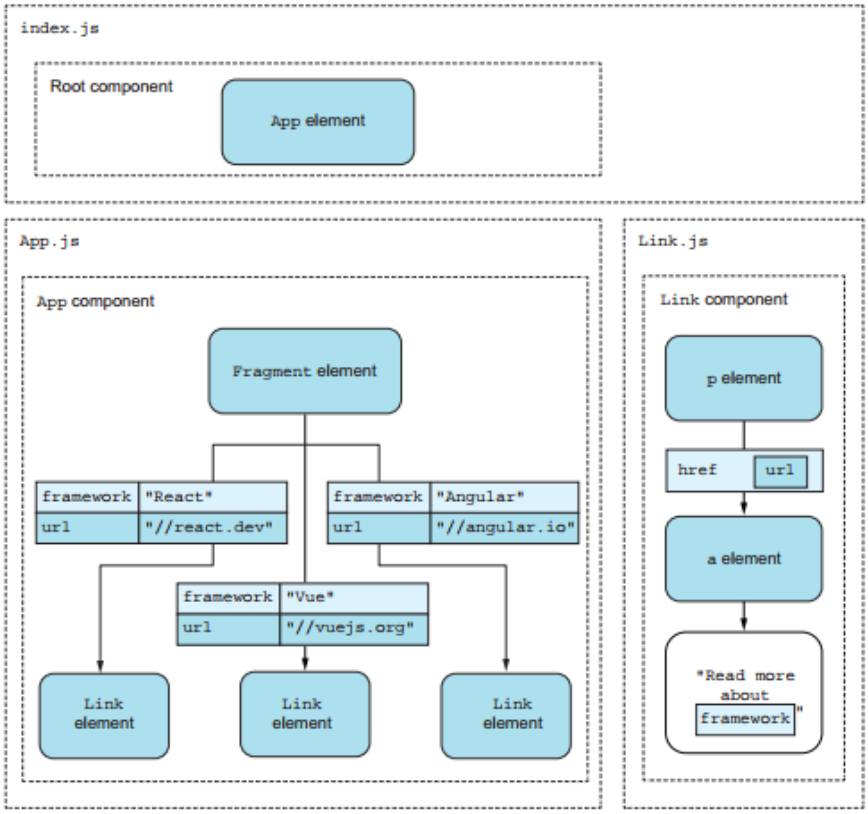
Цей приклад можна побачити в репозиторії rq02-links-app. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

```
$ npx create-react-app rq02-links-app --template rq02-links-app
```

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код у вигляді zip-файлу: https://rq2e.com/rq02-links-app

У міру того, як наші додатки збільшаться, ми перейдемо до включення всього в один файл у src/App.js. Коли нам потрібно розвиватися, ми можемо просто створювати нові файли та імпортувати їх за потреби. Хоча прийнято створювати окремий файл для кожного компонента та називати файл відповідно до компонента (включно з великою першою літерою), це не є суворим правилом. Якщо компонент потребує кількох інших невеликих компонентів для функціонування, ви можете вільно вирішити, чи хочете ви помістити це все в один файл, як ми робили з посиланням і додатком у лістингу 2.10, або розділити його на кілька файлів.

Давайте подивимося, як створити той самий приклад із компонентами програми та посилання в окремих файлах. Будь ласка, зверніться до діаграми на малюнку 2.23. Нам також знадобиться оновити src/App.js, щоб імпортувати компонент Link із src/Link.js, як показано в лістингу 2.11.



Малюнок 2.23 Використовуючи один файл на компонент, наша файлова структура виглядає так.

Лістинг 2.11 Один компонент на файл: src/App.js

```
import React, { Fragment, Component } from "react";
import Link from "./Link";

class App extends Component {
    render() {
        const link1 = React.createElement(Link, {
            framework: "React",
            url: "//react.dev",
        });
        const link2 = React.createElement(Link, {
            framework: "Vue",
            url: "//vuejs.org",
        });
        const link3 = React.createElement(Link, {
            framework: "Angular",
            url: "//angular.io",
        });
        return React.createElement(Fragment, null, link1, link2,
link3);
```

```
    }
}
export default App;
```

Потім ми повинні створити новий src/Link.js лише з компонентом Link і не забудьте експортувати його в кінці.

Лістинг 2.12 Один компонент на файл: src/Link.js

```
import React, { Component } from "react";
class Link extends Component {
    render() {
        return React.createElement(
            "p",
            null,
            React.createElement(
                "a",
                { href: this.props.url },
                `Read more about ${this.props.framework}`
            )
        );
    }
}
export default Link;
```

Репозиторій: rq02-links-app-alt

Цей приклад можна побачити в репозиторії rq02-links-app-alt. Ви можете використовувати це сховище, створивши нову веб-програму на основі відповідного шаблону:

$ npx create-react-app rq02-links-app-alt --template rq02-links-app-alt

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip: https://rq2e.com/rq02-links-app-alt

У книзі ми будемо використовувати обидва підходи. У багатьох наступних розділах наші програми будуть невеликими та компактними, тому ми розмістимо все в src/App.js. Але в наступних розділах наші програми збільшаться, і ми переростемо один файл. Ми також будемо створювати файли для інших речей, крім компонентів. Це можуть бути файли для часто використовуваних функцій або інших спільних функцій, які ми хочемо використовувати в багатьох файлах. Ми також будемо використовувати окремі файли для власних хуків, коли дійдемо до цієї теми в розділі 10.

Коли проекти стають ще більшими, вводяться структури папок. Немає визначених стандартів щодо того, як використовувати папки для структурування проекту React, тому команди часто придумують власні кращі практики.

Резюме

 Ви можете створювати нові проекти React за допомогою програми командного рядка createreact-app. Це дає змогу миттєво почати роботу з чудовим стартовим пакетом цінних бібліотек.

 Нові проекти React можна створювати з указаного шаблону, і всі приклади в цій книзі постачаються з шаблоном, який дозволяє вам переглядати приклад локально за допомогою трьох коротких команд без необхідності шукати або завантажувати щось безпосередньо. Натомість ми також надамо посилання для завантаження прикладів.

 Ви можете вкладати елементи React, помістивши вкладені виклики createElement() один в одного. Ви можете створювати однорідні вузли, використовуючи третій, четвертий і так далі аргументи в createElement().

 Ви можете створювати елементи на основі звичайних імен вузлів HTML, використовуючи назву вузла HTML як перший аргумент для createElement().

 Якщо ви хочете змінити отримані елементи за допомогою властивостей, ви можете передати їх як об’єкт як другий аргумент функції createElement().

 Щоб використовувати CBA (одну з функцій React), ви створюєте спеціальні компоненти. Користувацькі компоненти можуть внутрішньо використовувати властивості через змінну this.props. Дочірні вузли отримуються як дочірні властивості зі спеціальною назвою.

 Усі приклади в цій книзі будуть дотримуватись дуже простої файлової структури.

Розділ 3

Знайомство з JSX

Цей розділ охоплює

 Розуміння JSX та його переваг

 Використання JSX для швидшої та легшої реалізації спеціальних компонентів

 Проблеми React і JSX

JavaScript XML (JSX) — це синтаксичне розширення JavaScript. Це одна з речей, які роблять React чудовим, але це також був один із найбільш суперечливих елементів React, коли він був представлений свого часу.

Це приклад використання JSX у JavaScript:

```
const link = <a href="//react.dev">React</a>;
```

JSX — це елемент, який відображається між кутовими дужками: React. Це не рядок, не літерал шаблону і не HTML. Це об’єкт JavaScript, створений із розширенням синтаксису JSX. Це робить створення елементів React набагато швидшим і компактнішим, а також полегшує читання елементів React. Остання перевага принаймні така ж важлива, як і перша.

JSX створено лише для розробників. Сам по собі він нічого не робить для кращих або швидших веб-додатків. JSX перетворюється на той самий код, який ви отримуєте, коли не використовуєте JSX.

Хоча JSX не є обов’язковою умовою, він загальновизнаний як єдиний спосіб написання компонентів React. Ви можете знайти кілька команд, які не використовують JSX, але вони є меншістю.

. . . . . p.63

Розділ 4

Функціональні компоненти

Цей розділ охоплює

 Введення функціональних компонентів

 Порівняння функціональних компонентів з компонентами на основі класів

 Вибір між двома типами визначень компонентів

 Перетворення компонента на основі класу у функціональний компонент

У перші роки React тривалий час базувався на компонентах, заснованих на класах, але в якийсь момент з’явилася альтернатива для найпростіших компонентів. Функціональні компоненти — це більш стислий і, у деяких відношеннях, простіший спосіб написання компонентів React, і тепер вони мають той самий набір функцій, що й їхні родичі на основі класів.

Термін функціональний компонент не має на увазі як протилежність нефункціональному компоненту — вони нікому не потрібні. Натомість функціональна частина відноситься до самого визначення компонента, який є функцією JavaScript, а не класом JavaScript.

На початку функціональні компоненти були менш потужними, ніж компоненти на основі класів, але коли в React 16.8 з’явилися хуки React, функціональні компоненти раптово стали такими ж потужними, якщо не більшими, ніж їхні побратими на основі класів. Сьогодні багато розробників React використовують виключно функціональні компоненти, оскільки вони є основним методом, рекомендованим командою React.

Компоненти на основі класів все ще повністю підтримуються в React і, ймовірно, найближчим часом нікуди не дінуться. Ви також знайдете їх дуже поширеними «в дикій природі» з кількох причин:

 Не всі старі бази коду були перероблені з компонентів на основі класів, тому їх потрібно підтримувати.

 Деякі старіші бібліотеки все ще документують лише те, як вони взаємодіють із компонентами на основі класів, і тому вимагають, щоб ваш код використовував їх для правильного взаємодії з бібліотекою.

 Деякі давні розробники React почали використовувати компоненти на основі класів і почуваються з ними комфортніше, тому вони вважають за краще дотримуватися їх, коли це можливо.

 Ментальна модель життєвого циклу компонента суттєво змінилася під час переходу від компонентів на основі класів до функціональних компонентів, і в деяких випадках життєвий цикл повторного рендерингу може бути легше підтримувати при використанні старого підходу на основі класів.

 Крихітна підмножина основної функціональності в React можлива лише за допомогою компонентів на основі класу (зокрема, границя помилки).

Функціональні компоненти не тільки залишаться тут, але й збираються захопити світ — принаймні світ React. Усі показники вказують на те, що функціональні компоненти є основним способом написання React у майбутньому. Написання функціональних компонентів значно полегшує ваше життя як розробника без (майже) жодних недоліків.

У цьому розділі ми розглянемо, що таке функціональні компоненти, чим вони відрізняються (і чим вони не відрізняються) від компонентів на основі класу, як вибрати тип компонента для використання у ваших проектах і як можна перетворити клас компонент на основі функціонального.

ПРИМІТКА. Вихідний код для прикладів у цій главі доступний за адресою https://rq2e.com/ch04 . Але, як ви дізналися в розділі 2, ви можете створити екземпляри всіх прикладів безпосередньо з командного рядка за допомогою однієї команди.

. . . . . p.104

Розділ 5

Зробити React інтерактивним зі станами

Цей розділ охоплює

 Роль стану в компоненті

 Використання стану (state) у функціональних компонентах

 Перетворення компонентів на основі класів із збереженням стану на функціональні компоненти

Усі компоненти, які ми створили на даний момент, мають деякі властивості та відтворюють певний HTML на основі цих властивостей. Ми можемо передати властивість label кнопці, щоб кнопка відображалася, наприклад, із точним текстом кнопки. Але ми не можемо змусити текст кнопки змінюватися, коли щось трапляється, наприклад змінювати між Увімкнути та Вимкнути під час перемикання. Це тому, що нам не вистачає як здатності реагувати на те, що відбувається, так і здатності зберігати єдину інформацію про те, що щось динамічно змінюється.

. . . . p.136

Розділ 6

Ефекти та життєвий цикл компонента React

Цей розділ охоплює

 Запуск ефектів всередині компонентів

 Повний посібник із життєвого циклу компонента React

 Встановлення, демонтування та відтворення компонентів

 Представлення методів життєвого циклу для компонентів на основі класів

Компоненти React використовують JavaScript XML (JSX) для надсилання інформації користувачеві у формі HTML. Але компоненти повинні робити набагато більше, ніж це, щоб бути корисними в програмі. У React все, що відбувається, відбувається в якомусь компоненті, тож якщо ваша програма хоче встановити файли cookie, завантажити деякі дані, обробити введення даних, відобразити камеру користувача, запустити чи зупинити таймер чи безліч інших динамічних можливостей, ви потрібно більше, ніж просто JSX.

. . . . p.182

Розділ 7

Хуки для ваших веб-додатків

Цей розділ охоплює

 Більш широкий погляд на створення компонентів із збереженням стану

 Представлення складних тем, які можна вирішити за допомогою хуків

 Загальні правила використання хуків

Хуки – це те, що робить сучасні програми React ефективними. Вони є досить невеликою частиною загального React API, але все ж дуже важливою. Хуки також досить складні у використанні. У цьому розділі ми обговоримо всі хуки, їхню функцію та деякі важливі речі, які слід знати про використання хуків загалом.

Хуки — це особливий вид істот у біосфері React. Зовні вони здаються абсолютно непов’язаними за функціональністю, але при ближчому розгляді вони мають деякі спільні риси та поведінку, які ми повинні враховувати, використовуючи їх. Можна сказати, що вони походять від спільного предка десь на еволюційному дереві, навіть якщо вони просунулися, щоб стати дуже різними істотами .

Саме з цієї причини ми присвятили цю главу всім хукам. Отже, хоча ми збираємося охоплювати дуже різні теми, усі вони стосуються використання

хуків. Наприкінці ми зав’яжемо на ньому бантик, пояснюючи, як усі ці гачки насправді пов’язані, незважаючи на їхні, здавалося б, різні цілі.

Наразі ви бачили три різні хуки: useState (у розділі 5) і useEffect і useLayoutEffect (у розділі 6). На момент написання статті в React є 15 вбудованих хуків (станом на React 18), які ми коротко розглянемо, згруповані за їх функціональністю:

 Stateful hooks (Перехоплювачі з підтримкою стану) — ці функції спрямовані на те, щоб компоненти та програми мали статус на кількох різних рівнях і рівнях складності: useState, useReducer, useRef, useContext, useDeferredValue та useTransition.

 Effect hooks (Перехоплення ефектів) — ці функції пов’язані із запуском ефектів усередині компонента на різних етапах загального життєвого циклу компонента, а також під час кожного окремого циклу візуалізації: useEffect і useLayoutEffect.

 Memoization hooks (Хуки запам’ятовування) — ці функції використовуються для оптимізації продуктивності шляхом уникнення перерахунку значень, якщо їх складові частини не змінилися: useMemo, useCallback, useId.

 Library hooks (Бібліотечні перехоплювачі) — ці розширені функції майже виключно використовуються у великих бібліотеках компонентів, створених для спільного використання або спільнотою, або всередині великої організації.\ Ці функції рідко використовуються в програмах меншого або середнього розміру: useDebugValue, useImperativeHandle, useInsertionEffect і useSyncExternalStore.

Ці 15 хуків — це вбудовані «базові» хуки, з якими поставляється React. Ви можете створити більше гачків поверх них, але ви не можете створити власні базові гачки. Ви можете створювати лише хуки, які використовують один або декілька існуючих хуків. Ми обговоримо спеціальні хуки в розділі 10.

Зауважте, що React може бути розширено за допомогою додаткових вбудованих хуків у майбутніх випусках. React 18.0 поставляється з п’ятьма новими хуками, а додаткові випуски після React 18 можуть мати ще більше.

ПРИМІТКА. Вихідний код для прикладів у цій главі доступний за адресою https://rq2e.com/ch07 . Але, як ви дізналися в розділі 2, ви можете створити екземпляри всіх прикладів безпосередньо з командного рядка за допомогою однієї команди.

7.1 Stateful components (Компоненти з підтримкою стану)

Ми загалом розглянули компоненти з підтримкою стану в розділі 5, але з радістю повторимо цю інформацію тут для повноти. Компоненти з підтримкою стану та, у свою чергу, програми з підтримкою стану є важливими для того, щоб веб-програми були дійсно цікавими у використанні.

Додаток без стану є повністю статичним. Програма буде ідентичною протягом усього часу, коли вона відкрита в браузері, і вона буде ідентичною для кожного користувача, який використовує програму. Якщо вам потрібен вхід, сеанси, взаємодія, а також мінливість і зміни з часом, ваша програма має підтримувати стан.

Однак компоненти зі збереженням стану не однакові, як і не всі стани однакові. Один стан зберігається лише короткочасно, інший стан є гіперлокальним для окремого компонента, а ще один стан – для всієї програми. Крім того, стан може бути окремою змінною або величезною комплексною мережею взаємозалежних змінних, які мають оновлюватися в унісон. У цьому розділі ми розглянемо кілька різних варіантів використання компонентів і додатків із збереженням стану та обговоримо, як вирішити цю проблему за допомогою відповідних хуків.

7.1.1 Прості значення стану з useState

useState — це хліб і масло програм із збереженням стану. Ймовірно, ви використовуєте цей хук у більшості випадків, коли вам потрібен стан, тому він, безперечно, важливий.

Якщо у вас є меню, яке можна відкривати та закривати, ви зберігаєте його стан у локальному useState у вашому компоненті меню. Це одне просте значення використовується лише всередині цього компонента, і воно не пов’язане з будь-якими іншими значеннями стану в програмі.

Ми обговорили всі тонкощі useState у розділі 5, тому не будемо вдаватися в подробиці цього гачка тут. Однак у решті цього розділу ми

представимо деякі більш складні сценарії, коли useState недостатньо або є неоптимальним.

7.1.2 Створення складного стану за допомогою useReducer

Уявіть, що у нас є компонент завантажувача (loader), де ми хочемо знати, успішне чи невдале завантаження, яке повідомлення про помилку в разі невдачі та які дані є в разі успіху. Значення повідомлення про помилку має значення лише у разі помилки завантаження. Якщо завантаження вдається, повідомлення про помилку є абсолютно нерелевантним і його навіть не слід встановлювати, і навпаки для даних результату. Це приклад взаємозалежного стану. Окремі значення в стані залежать одне від одного, і ви часто оновлюєте кілька значень одночасно.

useReducer — це хук із збереженням стану саме для цієї мети. Це розширена версія useState, де ми можемо змінювати наш стан більш складним і контрольованим способом (майже як кінцевий автомат, але не насправді), якщо у нас є налаштування, складніше, ніж одне значення стану може розумно представити.

Використання useReducer — це спосіб генерувати новий стан («reduce») виключно на основі поточного стану та певної дії, яка вимагає певного корисного навантаження. Концепція зменшення стану відома з інших фреймворків, таких як Redux (звідси і назва), тому вона вже знайома багатьом розробникам React.

Зауважте, що useReducer ніколи не є суворо необхідним — усе, що ми можемо зробити за допомогою редуктора, також можна зробити за допомогою комбінації простіших useStates. Є багато випадків, коли ви, швидше за все, захочете використати редуктор, а не погоджуватися на кілька різнорідних станів, щоб забезпечити чіткіший потік даних і кращий контроль.

Ми наведемо кілька прикладів редуктора в розділі 10, коли ми перейдемо до більш складної архітектури програми. Редуктор актуальний лише для досить складних потоків даних, тому ми не будемо використовувати його часто в простих програмах, які ми створюємо в цій книзі.

7.1.3 Запам'ятовування значення без повторного відтворення за допомогою useRef

Уявіть, що ми хочемо створити кнопку, яка працює лише при подвійному клацанні протягом певної кількості мілісекунд. Щоб створити це, нам потрібно запам’ятати, скільки часу минуло між послідовними клацаннями. Запам'ятовування даних всередині компонента - це саме те, для чого ми маємо стан. У нас є значення, яке ми хочемо зберігати між візуалізаціями, але ми не використовуємо його для рендерингу. Кнопка не змінюється, коли ми натискаємо вперше. Нам просто потрібно запам’ятати значення в екземплярі компонента протягом деякого часу, але ми не будемо використовувати значення для визначення результату компонента.

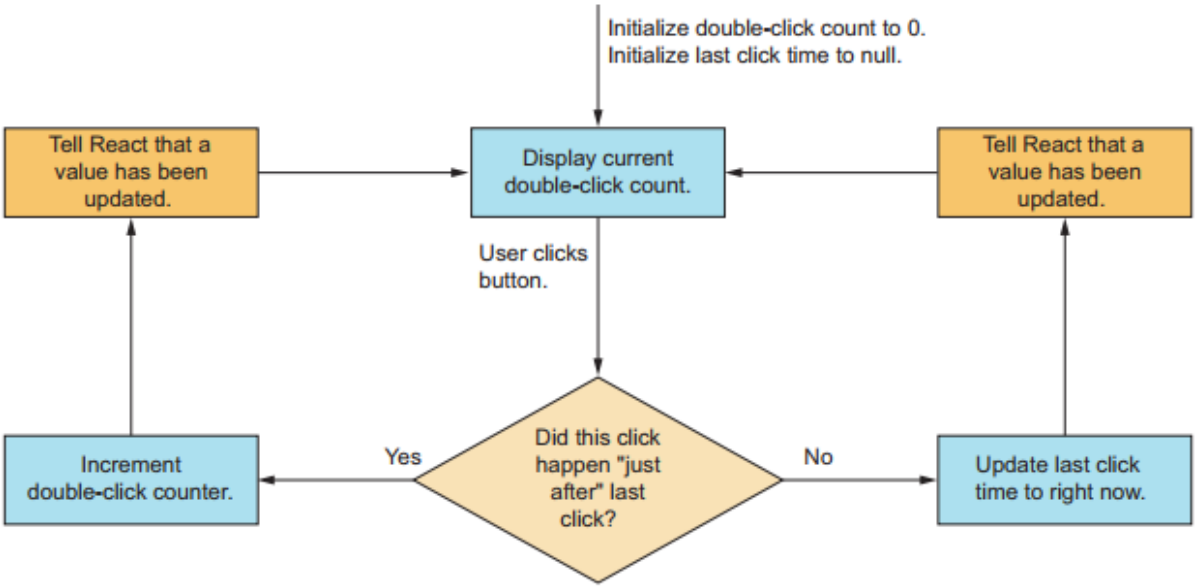
useRef є одночасно одним із найпростіших хуків у React, а також одним із найменш зрозумілих хуків. Це хук із пасивним станом, що означає, що хук може містити стан, але налаштування або оновлення стану не спричиняє повторне рендеринг.

useRef використовується для багатьох цілей, включаючи запам’ятовування значень між візуалізаціями та слугування посиланням на елементи DOM, що використовуються у рендерингу. Останнє є дуже важливим випадком використання (і причина назви useRef), оскільки це найкращий і найпростіший спосіб звертатися до елементів DOM через сценарій у ваших компонентах.

ЦІННОСТІ ПАСИВНОГО СТАНУ

Ви можете використовувати хук useRef, щоб запам’ятати певне значення, яке є релевантним між візуалізаціями компонента, але це не впливає безпосередньо на результат компонента. Це звучить трохи складно і навіть рідко. Коли ви матимете таке значення в компоненті? Як приклад, давайте ще раз відтворимо наш компонент лічильника, але цього разу з доданою функціональністю, що кнопка збільшення працює лише після подвійного натискання.

Нам потрібно зберегти час останньої події клацання десь у нашому компоненті, і він має бути в місці, яке зберігається між візуалізаціями. Ми вже знаємо, що ми можемо зберігати таке значення у стані, наданому хуком useState. Ескіз цього сценарію показаний на малюнку 7.1, а реалізація показана в лістингу 7.1.



Малюнок 7.1 Коли користувач натискає кнопку, виконання роздвоюється залежно від того, чи відбувається це клацання протягом дуже короткого часу після останнього зареєстрованого клацання.

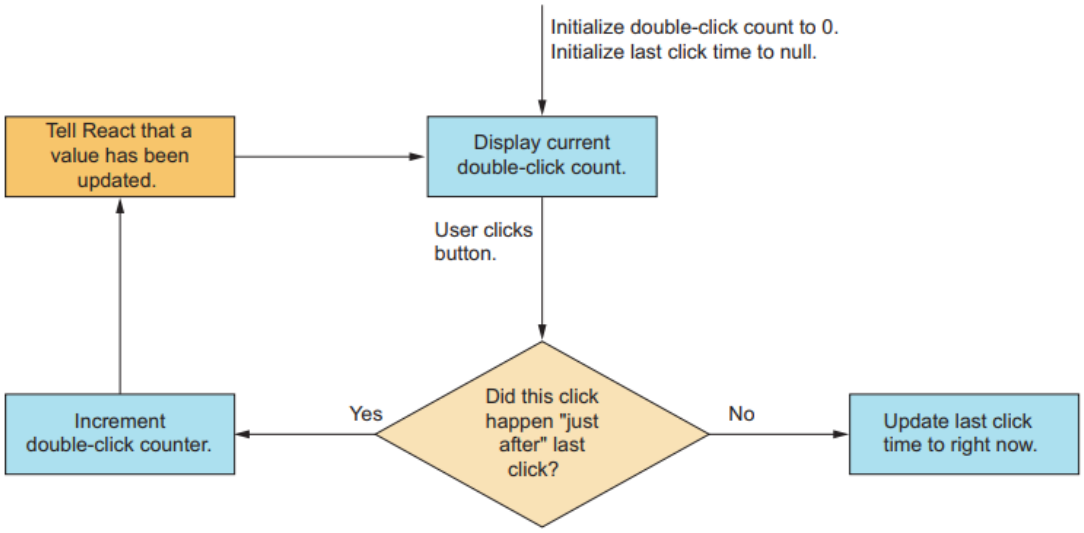
Лістинг 7.1 Лічильник подвійного клацання з useState

```
import { useState } from "react";
const THRESHOLD = 300;
function DoubleClickCounter() {
  const [counter, setCounter] = useState(0);
  const [lastClickTime, setLastClickTime] = useState(null);
  const onClick = () => {
    const isDoubleClick =
      Date.now() - lastClickTime < THRESHOLD;
    if (isDoubleClick) {
      setCounter((value) => value + 1);
    } else {
      setLastClickTime(Date.now());
    }
  };
  return (
    <main>
      <p>Counter: {counter}</p>
      <button onClick={onClick}>Increment</button>
    </main>
  );
}
function App() {
  return <DoubleClickCounter />;
```

```
}
export default App;
```

Однак це не є обов'язковим і призведе до непотрібних додаткових повторних візуалізацій. Коли ми викликаємо setLastClickTime, React повторно візуалізує компонент, оскільки змінюється значення стану. Однак JSX не зміниться в компоненті, і той самий вихід DOM буде відображено на екрані. Код у лістингу 7.1 працює, але він менш ніж оптимальний.

Оскільки нам потрібне лише значення стану всередині компонента, і не потрібне повторне відтворення компонента лише тому, що значення оновлено, замість цього ми можемо використовувати посилання через хук useRef. Щоб створити екземпляр хука useRef, ви просто викликаєте хук і зберігаєте його в змінній. Ви також можете передати початкове значення хуку. Щоб прочитати або оновити поточне значення хука useRef, ви отримуєте доступ до властивості .current значення, що повертається хуком. Порівняйте ескіз цього сценарію на малюнку 7.2 зі сценарієм на малюнку 7.1 — ми пропускаємо весь цикл рендерингу! Це реалізовано в лістингу 7.2.



Малюнок 7.2 Цього разу, якщо користувач клацає кнопку вперше, час останнього клацання записується, але це не викликає повторного рендерингу, оскільки це значення не активного стану, а пасивне. Це означає, що ми все ще можемо отримати доступ до значення пізніше (під час цього або майбутніх рендерів), але це не викликає нового рендеру само по собі.

Лістинг 7.2 Лічильник подвійного клацання з useRef

```
import { useState, useRef } from "react";
const THRESHOLD = 300;
function DoubleClickCounter() {
  const [counter, setCounter] = useState(0);
  const lastClickTime = useRef(null);
  const onClick = () => {
    const isDoubleClick =
      Date.now() - lastClickTime.current < THRESHOLD;
    if (isDoubleClick) {
      setCounter((value) => value + 1);
    } else {
      lastClickTime.current = Date.now();
    }
  };
  return (
    <main>
      <p>Counter: {counter}</p>
      <button onClick={onClick}>Increment</button>
    </main>
  );
}
function App() {
  return <DoubleClickCounter />;
}
export default App;
```

Репозиторій: rq07-double-counter

Цей приклад можна побачити в репозиторії rq07-double-counter. Ви можете використовувати це сховище, створивши нову програму на основі пов’язаного шаблону:

$ npx create-react-app rq07-double-counter --template rq07-double-counter

Крім того, ви можете перейти на цей веб-сайт, щоб переглянути код, побачити програму в дії безпосередньо у вашому браузері або завантажити вихідний код як файл zip: https://rq2e.com/rq07-double-counter

Це набагато краща версія нашого компонента, тому що у нас немає непотрібних повторних рендерів. Ми належним чином зберігаємо стан таким чином, щоб він працював, навіть якщо компонент повторно рендериться з іншої причини.

ПОСИЛАННЯ ДО ЕЛЕМЕНТІВ DOM

Інший варіант використання useRef — це, як уже згадувалося, отримання посилань на елементи DOM. Ви побачите, що це використовується багато разів протягом решти цієї книги. Ми використовуємо його, щоб мати посилання на фактичний елемент DOM, який відображається в документі як наслідок створеного нами елемента JSX. Синтаксис дуже простий:

```
function Component() {
  const ref = useRef(); //Creates a reference using the hook
  return <div ref={ref} />; //"Assigns" the reference to the DOM node,
                            // which, when the component is rendered,
                            // assigns a reference to the DOM node
                            // inside the ref object
}
```

Це ні для чого не використовує посилання, а лише створює його. Ви можете використовувати посилання в хуку ефекту, щоб, наприклад, викликати методи в елементі, що неможливо безпосередньо через властивості елемента DOM.

Наприклад, давайте автоматично сфокусуємо поле введення, коли компонент змонтовано:

```
function AutoFocusInput() {
  const ref = useRef();
  useEffect(() => ref.current.focus(), []);
  return <input ref={ref} />;
}
```

Останнє використання useRef є більш поширеним (і початковим призначенням). Можна ще багато сказати про те, як працюють useRef і властивість ref JSX, але поки що залишимо це тут. Ми розповімо про це в наступних розділах, де це буде доречно.

7.1.4 Простіший багатокомпонентний стан із useContext

useContext — це хук із збереженням стану, що означає, що він працює подібно до useState. Але замість того, щоб завантажувати й оновлювати значення в локальному сховищі, useContext працює в сховищі в батьківському компоненті десь вище в дереві компонентів. Це хук-версія React Context API, і ми побачимо більше про те, як це працює на практиці в розділі 10. Ми можемо показати, що це один із найпотужніших хуків, коли справа доходить до побудови хорошої архітектури.

7.1.5 Оновлення стану з низьким пріоритетом за допомогою useDeferredValue і useTransition

ПРИМІТКА. Ця тема є досить просунутою та абсолютно новою. Ці функції щойно додано з React 18, тож попереду ще багато відкриттів щодо кращих практик. Це також складна тема, яка, ймовірно, не актуальна для повсякденних програм. У цьому розділі ми не будемо повністю описувати функціональні можливості цих хуків, а лише коротко окреслимо, чому вони існують. Якщо на даний момент вони вас не цікавлять, сміливо пропустіть цей розділ.

Уявіть, що ми створили онлайн-додаток для редагування документів, подібний до Google Docs, і ми додали до нього багато функцій. Зараз ми додаємо перевірку орфографії до програми та хочемо зробити це, додавши кнопку, яку можна натиснути, щоб увімкнути перевірку орфографії, і натиснути ще раз, щоб вимкнути її. Кнопка має інший колір тла, якщо ввімкнено.

Коли користувач натискає цю кнопку, він повинен миттєво реагувати. Кнопка має змінити відображення протягом кількох мілісекунд, щоб користувач відчув, що кнопка працює. Ми можемо реалізувати це за допомогою стану, просто перемикаючи властивість enabled всередині кнопки за допомогою сетера useState.

Але наша кнопка зробить більше, ніж це. Якщо перевірку орфографії ввімкнено, усі помилки в документі мають бути виділені червоною лінією під ними. Якщо користувач працює над великим документом, пошук і виділення всіх помилок може бути дорогою операцією. Вам потрібно виконати багато операцій над усіма словами в документі, щоб знайти помилки та, можливо, навіть знайти пропозиції щодо правильного написання для кожного. Виконання цих завдань може легко зайняти десяті частки секунди або навіть цілі секунди.

Якщо ми оновимо як внутрішній прапор увімкненого стану всередині кнопки, так і глобальний прапор, який ініціює обчислення перевірки орфографії з однаковим пріоритетом одночасно в React, внутрішні елементи React розглядатимуть обидва оновлення як такі, що відбуваються одночасно, і вирішать нічого не рендерити, доки обидва оновлення не будуть повністю обчислені. Це не ідеально, оскільки кнопка перевірки орфографії здаватиметься нефункціональною. Коли користувач натискає кнопку і нічого

не відбувається, він натискає її знову. Тоді раптом кнопка фактично запрацює після завершення обчислення, але оскільки користувач уже натиснув її знову, користувач вимикає цю функцію. Це жахливий досвід користувача.

Що, якби ми могли повідомити React, що оновлення стану кнопки має високий пріоритет і має відбуватися негайно, тоді як виділення всіх орфографічних помилок у документі має нижчий пріоритет, і ми не заперечуємо, якщо це відстає від натискання кнопки на кілька циклів візуалізації? React 18 представив абсолютно нову концепцію Concurrent Mode, яка дозволяє саме це. Хуки useDeferredValue і useTransition — це два різні способи визначення низькопріоритетних оновлень стану з двох різних точок зору. Враховуючи складність цих хуків, ми не будемо розглядати їх далі в цій книзі, але, будь ласка, зверніться до наступних онлайн-матеріалів для отримання додаткової інформації:

- Стаття: http://mng.bz/jPAP
- Відео: http://mng.bz/WzM1

7.2 Ефекти компонентів

Цей розділ буде коротким, тож намагайтеся не моргати, інакше ви можете його повністю пропустити! Компонентні ефекти — це група хуків, призначених для запуску побічних ефектів із трьох різних цілей:

- Впливати ззовні залежно від стану компонента
- Щоб оновити стан компонента на основі чогось іззовні
- Одночасно впливати назовні та оновлювати стан компонента

Ми вже бачили два таких хуки, представлені в попередньому розділі, useEffect і useLayoutEffect. Існує лише ще один такий хук, useInsertionEffect, але він зарезервований для розширеного використання певними бібліотеками, тому його не рекомендується використовувати «звичайним» розробникам.

Ми не будемо додавати жодної інформації про useEffect і useLayoutEffect у цьому розділі, оскільки ми розглянули все, що потрібно знати в розділі 6. Останній хук ефекту, useInsertionEffect, буде коротко розглянуто в розділі 7.4.

7.3 Оптимізація продуктивності шляхом мінімізації повторного відтворення

ПРИМІТКА Це розширена тема, яка не потрібна більшості простих програм. Ми лише коротко познайомимося з цією темою, а не детально, оскільки це не обов’язково для вашої першої, другої чи навіть десятої заявки. Лише коли ви починаєте переходити до більших програм із десятками чи навіть сотнями компонентів, ці хуки починають сяяти.

Якщо ви працюєте над більшою програмою з багатьма рухомими частинами, оновленням даних з багатьох джерел і подіями, які прослуховують багато типів вхідних даних, продуктивність може почати знижуватися, якщо компоненти рендеряться без потреби. Коли ваша програма дійде до цієї точки, мемоізація (memoization) може стати тим трюком, який допоможе вашій програмі відновити свою чуйність.

За своєю суттю, запам’ятовування — це принцип кешування результату даного обчислення, щоб, якщо те саме обчислення було виконано пізніше, повертався кешований результат. Це можна використовувати в React кількома способами, включаючи три хуки, які ми представимо в цьому розділі.

Як згадувалося, ми не будемо вдаватися в подробиці про ці хуки в цій главі, оскільки вони не є необхідними, якщо ви починаєте роботу як React-розробник. Насправді, неправильне застосування запам’ятовування може призвести до погіршення продуктивності, а не до покращення. Тому, враховуючи складний характер цієї теми, ми взагалі не використовуємо запам’ятовування в цій книзі, і ми лише коротко познайомимося з трьома хуками в наступних підрозділах.

Ви можете прочитати більше про оптимізацію продуктивності React у Job-Ready React (Morten Barklund, Manning, 2024), який охоплює не лише ці хуки та мемоізацію загалом, але й інші способи зробити вашу програму більш продуктивною.

7.3.1 Запам'ятовування будь-якого значення за допомогою useMemo

Скажімо, вам потрібно відобразити криптографічний хеш даного пароля, який введено в поле введення. Розрахунок такого хешу є досить дорогим, тому ви не хочете виконувати обчислення, якщо пароль не змінюється. Але ваш компонент повторно рендериться кілька разів, навіть без зміни пароля. Для цього та подібних випадків ви можете використовувати хук

useMemo, щоб перерахувати дане значення в компоненті, лише якщо його залежності змінюються.

7.3.2 Функції запам'ятовування за допомогою useCallback

useCallback — це лише спеціалізована версія useMemo, яка корисна, коли useMemo використовується для запам'ятовування функції. Але оскільки це трапляється дуже часто, хук useCallback існує для цієї мети і часто використовується більше, ніж useMemo.

7.3.3 Створення стабільних ідентифікаторів DOM за допомогою useId

Це ще більш просунута тема, яка стосується лише створеного сервером React. Щоб зрозуміти обставини цього хука, який має таке надзвичайно вузьке використання, потрібні досить великі знання.

useId гарантує, що для двох повністю ідентичних дерев компонентів, якщо окремий компонент всередині будь-якого дерева викликає useId, він отримає той самий ID, незалежно від того, на якій платформі запущено хук. Це використовується для того, щоб згенерований HTML був ідентичним на клієнті та на сервері.

7.4 Створення складних бібліотек компонентів

Цей розділ включено лише для завершення, тому ми розглядаємо всі хуки в React. Чотири хуки, описані в наступних підрозділах, усі дуже просунуті та використовуються рідко. Вони призначені для пакетів багаторазового використання, таких як бібліотеки компонентів або модулі з відкритим кодом.

Останні два хуки, згадані в цьому розділі, представлені в React 18 як наслідок нового режиму Concurrent. Деякі бібліотеки потрібно оновити для правильного рендерингу в паралельному режимі, щоб уникнути логіки обчислень, яка не потрібна або є передчасною через паралельність. Не соромтеся пропустити цей розділ і перейти до розділу 7.5, якщо ви хочете перейти до більш практичних речей.

. . . . p.223

7.5 Два ключових принципи хуків

Вам потрібно лише дотримуватися двох правил щодо хуків React:

 Беззастережно викликайте хуки лише на верхньому рівні функціональних компонентів.

 Викликайте хуки лише всередині функціональних компонентів.

Перше правило, яке ми вже обговорювали: ви можете використовувати хуки лише безпосередньо у ваших компонентах, і ви повинні завжди включати однакову кількість хуків. Це означає, що ви ніколи не можете викликати хуки всередині функції (зокрема всередині функції, що використовується в хуку) або вкладеного блоку (як умовного, так і циклічного), і ви не можете отримати ранні повернення у своєму компоненті, доки не відобразите всі свої хуки.

Друге правило начебто очевидне, але, можливо, неочевидне: ви можете використовувати хуки лише всередині функціональних компонентів. Ви не можете створити якусь допоміжну функцію або зворотний виклик, які викликають хук. Ви також не можете використовувати їх всередині компонентів на основі класу.

Єдиний виняток із цього правила полягає в тому, що ви можете використовувати хуки всередині інших хуків, які називаються користувальницькими хуками, і ви знову можете використовувати користувацькі хуки всередині інших користувацьких хуків і так далі. Але ви можете використовувати ці користувацькі хуки лише в інших користувальницьких хуках або у своїх компонентах, тому ви не можете обійти це правило — ви можете просто приховати його одним (або кількома) шарами вниз. Ми розглянемо спеціальні хуки в главі 10.

Резюме

 React має 15 різних вбудованих хуків, але деякі з них використовуються рідко, залишаючи близько 10 основних API, на яких побудовані всі програми React.

 Хуки використовуються для різноманітних цілей, які роблять компоненти розумними та здатними взаємодіяти з веб-сторінкою в цілому. Незважаючи на те, що всі хуки дуже відрізняються за призначенням, усі вони мають деякі спільні риси.

 Хуки з підтримкою стану потрібні, щоб зробити додатки такими, що мають стан. Ви можете використовувати кілька різних хуків залежно від складності вашої програми та значень у вашому стані. За допомогою React 18 ви навіть можете робити оновлення стану з нижчим і вищим пріоритетом, щоб допомогти React зробити ваш інтерфейс користувача максимально адаптивним.

 Перехоплювачі ефектів використовуються для запуску побічних ефектів всередині компонентів, як ви дізналися в розділі 6. Використовуючи масив залежностей, ви можете запускати свій ефект у потрібний час(и).

 Хуки запам’ятовування використовуються для оптимізації візуалізації в React, коли ваша програма стає великою та складною.

 Бібліотечні хуки призначені лише для більш складних кодових баз і, ймовірно, не стосуються ваших повсякденних програм.

 Якщо ви використовуєте хук, ви повинні дотримуватись двох законів хуків: викликати хуки лише на верхньому рівні компонента (тобто ніяких умовних хуків чи циклів хуків) і використовувати хуки лише всередині функціональних компонентів (тобто хуки поза межами компонент, у допоміжній функції або навіть у компоненті на основі класу).

Розділ 8

Обробка подій у React

Цей розділ охоплює

 Реагування на введення користувача за допомогою подій

 Обробка захоплення подій і підсвічування

 Керування діями подій за замовчуванням

 Приєднання слухачів подій безпосередньо до DOM

Події – це спосіб взаємодії користувачів із веб-програмою JavaScript. Події можуть бути спричинені рухом або клацанням миші, клацанням і перетягуванням сенсорного інтерфейсу, натисканням кнопок на клавіатурі,

прокручуванням, копіюванням і вставленням, а також непрямими взаємодіями, такими як фокусування та скасування фокусування елементів або всієї програми.

Наразі ми створювали додатки React з дуже невеликою кількістю взаємодії з користувачем. Ми обробляли натискання кнопки тут і там, але не детально розповідали про те, як працює подія натискання та як ми, розробники, обробляємо це. Ми збираємося змінити це в цьому розділі, присвяченому обробці подій.

Ви можете розглядати події як спосіб обробки введених даних від користувача. Наша веб-програма створює JavaScript XML (JSX), який перетворюється на HTML. Потім користувач взаємодіє з цим HTML, і результатом цих взаємодій є події, які надсилаються з елементів HTML до нашої програми React. Цей простий потік інформації показано на малюнку 8.1.

Рисунок 8.1 Потік інформації між React і користувачем проходить через HTML. Уявіть користувача, який відвідує сторінку входу. Користувач вводить адресу електронної пошти та пароль, браузер пересилає ці взаємодії як події до React, програма потім генерує JSX, необхідний для відображення зеленої галочки біля кожного введення, а браузер відтворює відповідний HTML для відображення користувачеві .

Події також використовуються всередині браузера, щоб позначити, коли щось змінюється між елементами. Це може бути, коли відео відтворюється/призупиняється/буферизується, анімація завершена, вузол DOM змінено, дані завантажено (або не вдалося завантажити) тощо. Існують сотні можливих подій, і будь-яка інтерактивна веб-програма використовуватиме значну їх частину. (Ви можете прочитати більше про всі можливі події DOM у довідковому документі подій за адресою http://mng.bz/9D1j .)

Є два способи обробки подій у React:

 Ви можете використовувати React для керування слухачем подій.

 Ви можете вручну додавати та видаляти слухач подій безпосередньо на вузлі DOM

Покладаючись на React для обробки слухачів, ви заощаджуєте купу виснажливої роботи та головного болю (і потенційних витоків пам’яті), але це супроводжується незначною втратою гнучкості. Безпосереднє додавання слухачів подій дозволяє вам прослуховувати всі типи подій і призначати слухачів будь-яким вузлам, які вам потрібні, коли вам це потрібно, але це пов’язано з вартістю керування слухачами (і не забувайте видаляти їх знову), а також роботи з власні події, які можуть відрізнятися в різних браузерах.

У цьому розділі ми покажемо вам обидва підходи та обговоримо, коли краще застосувати той чи інший. Зауважте, що обробка подій у React набагато простіша та рекомендована. Тому в цій главі цей сценарій буде розглянуто більш детально.

Коли ми розповідаємо про те, як ви можете прослуховувати події за допомогою інтерфейсу React, ми обговоримо кілька тем про те, як React обробляє події та як ви можете працювати з React API для прослуховування конкретних подій, які вам потрібні. Ми відповімо на такі питання:

 Які події підтримуються?

 Як створити функцію обробки подій?

 Які предмети події ви отримаєте?

 Як працюють фази події та поширення?

 Як ви обробляєте події на етапі захоплення розсилки подій?

 Що таке дії за замовчуванням і як їм запобігти?

 Коли слід продовжувати (persist) подію?

 Чи можна використовувати властивості як обробники подій?

 Що таке генератори обробників подій?

Потім ми перейдемо до ситуацій, коли вбудована обробка подій React недостатня, і нам потрібно обробляти події вручну в DOM. Ми також дамо вам усю інформацію про те, як це зробити найкраще.

Усе це призведе до наступного розділу, де ми використаємо наше нове розуміння обробки подій для створення інтерактивних вводів форм і форм загалом, що є наріжним каменем багатьох веб-програм.

ПРИМІТКА. Вихідний код для прикладів у цій главі доступний за адресою https://rq2e.com/ch08 . Але, як ви дізналися в розділі 2, ви можете створити екземпляри всіх прикладів безпосередньо з командного рядка за допомогою однієї команди.

8.1 Обробка подій DOM у React

Події є важливим способом спілкування в браузері між користувачем і сценарієм, а також між різними елементами програми. Через це належна обробка подій є first-class citizen у React, а це означає, що React присвятив велику частину свого основного API саме цій меті.

API дуже простий. Якщо ви визначаєте властивість елемента JSX, яка посилається на вузол HTML, і ця властивість відповідає відомій події зі списку підтримуваних подій React, React розглядатиме властивість як слухач події, а не як атрибут DOM. Тоді React переконається, що правильно додає та видаляє слухач подій, коли компонент монтується та демонтується.

8.1.1 Основна обробка подій у React

Найважливішою подією майже в будь-якій веб-програмі є подія клацання (click event). Всупереч своїй назві, вона використовується не лише для прийому клацань миші. Подія клацання в HTML також викликається, коли користувач сенсорного екрана натискає кнопку (або посилання) або коли користувач клавіатури активує кнопку (або посилання) за допомогою клавіші Enter.

Давайте повернемося до нашого надійного компонента лічильника та розглянемо докладніше, як ми обробляємо подію кліку. Якщо ви пам’ятаєте, у цій програмі була кнопка, і ми збільшували значення стану у відповідь на натискання користувача. Спочатку давайте повторимо код для цієї простої програми.