МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
_____Аліна САВЧЕНКО
"____"_____2021 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

"МАГІСТРА"

ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ "ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ"

Тема: "Багаторазова бібліотека компонентів React в менеджері пакетів npm"

Виконавець: Петренко Олексій Сергійович

Керівник: д.т.н., проф. Воронін Альберт Миколайович

Нормоконтролер: _____ Ігор РАЙЧЕВ

Київ - 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра Комп'ютерних інформаційних технологій

Галузь знань, спеціальність, освітньо-професійна програма: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології".

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Аліна САВЧЕНКО

«____»_____2021р.

ЗАВДАННЯ

на виконання дипломної роботи студента

Петренка Олексія Сергійовича

(прізвище, ім'я, по батькові)

1. Тема роботи: «Багаторазова бібліотека компонентів React в менеджері пакетів npm» затверджена наказом ректора від 12.10.2021 за № 2228/ст.

2. Термін виконання роботи: з 12.10.2021 по 31.12.2021.

3. Вихідні дані до роботи: теоретичні відомості та основи проектування багаторазових бібліотек, множина відомих архітектур програмних компонентів, патерни створення документації, множина відомих патернів проектування.

4. Зміст пояснювальної записки: вступ, аналітичний огляд і постановка завдання, структура багаторазової бібліотеки, вибір технологій, реалізація, тести, висновок.

5. Перелік обов'язкового ілюстративного матеріалу: слайди, скріншоти, презентація.

6. Календарний план-графік

№ п/п	Завдання	Термін виконання	Підпис керівника
1.	Отримання завдання на дипломну роботу та побудова плану-графіку виконання робіт.	12.10.2021 – 15.10.2021	
2.	Огляд та аналіз відомих програмних архітектур і патернів.	16.10.2021 – 19.10.2021	
3.	Огляд та створення концепції багаторазової бібліотеки компонентів React.	20.10.2021 – 24.10.2021	
4.	Проектування багаторазової бібліотеки компонентів React.	25.10.2021 – 31.10.2021	
5.	Написання Розділу 1 дипломної роботи.	01.11.2021 – 07.11.2021	
6.	Розробка та реалізація багаторазової бібліотеки компонентів React. Написання Розділу 2 дипломної роботи.	08.11.2021 – 17.11.2021	
7.	Написання Розділу 3 дипломної роботи. Завершення створення пояснювальної записки дипломної роботи.	18.11.2021 – 01.12.2021	
8.	Оформлення та друк пояснювальної записки.	02.12.2021 – 11.12.2021	
9.	Створення презентації, доповіді та підготовка до захисту дипломної роботи	12.12.2021 – 20.12.2021	

7. Дата видачі завдання: 12.10.2021р.

Керівник дипломної роботи _____ Альберт ВОРОНІН

(підпис керівника)

Завдання прийняв до виконання _____ Олексій ПЕТРЕНКО

(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи “Багаторазова бібліотека компонентів React в менеджері пакетів npm” складається зі вступу, трьох розділів, висновків, списку використаних джерел і містить 119 сторінок тексту та 26 рисунків. Список використаних джерел містить 8 найменувань.

Метою є створення інструменту для полегшення розробки та підтримки проектів на основі React.

Предметом дослідження є проектування та розробка є багаторазової бібліотеки компонентів на основі React.

Об’єктом дослідження є багаторазова бібліотека компонентів React.

Ключові слова: БАГАТОРАЗОВА БІБЛІОТЕКА, КОМПОНЕНТИ, REACT, АТОМНИЙ ДИЗАЙН, ПЕРЕВИКОРИСТАННЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

iOS – iPhone OS, мобільна операційна система продуктів компанії Apple

Android – мобільна операційна система продуктів компанії Google

NPM – Node Package Manager, менеджер пакетів

CRA – create react app

TS – TypeScript

NVM – Node Version Manager

JSON – JavaScript Object Notation

HTML – HyperText Markup Language

DOM – Document Object Model

API – application programming interface

CSS – Cascading Style Sheets

SASS – Syntactically Awesome Stylesheets

LESS – Leaner Style Sheets

JSX – JavaScript XML

ES – ECMAScript

UMD – Universal Module Definition

TDD (Test Driven Development) – розробка на основі тестів

BDD (Behavior Driven Development) – розробка на основі поведінки.

ВСТУП

Програмне забезпечення проникає в наше повсякденне життя, можливо, немає іншого створеного людиною матеріалу, який був би більш повсюдним, ніж програмне забезпечення в нашому сучасному житті. Він став невід'ємною частиною багатьох частин суспільства, магазинів, телекомунікацій, побутової техніки, літаків, особистих розваг, аудиту, автомобілів тощо. Зокрема, технології та наука вимагають високоякісного програмного забезпечення для вдосконалення та проривів. Крім того, Гілл сказав, що спільнота розробників програмного забезпечення неухильно рухається до широкого повторного використання програмного забезпечення, коли будь-яке програмне забезпечення може бути отримано з існуючого коду. В результаті все більше розробників програмного забезпечення використовують програмне забезпечення не тільки як комплексну систему, але і як модульну частину більшої системи. Повторне використання коду не означає, що ми зможемо копіювати та вставляти той самий код у багатьох частинах системи. Насправді це означає зовсім протилежне. Зокрема, фрагмент коду для повторного використання означає, що той самий код можна повторно використовувати в багатьох частинах без його переписування.

Ось лише чотири причини покладатися на багаторазові компоненти під час створення програми:

Ефективність — 80 відсотків роботи виконано з уже створеною базою. Ми можемо заощадити час і гроші, покладаючись на деякі багаторазові компоненти. Це означає, що програма працює швидше і за меншу вартість. У дуже великих організаціях наявність спільної бази даних компонентів запобігає подвійності між командами.

Послідовність — основні функції будуть узгоджені, що забезпечить більший контроль і масштабованість для тих, хто створює програми, а також більш послідовний досвід для користувачів.

Перевірений код — код, який багато разів використовувався різними людьми, попередньо перевірений і вже перевірений у цій галузі. Ми порівнюємо це з виробничими потужностями, які покладаються на деталі, створені в інших місцях, щоб зібрати повний продукт. Деталі перевірені, і ви знаєте, що вони працюють; таким чином, коли ви підключаєте їх до свого продукту, ви менше можете зіткнутися з проблемами під час розробки.

Простіше тестування — оскільки код уже перевірений, тестування також легше. Базову функціональність деяких функцій уже перевірено. Тепер ми тестуємо, щоб переконатися, що все це працює разом, і щоб будь-які нові функції або розділи працювали добре. Якщо ми виявимо проблеми, у нас буде менше місць для пошуку причини, тому що, швидше за все, проблема в новій частині, яку ми створили.

Компоненти для багаторазового використання доступні незалежно від того, кодуєте ви рідну програму для Android чи iOS. Фреймворк дизайну інтерфейсу користувача React Native від Facebook викликає багато дискусій у наші дні. Ця бібліотека JavaScript полегшує роботу розробникам гібридних мобільних додатків, надаючи основу для багатьох стандартних функцій програми, таких як коментування, пошук, реклама публікації та перевірка сповіщень.

Ми розглянемо, як розробляти, створювати та публікувати багаторазові компоненти React, якими можна поділитися зі своєю командою, своєю компанією чи навіть світом. Деякі з основних тем, які ми розглянемо, включають поради, підказки та шаблони для створення компонентів React, принципи атомарного проектування, методи створення спеціальної документації, стилізацію компонентів багаторазового використання та тестування, і завершимо упакуванням та опублікуванням бібліотеки компонентів, яку ми створюємо для npm.

РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ

1.1. Багаторазові компоненти

Компоненти — це маленькі шматочки Lego, які дають нам більше контролю над тим, як реалізуються майбутні інтерфейси користувача. Так само, як і Lego, багаторазові компоненти дозволяють нам об’єднувати разом і створювати все більш вражаючі додатки. Розглянемо десять причин, чому багаторазові компоненти варті роботи. Узгодженість – компоненти допомагають забезпечити узгоджений вигляд, відчуття, доступність, інтернаціоналізацію та функції. Тому споживачам багаторазових компонентів не потрібні глибокі знання з цих питань. Компоненти допомагають створити спільний словник між командами, оскільки спільні компоненти чітко оголошують запропоновану термінологію для концепцій інтерфейсу користувача. Забезпечена послідовність підвищує ефективність, уникаючи втоми від прийняття рішень. Чим менше рішень потрібно прийняти розробникам, тим швидше вони зможуть рухатися. Багаторазові компоненти також можуть підвищити ефективність за рахунок економії пропускної здатності. Ми відправляємо менше байтів, коли можемо оголосити один компонент, який використовується в кількох місцях на даній сторінці. Компоненти полегшують співпрацю та швидке переміщення між командами. Як сказав Ден Абрамов: «Компоненти допомагають сотням інженерів працювати на одній кодовій базі, переміщатися між командами, швидко розвивати та зосереджуватися на продуктах».

Зовнішній вигляд, доступність та інтернаціоналізація мають бути загальними, щоб інженери продуктів могли використовувати їх без глибоких знань. Компоненти React можуть бути багатоплатформенними. Це означає, що одна команда може володіти своїм кодом для всіх платформ. З React Native той самий компонентний підхід працює для iOS та Android. Одна команда продукту володіє своїм кодом для всіх платформ. Компоненти багаторазового використання допомагають підтримувати архітектуру інтерфейсу. Це не дозволяє кожній команді придумувати свою власну архітектуру. Вкладені компоненти забезпечують чіткий шлях до складання складних додатків з використанням невеликих комбінованих будівельних блоків.

Компоненти створюють зручну для майбутнього основу для команд, які можна змінювати, розширювати та вдосконалювати з часом. Це пришвидшує розробку як в короткостроковій, так і в довгостроковій перспективі, оскільки дозволяє уникнути створення майбутніх проектів з нуля. Оскільки бібліотека компонентів розростається в ширину та глибину, кожен проект вимагає менше спеціальної роботи, оскільки у нас є міцніший і багатий фундамент для розбудови. Багаторазові компоненти є автономними. Вони інкапсулюють набір проблем у доступний і багаторазовий спосіб. Нам не потрібно бути експертом, щоб використовувати компонент, оскільки він забезпечує зрозумілий загальнодоступний інтерфейс. Це означає, що багаторазові компоненти можуть допомогти нашій команді масштабуватися. Компоненти дозволяють нам створювати користувацькі інтерфейси з дедалі багатшими будівельними блоками. Всі ці переваги ведуть до пришвидшеного розвитку. У міру зростання нашої бібліотеки компонентів ми все частіше зніматимемо наявні компоненти з полиці та поєднуватимемо їх разом. Це призводить до набагато швидшої доставки, ніж починати з нуля. І, нарешті, менше написаного коду також призводить до менше помилок. Компоненти можна тестувати ізольовано, щоб гарантувати відповідальність, тож коли вони складаються разом, нам потрібно лише турбуватися про проблеми інтеграції між компонентами.

1.2. Веб-компоненти

Кілька років тому Google допоміг розпочати ініціативу щодо оголошення стандартизованих веб-компонентів, які підтримувалися веб-платформою. Ідея приваблива, нам не потрібна бібліотека JavaScript, щоб вони працювали. Більшість розробників продовжують використовувати Angular і React. То чому ж розробники не використовують стандарт веб-компонентів? Ну, по-перше, сучасні бібліотеки JavaScript, такі як React, Angular, Ember і Vue пропонують переконливі історії компонентів, які продовжують удосконалюватися. Доведено, що органам зі стандартизації важко йти в ногу з історією компонентів та інструментів на основі JavaScript, що постійно покращується. Сьогодні сучасні бібліотеки пропонують ряд переконливих функцій для зв'язування, об'єднання в пакети, упаковки та тестування компонентів на основі JavaScript. Але, ймовірно, найбільша причина, по якій люди не використовують звичайні веб-компоненти, полягає в тому, що підтримка браузера залишається досить неякісною. Наприклад, тег шаблону не підтримується жодною з версій Internet Explorer. Імпорт HTML підтримується лише в Chrome, Opera та Android. Користувацькі елементи підтримуються лише в Chrome, Opera та деяких найновіших браузерах Android. І майже така сама сумна історія з тіньовим DOM. На жаль, через роки очікування це стало зрозуміло. Постачальники веб-переглядачів виявляли невеликий інтерес до впровадження повного набору функцій веб-компонентів HTML5. Тож ми застрягли, додаючи поліфіл або шукаючи бібліотеки, як от полімер, щоб усе це працювало. І коли ми залучаємо додатковий JavaScript, ми повинні запитати, чому ми вибираємо не популярний і погано підтримуваний стандарт, як веб-компоненти, а не надзвичайно популярну технологію, як React. Тепер ми можемо до певної міри заповнювати ці відсутні функції, але додавання поліфіла обтяжує сторінку, що пом'якшує деякі переваги використання вбудованого стандарту. Це проблема, оскільки веб-компоненти не дозволяють нічого нового. Все, що ми можемо робити у веб-компонентах, сьогодні також може бути

виконано в крос-браузерний спосіб, використовуючи різноманітні сучасні бібліотеки JS, включаючи React. Навіть спочатку нові функції, такі як тіньова DOM, яка інкапсулює стилі, можна реалізувати за допомогою альтернативних методів, таких як модулі CSS або CSS в JavaScript, як ми будемо досліджувати далі в модулі стилів. Наразі стало зрозуміло, що в осяжному майбутньому найпоширенішим варіантом використання веб-компонентів буде як основа для таких бібліотек, як Angular, Ember тощо.

1.3. React

Наразі ми виключили веб-компоненти, але сьогодні існують інші переконливі способи створення компонентів. Якщо ми створюємо багаторазові компоненти, пов’язані з бібліотекою або фреймворком, найпопулярнішими варіантами сьогодні є React, Angular, Ember і Vue. Звичайно, є й інші варіанти, які варто розглянути, але це основні гравці. То чому ж ми повинні вибрати створення компонентів для багаторазового використання в React? Очевидно, якщо ми хочемо, щоб люди використовували наші компоненти React, вони повинні використовувати React. Тепер React досить легкий і бездумний, щоб його можна було змішувати з іншими технологіями. Тим, хто працює в альтернативних бібліотеках, доведеться погодитися на те, щоб залучити іншу бібліотеку. І хоча в React за останні пару років було відносно небагато значних змін, вона, як і будь-яка інша бібліотека, буде розвиватися з часом. React — це легка бібліотека. На відміну від інших великих фреймворків, React зосереджений на дуже чіткій проблемі - компонентах, які можна використовувати повторно. Це означає, що ми можемо використовувати React у застарілих програмах як підхід для повільного переходу до сучасного інтерфейсу, відтвореного на стороні клієнта. Насправді це саме те, що Facebook зробив з React. Це PHP-магазин, який повільно інтегрує React у свою програму по одному компоненту. React — це найпривабливіша компонентна платформа, доступна сьогодні.

1.4. Огляд рішень

1.4.1. Аудиторія

Перше – це наша аудиторія. Найпростіша аудиторія для обслуговування – це один проект. У цьому сценарії ми знаємо команду, з якою працюємо, і наша мета — просто розробити компоненти, які можна використовувати в кількох місцях в межах однієї програми. Таким чином, наші багаторазові компоненти забезпечують узгодженість сторінок. Або, наша команда планує використовувати ці компоненти для кількох проектів, створених вашою командою? Можливо, у нас є більш грандіозне бачення створення бібліотеки компонентів для всього бізнесу або для використання в кількох бізнес-підрозділах. Звичайно, найважче обслуговувати громадськість. Розробники люблять думати про повторне використання. Легко мріяти про вирішення проблеми один раз для всіх, але визнаємо, що коли ми рухаємося до вирішення проблеми, наша робота стає набагато складнішою. Збільшений обсяг також збільшує навантаження на дизайнера, оскільки потреби в налаштуванні зростають із зростанням аудиторії. Тому ми будемо фокусуватися на одному проекті.

1.4.2 Жорсткі та гнучкі компоненти

Створюючи багаторазові компоненти, ми стикаємося з другим основним рішенням. Компоненти повинні бути жорсткими чи гнучкими? Звичайно, тут немає правильної відповіді. Коли ми створюємо багаторазові компоненти, користувачі хочуть зробити багато елементів гнучкими на основі контексту. Кожен компонент існує в спектрі від жорсткого до налаштовного. Жорсткий, мається на увазі, що він не дуже гнучкий або настроюється. Жорсткий компонент має кілька реквізитів для налаштування його зовнішнього вигляду та поведінки. При розгляді дизайну компонентів може здатися очевидним, що

гнучкість краще, ніж жорстка, але не настільки швидка. Історія не така проста. Насправді, жорсткі компоненти мають багато переваг. Жорсткі компоненти, як правило, простіше створювати та обслуговувати. Чому? Тому що менше випадків використання і менше коду. Це означає, що для створення жорсткого компонента потрібно менше часу, а також для його налаштування. Жорсткі компоненти часто легше зрозуміти, оскільки їх використання має тенденцію бути конкретним і зрозумілим. Їх легше перевірити, тому що в них задіяно менше коду і менше граничних випадків, які потрібно розглянути. Це в кінцевому підсумку призводить до менших робіт з технічного обслуговування. Жорсткий компонент дуже упевнений. Якщо наша мета — створити узгоджену взаємодію з продуктами, жорсткі компоненти допомагають програмно забезпечити цю узгодженість. Вони допомагають уникнути втоми від прийняття рішень, але усувають набір рішень і програмно забезпечують їх. Нарешті, реквізити – це як функції. Легше додати більше реквізитів пізніше, щоб підвищити гнучкість, але насправді важко видалити реквізити пізніше, оскільки це порушить існуючі варіанти використання. Тому з усіх цих причин при розробці компонентів ми почнемо з більш жорсткого компонента, а потім лише додамо гнучкості, оскільки це очевидно необхідно та виправдано. Крім того, потрібно подумати, коли має сенс створювати новий компонент, а не додавати додаткові налаштування до наявного компонента. Два окремих жорстких компонента можуть бути кращими, ніж один компонент, що налаштовується, оскільки два жорстких компонента можуть продовжувати виконувати дуже специфічні вимоги. Тепер, звичайно, очевидна перевага розробки більш гнучкого компонента полягає в тому, що він, швидше за все, буде повторно використовуватися в більш широкому наборі контекстів. Це велика перемога, особливо коли ми створюємо компоненти для широкої публіки. Ми завжди можемо створити два окремих компонента, які виконують дещо різні речі та використовують загальний код за кадром. Це дає нам усі переваги жорсткості, одночасно обслуговуючи два різні варіанти використання. Але кожного разу, коли ми створюємо компоненти для багаторазового

використання, наша база користувачів посилить їх, щоб зробити компоненти більш гнучкими, тому потрібно мати на увазі цей компроміс.

1.4.3. Зв’язати, обгорнути чи розширити?

У мережі вже є сотні компонентів React. Уявіть, що ми знайшли в Інтернеті чудовий компонент, який уже виконує те, що нам потрібно. Тепер нам належить прийняти рішення. Чи варто посилатися на нього? Загорнути це? Або зробити розширення? Розглянемо переваги недоліків кожного з цих підходів. Перший і найпростіший підхід — просто вказати посилання на нього у своїх документах. В основному ми говоримо людям, що ми не маємо вбудованої підтримки цієї функції, але ось схвалений компонент. Ось чому ми можемо обернути компонент. Це дає нам можливість абстрагувати компонент і покращувати його за допомогою додаткових функцій або приховувати функції, які не стосуються нашого випадку використання або мають бути завжди встановлені на певне значення. Розширення — це найбільш налаштований підхід. На той момент ми повністю володіємо кодом, тому можемо змінювати його, як вважаємо за потрібне. Звісно, мінусом є те, що з цього моменту ми самостійно займаємося обслуговуванням. Розглянемо переваги кожного підходу. Посилання на існуючий компонент, очевидно, є найпростішим, що ми можемо зробити. Завдяки такому підходу, коли ми знаходимо компонент, який хочемо зробити частиною своєї бібліотеки, ми насправді не робитимемо його частиною своєї бібліотеки взагалі. Замість цього ми просто додаємо посилання на свої документи і, можливо, короткий опис запропонованого компонента. Цей підхід має ряд очевидних недоліків. По-перше, нам не потрібно створювати документацію, що є плюсом. Але ми також повністю покладаємося на сторонні документи. Це створює роз’єднаний досвід, оскільки у наших користувачів немає єдиного місця, де можна було б знайти інформацію про компоненти, які ми рекомендуємо. Ми не можемо контролювати, коли відбуваються зміни в цих компонентах, тому ми повинні просто спостерігати за сховищем, щоб знати, як

компонент змінюється з часом. Це означає, що в будь-який момент компонент може змінитися таким чином, що ми не зможемо прийняти. Звичайно, на цьому етапі ми завжди можемо вибрати перехід до іншого компонента або замість нього обгорнути чи розширити. Якщо нам подобаються нові функції, ми, звичайно, зможемо користуватися ними безкоштовно. Зі свого боку нам немає роботи. Однак, якщо ми хочемо будь-яким чином налаштувати компонент, нам не пощастило, оскільки ми насправді не керуємо кодом. Ми не можемо істотно змінити зовнішній вигляд, але багато компонентів надають певний рівень параметрів стилів залежно від того, наскільки компонент покладається на вбудовані стилі, які ми не можемо змінити, або забезпечити вбудовану підтримку тем. Усе це призводить до дуже низького рівня згуртованості. Компонент насправді не здається, що він є частиною нашої бібліотеки компонентів, тому що це не так. Це справді лише пропозиція. Але це означає, що наші зобов'язання дуже низькі. І ми можемо в будь-який момент переїхати в інше місце, якщо один із вищенаведених недоліків стане неприйнятним. І, звичайно, найбільша перемога з таким підходом – це легко. Просто вставляємо посилання в свої документи і рухаємося далі. Але через багато недоліків, наведених вище, ми, швидше за все, віддамо перевагу обгортанню або роздвоєнню. Другий варіант — обгорнути сторонній компонент. Ось приклад обгортання компонента. Уявіть собі, що ми знайшли в Інтернеті чудовий DatePicker під назвою SuperDate. Ми можемо обгорнути його, імпортувавши компонент і, таким чином, приховати той факт, що ми використовуємо DatePicker з таким дурним ім'ям. Усі наші споживачі тепер знають, що наш DatePicker називається DatePicker. Звичайно, у рендері, ми можемо вирішити, які реквізити ми хотіли би передати в SuperDate. Ми можемо взагалі змінити назви реквізитів. І ми можемо вирішити, які реквізити на SuperDate ми хочемо виставити. Ми навіть можемо жорстко кодувати стилі, щоб зробити SuperDate ще більше, ну, супер. Таким чином, обгортаючи існуючий компонент, ми отримуємо багато переваг від створення власного компонента з нуля без необхідності робити це. Обгортання наявного компонента — це хороша золота

позиція. Ми можемо створити документацію, завдяки якій компонент буде повністю інтегрованим у нашу бібліотеку. А коли в базовому компоненті відбуваються зміни, ми завжди можемо вирішити, коли приймати ці нові версії, а також чи відкривати нові функції. Обгортаючи компонент, ми маємо можливість налаштувати досвід деякими ключовими способами. Ми можемо покращити явні функції, приховати інші, які не застосовуються, додати думки та абстрагувати складні елементи для своїх користувачів. Наша здатність до стилю компонента буде відрізнятися в залежності від того, наскільки компонент покладається на вбудовані стилі та чи забезпечує він підтримку тем. І рівень згуртованості, який забезпечує цей підхід, значною мірою залежить від того, наскільки ми можемо налаштувати зовнішній вигляд компонента відповідно до нашої бібліотеки, просто обгорнувши його. Оскільки ми не можемо змінити базовий код, ми можемо зайти лише так далеко. Але наша прихильність тут все ще досить низька. Створення обгортки, яка відкриває компонент за вашим вибором, і створення певної документації – це дійсно невелика ціна за отримання цієї додаткової гнучкості. Що стосується загальної складності, то це більше роботи, ніж створення зав’язків, але, як правило, менше роботи, ніж розширення репозиторію та володіння кодом самостійно. Тепер, якщо ми знайдемо компонент, який нам дійсно подобається, може бути найбільш сенсом роздвоїти його і зробити його повністю власним. Forking означає, що ми створюємо копію бази коду та керуємо нею самостійно. Зробивши це, ми можемо повністю інтегрувати компонент у свою бібліотеку. Ми можемо створити інтегровану документацію. Ми маємо повний контроль над будь-якими змінами, незалежно від того, як змінюється початковий проект з часом. Однак це означає, що нові функції стають нашою відповідальністю. Коли оригінальний компонент отримує нову функцію, може бути непрактично інтегрувати цю функцію у наш компонент. Роздвоєні кодові бази можуть швидко розходитися в кардинально різних напрямках. Але ми маємо повний контроль, щоб налаштувати компонент так, як вважаємо за потрібне, а також стилізувати компонент, як нам подобається, включаючи надання дуже

впевнених вбудованих стилів, якщо хочемо. Це призводить до бібліотеки компонентів з високим рівнем згуртованості. Здається, що все поєднується і задокументовано в одному місці. Однак ці переваги вимагають більш високого рівня відповідальності, оскільки, очевидно, це більше роботи, ніж просто зв’язування із запропонованим компонентом або упаковка існуючого проекту. Коли ми намагаємося зробити вибір: зв’язати компонент, обгорнути чи розширити, ось кілька запитань, які потрібно задати: чи достатньо це добре, як просто зв'язати? Чи підходить це нашій моделі? Якщо люди використовують його з коробки, чи буде компонент виглядати так, ніби він підходить для наших програм? Чи практично перевизначати стилі компонента? Чи активно підтримується проект? Чи дасть нам обгортання компонента достатньо потужності? Коли ми рухаємося до вирішення проблеми, контроль підвищується, але також зростає зусилля та відданість.

Висновок до розділу 1

Ми заклали міцну основу для багаторазового дизайну компонентів. Ми почали з того, що поставили важливе перше запитання: чому взагалі варто використовувати багаторазові компоненти? Ми розглянули основні переваги багаторазового дизайну компонентів, включаючи швидкість розробки, узгодженість і продуктивність. Ми розглянули метафору автомобільних деталей, які базуються на простих деталях, які складаються разом у великі вузли. Ми розглянули альтернативи React, такі як стандарт веб-компонентів, і побачили низку конкретних причин того, що React сьогодні є відмінним вибором для розробки компонентів багаторазового використання. Нарешті, ми розглянули чотири основні рішення. Вибрали аудиторію і почнемо з одного проекту. Вирішили проблему для свого конкретного проекту, перш ніж намагатися працювати з більшою аудиторією, як інші команди або широка громадськість. Вирішили, що будемо створювати жорсткі компоненти, тому що їх простіше відтворити і протестувати. Запам'ятали правило трьох і спробуємо

компоненти у трьох різних місцях, перш ніж помістити його в бібліотеку компонентів для багаторазового використання. Таким чином, ми можемо бути впевнені, що він добре розроблений для повторного використання в багатьох різних контекстах.

РОЗДІЛ 2. СТРУКТУРА ТА ВИБІР ТЕХНОЛОГІЙ

2.1. Бібліотека проти автономних компонентів

Перше ключове рішення полягає в тому, що нам потрібно вирішити, хочемо ми створити цілісну бібліотеку компонентів, чи краще створити окремі компоненти. Існують десятки прикладів обох стратегій, тому розглянемо кілька прикладів кожної з них. Material UI — популярна бібліотека, яка пропонує набір компонентів React, які реалізують рекомендації Google щодо дизайну матеріалів. React Bootstrap містить компонент React, який полегшує роботу з Bootstrap, і є Blueprint, який має широкий набір компонентів інтерфейсу користувача з послідовним і сучасним зовнішнім виглядом, але є також чимало окремих компонентів, таких як react-dates, який є інструментом вибору дати React від AirBnB, і Griddle, який є популярною табличною сіткою для React, яка включає в себе вбудовану пагінацію та фільтрацію, або React DnD, який являє собою набір компонентів, які забезпечують підтримку перетягування. Тож незалежно від того, створюємо ми компоненти для внутрішнього або зовнішнього використання, переваги мають як бібліотека, так і підхід окремих компонентів. Розглянемо конкретні переваги кожного підходу. За допомогою бібліотечного підходу набір пов’язаних компонентів React об’єднано в єдиний пакет NPM, тому найочевиднішою перевагою бібліотечного підходу є узгодженість. Як правило, бібліотека компонентів забезпечує певний вигляд і відчуття, і вона пропонує послідовний API для всіх своїх компонентів. Коли весь код знаходиться в одному репозиторії, легше дотримуватися однакових

шаблонів для всіх компонентів. Ми можемо запропонувати набір компонентів, які мають загальний вигляд і відчуття, а також використовувати єдиний набір стилів. Це означає, що компоненти можуть бути тематичними за допомогою централізованих стилів, і клієнт може відчувати себе більш впевненим, що створений ним додаток в кінцевому підсумку відчує згуртованість. Бібліотечний підхід полегшує додавання нового компонента, оскільки всі основи вже створені для нас. Уже є репозиторій, пакет NPM, структура документації тощо. Все, що нам потрібно зробити, це додати новий компонент, який вписується в існуючу структуру. Це також легше організувати. Єдина бібліотека допоможе нам дотримуватися сухого принципу, який означає «не повторюйся». Нам не потрібно керувати кількома пакетами NPM. Репозиторіями, збірками, тестовими конфігураціями, процесами розгортання тощо. Коли ми створюємо цю основу, як тільки це буде зроблено, ми зможемо використовувати її для всіх майбутніх компонентів. Це означає, що ми можемо швидко опублікувати нові компоненти без особливого тертя. Ми можемо скористатися всією наявною інфраструктурою, яку ми вже налаштували. Коли ми створюємо більше компонентів для багаторазового використання, ми, ймовірно, виявимо, що вони будуються один на одному. Ми створимо кілька невеликих простих компонентів, щоб створити щось більш складне. Наявність цих компонентів в одній бібліотеці зменшує витрати на керування залежностями, оскільки нам не потрібно керувати посиланнями на декілька окремих пакетів під час складання компонентів разом. Це також більш доступне для виявлення. Коли хтось знає про наш проект, йому набагато легше дізнатися про нові компоненти, які ми створили. Якщо наші компоненти окремі, то нам потрібно озвучувати кожен новий компонент, тому бібліотеку легше продати, тому що нам потрібно озвучити лише один раз, а не постійно рекламувати ще одну нову річ. Завдяки моноліту кожен новий компонент автоматично стає доступним для всіх, хто використовує наш пакет, у момент оновлення. Коли нам потрібно внести серйозні зміни до існуючого компонента, легше оновити будь-які місця, де він складається, іншими компонентами

вищого рівня. Припускаючи, що у нас є надійна історія з підготовкою та тестуванням, ми можемо спертися на свій інструмент і бути впевненим, що ми усунули всі плями. Якщо у нас багато окремих сховищ, ця робота стає складнішою. Нарешті, навіть якщо ми використовуємо бібліотечний підхід, люди все одно можуть використовувати лише те, що вони хочуть. Якщо ми публікуємо свої компоненти стратегічно, користувачі можуть просто імпортувати окремі компоненти, не змушуючи своїх користувачів завантажувати всю бібліотеку. Підхід до бібліотеки виглядає як безглуздя, але, безумовно, є деякі переваги створення окремих компонентів. По-перше, побудувати його легше, ніж бібліотеку. Нам не потрібно занадто багато думати про велике, цілісне бачення чи мову дизайну. Ми можемо просто показати корисний компонент і дозволити людям зняти його з полиці, якщо їм це подобається. Це полегшує прийняття. Користувачам не потрібно переглядати та розглядати всю нашу бібліотеку. Вони можуть захопити один компонент, який їм потрібен, і не турбуватися про потенційне роздуття свого додатка невикористаним JavaScript і стилями. Окремі компоненти вимагають низьких зобов’язань як для автора, так і для споживача. Замінити один компонент легко, але використання цілої бібліотеки компонентів — це велике рішення, яке має довгострокові наслідки, і, отже, його важко змінити пізніше. Окремі компоненти зручні для розробників, які вважають за краще знаходити найкращі компоненти для певного завдання та змішувати і поєднувати компоненти разом, а не змушені приймати цілу бібліотеку, яка може мати деякі слабкі компоненти всередині. Окремий компонент може бути більш гнучким, оскільки йому не потрібно забезпечувати узгодженість у бібліотеці компонентів, а також рідше змінювати версію. З бібліотекою ми повинні змінювати всю версію бібліотеки щоразу, коли будь-які компоненти змінюються, але якщо кожен компонент знаходиться в окремому сховищі, ми можемо змінити версії кожного окремо. Це може полегшити життя споживачам, оскільки вони рідше зазнають зміни версії для певного компонента, тому обидва ці підходи мають переваги, але важливо враховувати ці переваги наперед, щоб ми могли вибрати

підхід, який найкраще відповідає вашим потребам. А що робити, якщо ми хочемо отримати переваги інтеграції від одного репозиторію, але також хочемо версії окремих пакетів? Ми все ще можемо керувати кількома пакетами з одного сховища вихідного коду. Тут може допомогти Lerna. Lerna — це інструмент для керування проектами JavaScript з кількома пакетами всередині. За допомогою Lerna ми можемо створити єдине сховище, але в цьому репозиторії може бути розміщено кілька пакетів NPM. Таким чином ми можемо зберігати всі проекти всередині окремо. Lerna пов’язує перехресні залежності в межах одного репозиторію, щоб ми могли тестувати зміни в одному пакеті в усіх наших пакетах. Наприклад, у нас може бути компонент, який залежить від інших компонентів, і коли ми змінимо один компонент, Lerna дозволить нам підтвердити, що ми не зламав жодних залежних компонентів. Lerna використовується деякими з найпопулярніших проектів JavaScript у світі, включаючи Babel і програму create-react-app від Facebook.

2.2. Середовище розробки

2.2.1. Шаблон

Перше ключове рішення щодо середовища розробки полягає в тому, чи варто вибрати існуючий шаблон, проект стилю бібліотеки документації або створити власне середовище розробки. Є три основні способи. По-перше, ми можемо вибрати існуючий загальний шаблонний проект, орієнтований на розробку додатків, а потім налаштувати його для роботи з бібліотекою компонентів, або ми можемо розглянути довгий список проектів інструментів документації, які забезпечують вбудований досвід розробника швидкого зворотного зв’язку. Нарешті, звичайно, ми можемо розглянути можливість створення власного середовища розробки. Перший шаблон, який ми розглянемо, це create-react-app. Ось кілька причин. Create-react-app є найпопулярнішим шаблонним проектом React на сьогодні з понад 24 000 зірок

на GitHub. Це єдиний шаблонний проект react, який офіційно підтримується командою Facebook, і create-react-app об’єднує найпопулярніші технології для розробки React сьогодні разом, включаючи Babel, Webpack, ES Lint і Jest. Він добре перевірений і ретельно розроблений, щоб гарантувати, що він працює на різних платформах, і його легко оновити, оскільки він абстрагує всі свої залежності та сценарії за пакетом NPM, і ми можемо видалити проект у будь-який момент, щоб отримати повну можливість налаштування. Тож які мінуси? Що ж, спочатку ми повинні вимкнути для повної конфігурації його базові інструменти, такі як Webpack, а після вимкнення ми змушені обробляти оновлення вручну, тому немає автоматичного способу залишатися в курсі, оскільки нові функції та виправлення помилок будуть випущені в майбутньому, коли ми зробимо eject. Так що тут є справжній компроміс. Нарешті, ми можемо не погодитися з думками, такими як Babel з TypeScript, Webpack over Rollup або Browserify, або Jest of Mocha, jasmine тощо, але в кінцевому підсумку програма create-react-app дуже популярна з поважної причини. Вона добре продумана, добре розроблена і дуже швидко дає можливість розпочати роботу. NWB є ще одним переконливим шаблоном для розробки компонентів багаторазового використання. Подібно до програми create-react-app, NWB об'єднує Node, Webpack і Babel в єдиний упевнений пакет NPM. Так чому саме NWB? Що ж, NWB — це переконливе середовище розробки, оскільки воно призначене не лише для додатків. Він може автоматично налаштувати середовище розробки, спеціально розроблене для створення середовища багаторазового використання компонентів. Він має багато вбудованих корисних функцій, включаючи тестування через Mocha і Karma, він використовує Babel для транспілювання нашого коду в ES5, і ми можемо за бажанням увімкнути збірку імпорту ES, яка залишає наші оператори імпорту ES, щоб користувачі могли виконувати струшування дерева у нашій бібліотеці, щоб допомогти зменшити розміри пакетів. Нарешті, він також включає додаткову збірку UMD, що означає універсальне визначення модуля. Цей тип збірки корисний, якщо ми хочемо показати свою бібліотеку як глобальну змінну, щоб її можна було

використовувати з онлайн-редакторами, такими як JSBin або JSFiddle, або вставляти на наявну сторінку, яка не використовує модулі React або ES, тож це багато вагомих причин використовувати NWB, але це не для всіх. Найважливіша відмінність полягає в тому, що NWB, здається, припускає, що ми публікуємо один компонент, а не бібліотеку компонентів. Тепер, щоб уточнити, ми, безумовно, можемо опублікувати бібліотеку з NWB, але її початкова конфігурація призначена для публікації одного компонента. По-друге, NWB розшифровується як Node, Webpack і Babel, тому в нього є вбудовані тверді думки. NWB припускає, що ми віддаємо перевагу Babel над TypeScript. Ми віддамо перевагу Webpack перед альтернативними пакетами, такими як Rollup або Browserify. нарешті, він використовує Mocha та Karma для тестування альтернативних бібліотек, таких як Jest, тож якщо нам не подобаються ці думки, очевидно, NWB не для нас, але це, безумовно, швидкий спосіб розпочати роботу, який просто працює.

2.2.2. Інструмент документації

Замість того, щоб вибирати загальний шаблон, ми можемо вибрати те, що називають середовищем розробки стилю документації. Деякі люди називають цей підхід Керівництвом по стилю. Існує багато існуючих бібліотек на вибір, і всі вони пропонують подібний набір основних функцій. Опції включають React Storybook, React Styleguide, React Styleguidist, Atelier, Bluekit і Cosmos. Коротко розглянемо кожен. Коли ми створюємо багаторазові компоненти React, середовище розробки стилю документації має великий сенс. Чому? Оскільки наша документація — це в основному наш додаток, тому ваше рішення щодо документації повністю впливає на наш потік розробки. Наприклад, ми можемо задокументувати свої компоненти за допомогою React Storybook, щоб ми могли запускати свої компоненти, побачити їх у дії та перевірити різні перестановки реквізитів. React Storybook наразі є найпопулярнішим інструментом React doc. Його просто та легко інтегрувати з існуючим проектом, він надає список

компонентів з можливістю пошуку зліва та інтерактивний попередній перегляд різних прикладів праворуч. React Styleguidist пропонує подібний досвід React Storybook, але додає можливість бачити фрагмент коду, який використовується за лаштунками. На жаль, фрагмент коду не містить усього необхідного контексту, як оператор імпорту, але навіть у цьому випадку це приємний штрих. Цей проект використовує react-docgen за кадром, який ми розглянемо в модулі документації пізніше, і він включає автоматичне тестування за допомогою тестової рамки Ava. Дуже схожий проект із дуже схожою назвою — генератор React Style Guide. Він також відображає список компонентів, доступний для пошуку, ліворуч, живі демонстрації кожного та приклад коду нижче. Atelier — це іменник, що означає кімнату, де працює художник, тому насправді це досить гарна назва для цього. На відміну від попередніх інструментів, Atelier зосереджується на більш інтерактивному стилі документації. Він інтелектуально зчитує всі наші компоненти реквізитів і надає вхідні дані для перемикання кожного значення на основі типу опори. Цей інтерактивний підхід до ігрового майданчика дійсно веселий, і це глибокий спосіб для людей познайомитися з вашою бібліотекою компонентів. BlueKit використовує подібний підхід. Як і Atelier, він забезпечує інтерактивний майданчик для наших компонентів; однак він також читає наші коментарі, щоб створити документацію prop, і показує повний приклад використання нижче. Нарешті, Cosmos описує себе як інструмент для розробки багаторазових компонентів React, але ми також можемо використовувати його як інтерактивну форму документації. Особливим Cosmos є те, що він відображає інтерактивне вікно коду зліва, і все, що ми змінюємо у вікні коду, відображається в попередньому перегляді праворуч, тому в цьому просторі є багато переконливих варіантів, і всі вони забезпечують цікаву суміш готові середовища розробки разом з інтерактивною документацією. Усі ці інструменти надають список наших компонентів із можливістю пошуку та дають чіткий шаблон для створення та відображення робочих прикладів. Вони

забезпечують підхід для документування наших реквізитів, і багато з них надають фрагменти коду того, як приклади працюють за лаштунками.

2.3. Документація

2.3.1. Цілі документації

Під час створення компонентів для багаторазового використання наші документи є вашим середовищем розробки. Зараз це звучить дивно, але якщо подумати, це рішення цілком очевидно. Коли ми створюємо додаток, ми бачимо, як наші компоненти React дійсно працюють, але коли ми створюємо бібліотеку компонентів для багаторазового використання, нам все одно потрібен спосіб побачити наші компоненти в дії, тому має сенс використовувати нашу документацію, як середовище розробки. По суті, наша документація — це наша програма. В іншому випадку ми в кінцевому підсумку повторимо свою роботу, підключивши свої компоненти в одному місці для цілей розробки, а в іншому місці для документації. Це не добре, і всі популярні бібліотеки компонентів React використовують цей підхід, включаючи Material-UI, React Toolbox, React Bootstrap, Blueprint та багато інших. Насправді, ми можемо зайти на GitHub і розгорнути будь-який з цих проектів і ввести “npm start”. Ми побачимо, що запуститься їхній сайт зі спеціальною документацією, щоб ми могли працювати з документами та бачити свої багаторазові компоненти в дії. Документація повинна відображати опис компонента, вона повинна задокументувати реквізити компонентів, включаючи їх тип, опис, значення за замовчуванням і те, чи потрібний prop. Він також повинен відображати робочі приклади, включаючи код, який створює приклад, і ось цікава частина. Наша мета — автоматично генерувати більшість того, що ми пишемо. Це потужний інструмент. Тепер до коду багаторазового компонента слід ставитися так само суворо, як і до коду програми, тому, коли ми

створюємо багаторазові компоненти, важливо переконатися, що всі мають спільне бачення.

2.3.2. Навіщо створювати документацію?

Просте правило щодо документації: не створюйте її вручну. Ми розробники. Нам слід використати наші навички, щоб автоматизувати цю проблему, забезпечити послідовність і переконатися, що наші документи залишаються синхронізованими з поточним кодом компонентів. Так чому ж варто по працювати над створенням документації, читаючи вихідний код? По-перше, це гарантує, що документи синхронізовані з кодом. Якщо ми створюємо документи вручну, це означає, що ми повинні пам’ятати, що потрібно вручну оновлювати документи щоразу, коли ми змінюємо код, і будь-яке місце, де ми очікуємо, що розробник зробить щось вручну, є шансом на помилку. Автоматизація економить час і покращує якість. Коментарі, необхідні для створення документів, також корисні під час читання коду. Це означає, що повсякденні кодери також отримують користь від коментарів, які ми збираємося створити тут, щоб допомогти створити наші документи. Нарешті, ми можемо бути більш впевнені в тому, що створена документація узгоджена, оскільки вона створюється автоматично. Ми можемо створити структуру пов’язаних компонентів для відображення нашої документації, і таким чином дизайн буде заблокований в одному місці. Це дозволяє уникнути очікування від розробників, що пам’ятають, як слідувати певним шаблонам для документації, тому ми створимо структуру для автоматичного генерування нашої документації компонентів, і ми будемо використовувати її, створюючи власні компоненти.

2.3.3. Інструменти документації

Існує багато інструментів, орієнтованих на документи, які допомагають створювати документацію компонентів. Можна було б переглянути ці інструменти, щоб побачити, чи відповідають вони нашим потребам, однак ми не збираємося використовувати жодне з них, тому що, чесно кажучи, усі популярні бібліотеки компонентів React створюють власну документацію. Тож замість цього ми будемо створювати власну документацію компонентів за допомогою React та кількох інструментів з відкритим кодом. Створення користувальницьких документів не так велика робота, і це дає нам повний контроль, щоб адаптувати налаштування відповідно до наших потреб. Ми можемо автоматично створювати документи під час додавання нового компонента.

2.3.4. Підхід до створення документації

Ми збираємося налаштувати фреймворк для документування наших компонентів для повторного використання під час їх написання. Все, що ми бачимо тут, буде автоматично згенеровано шляхом читання наших компонентів, і оскільки ми збираємося створювати саму документацію за допомогою компонентів React, ми матимемо повну можливість налаштувати це будь-яким способом. На навігаційній панелі ліворуч відобразиться список наших компонентів для багаторазового використання, відсортованих за алфавітом. Коли ми клацнемо на компоненті, ми побачимо Component Page з усіма відповідними деталями, включаючи опис зі списком прикладів і точний код, який використовується для створення прикладу коду, разом із виділенням синтаксису. Унизу він відобразить таблицю з документацією всіх наших компонентів. Щоб це сталося, ми збираємося згенерувати деякі метадані про кожен з наших компонентів, використовуючи зручну бібліотеку під назвою react-docgen, а також кілька інших бібліотек і пакетів npm. Потім ми

використаємо ці дані для створення документації. Для створення власних документів ми будемо використовувати ці пакети npm. Найважливішим є react-docgen. React-docgen може читати компонент React і витягувати важливі метадані компонента, такі як опис компонентів і реквізити. Він прочитає як наші коментарі, так і сам код для створення цих даних. Chokidar — це багатоплатформний спосіб перегляду файлів, тому ми можемо виконувати команду будь-коли, коли файл змінюється. Таким чином ми можемо відновлювати метадані наших компонентів щоразу, коли ми натискаємо «Зберегти» і бачимо, що вони негайно відображаються в нашій програмі. highlight.js забезпечить підсвічування синтаксису для наших прикладів коду. Material-UI також використовує цю бібліотеку, щоб додати забарвлення коду до своїх прикладів коду, а щоб спочатку запустити сценарій генерування даних компонента, який ми збираємося написати, ми будемо використовувати npm-run-all. Цей пакет пропонує дружній міжплатформенний спосіб паралельного запуску наших скриптів. Іншими словами, водночас. react-docgen є основою нашого підходу до документації. Ця бібліотека використовується за лаштунками деякими з найпопулярніших бібліотек компонентів React, включаючи Material-UI. React-docgen генерує метадані, необхідні для відображення деяких даних компонентів. Ось приклад react-docgen в дії. Ми можемо додати коментар над компонентом як опис і коментар над кожним атрибутом, щоб описати кожну властивість. Тепер react-docgen прочитає як наші коментарі, так і сам код, щоб створити цей дійсно корисний фрагмент JSON. Він зчитує назву компонента, коментар над компонентом заповнюється як опис, а також генеруються метадані для трьох опорних елементів. Він навіть читає реквізит за замовчуванням, який ми оголосили у верхній частині, і вказано як частину висоти опори, а також типи даних для кожного опору, і чи є він потрібним, тож це дає нам майже всі дані, які нам потрібно створити багату документацію компонентів. Наприклад, Code Example посередині екрана показує фактичний код, який був використаний для створення прикладу, який відображається. Це дійсно корисно, тому користувачі можуть чітко зрозуміти

повну картину необхідного імпорту та значень параметрів, необхідних для отримання відображених результатів. Щоб це сталося, це двоетапний процес. Спочатку ми напишемо один або кілька робочих прикладів для нашого компонента, а потім використаємо Node для читання файлів прикладів, які ми створюємо, і введемо цей вміст у структуру даних, повернену з react-docgen. Тоді ми матимемо всі метадані, необхідні для автоматичного створення всього необхідного, щоб відобразити на екрані всі характеристики кожного компонента.

2.3.5. Підсвічування синтаксису

Ми збираємося відображати код у браузері. Існує два популярних способи додавання підсвічування синтаксису в браузері, Highlight.js і Codemirror. Highlight.js є простішим із двох, оскільки він зосереджений виключно на додаванні підсвічування синтаксису до коду. Він підтримує різні мови, і, на щастя, однією з них є JavaScript. Він чудово виглядає і пропонує десятки кольорових схем на вибір. Codemirror також забезпечує забарвлення коду, але він робить набагато більше. Codemirror насправді підтримує редагування виділеного синтаксису коду, тому ми можемо використовувати Codemirror, щоб створити веб-редактор коду. Це дуже гарні демонстрації, і Codemirror — це те, що React Toolbox використовує за лаштунками. Ми будемо використовувати highlight.js, оскільки редагування в реальному часі — це набагато більше роботи для ефективного налаштування, і редагування коду є зайвим для більшості проектів. Тепер у репозиторії є навіть зручний компонент React, який називається react-highlight, який огортає highlight.js.

2.4. Багаторазовий дизайн компонентів

2.4.1. Слабкі елементи

Принцип перший: уникнення слабких елементів обгортки. Наприклад в тег абзацу не можна розміщувати div, тому потрібно уникати використання слабких елементів DOM, як абзац, всередині компонентів, які в кінцевому підсумку можуть обернути інший вміст. Це призведе до недійсної розмітки HTML. Натомість потрібно віддавати перевагу більш гнучким контейнерам, таким як div.

2.4.2 PropTypes

Принцип другий: оголошення PropTypes. PropTypes особливо корисні для багаторазових компонентів, тому нам потрібно послідовно визначати PropType для всіх наших компонентів, і застосовувати їх за допомогою linting. PropTypes уточнюють API нашого компонента. Вони чітко визначають наші очікування щодо реквізитів, включаючи типи даних та обов'язкові поля, а коли трапляються недійсні чи відсутні типи реквізитів, React генерує попередження під час виконання. Це допомагає пришвидшити розробку та уникає трудомістких сеансів налагодження пізніше. Використання PropTypes дає нам деякі переваги сильного введення з дуже невеликими витратами. Нарешті, сучасні редактори, такі як WebStorm, навіть достатньо розумні, щоб забезпечити підтримку автодоповнення, аналізуючи наші PropType, тож ми можемо заощадити користувачів від введення тексту та швидше попередити їх про помилки. І, звичайно, ми також використовували PropTypes для створення нашої документації. Тепер, як примітка, ми можемо розглянути Flow. Команда React також пропонує потік для додавання статичної перевірки типів у наш JavaScript. Ми потенційно можемо використовувати Flow як альтернативу PropTypes, але, звичайно, всі інструменти, які ми налаштували в модулі

документації, припускають, що ми використовуєте PropTypes, а PropTypes залишаються одним з найпопулярніших і основних способів обробки компонентів, які можна використовувати повторно, тому ми будемо дотримуватись традиційних PropTypes для наших компонентів багаторазового використання.

2.4.3. HTML-ідентифікатори жорсткого коду

Принцип третій: не потрібно кодувати HTML-ідентифікатори жорстко. Наші компоненти використовуватимуться в кількох місцях, часто кілька разів на одній сторінці, тому важливо переконатися, що ми не вказуємо жорсткий код HTML-ідентифікаторів у розмітці. Ідентифікатори HTML мають бути унікальними на сторінці, тому використання ідентифікаторів HTML у компоненті для повторного використання проблематично з двох причин. По-перше, будь-який HTML-ідентифікатор, який ми виберемо, може конфліктувати з іншою розміткою на сторінці, де споживач використовує наш компонент, але що важливіше, розміщення статичного HTML-ідентифікатора у нашому компоненті для багаторазового використання створює потенціал для недійсних повторюваних ідентифікаторів, якщо споживач використовує кілька екземплярів. нашого компонента на тій же сторінці.

2.4.4. Логічні параметри за замовчуванням

Принцип четвертий: потрібно оголошувати логічні параметри за замовчуванням. Продумані параметри за замовчуванням роблять наш компонент доступнішим і зменшують обсяг роботи з налаштування, тому ретельно подумайте про логічні параметри за замовчуванням, які ми можемо оголосити. Хороші налаштування за замовчуванням рятують наших споживачів від введення тексту, оскільки їм не потрібно явно заповнювати стільки реквізитів. Вони допомагають забезпечити корисну поведінку, передаючи

основний підхід без будь-яких налаштувань, а реквізити за замовчуванням допомагають передати, як працює компонент, показуючи звичайний варіант використання, наповнений реалістичними даними.

2.4.5. Доступність

Принцип п’ятий: потрібно звертати увагу на доступність. Доступність має велике значення. Це обширна тема, і її легко не помітити, коли ми поспішили вкластися в термін, але багаторазові компоненти пропонують потужну можливість, щоб допомогти створити доступність у нашій програмі. Люди не можуть натискати на наші дані. Вони можуть взаємодіяти виключно за допомогою клавіатури, тому важливо реагувати на звичайні введення з клавіатури, як клавіші зі стрілками, і встановлювати фокус за допомогою Tab. Користувачі клавіатури часто використовують Tab для переміщення по вводу, тому переконаємося, що індекси Tab дотримуються та встановлюються в логічному порядку. Порядок табуляції за замовчуванням заснований на позиції DOM, але через стиль позиція DOM може повністю відрізнятися від візуального макета. Встановимо відповідним чином індекси Tab та подумаємо про те, щоб індекс Tab був обов’язковим полем у наших компонентах, пов’язаних із фокусом. Намагаємося використовувати належну семантичну розмітку у своїх компонентах. Наприклад, уникаємо використання div або span як кнопки. Замість цього використовуємо кнопку, оскільки вона має належну поведінку для обробки подій, фокусування та введення з клавіатури. Уникаємо спокуси винайти велосипед, що може призвести до проблем із доступністю. Надаємо мітки для входів, щоб мітка була програмно прив’язана до входу. Це гарантує, що програми зчитування з екрана чітко розуміють, для чого призначено кожне поле. Ми побачимо приклад цього в наступному кліпі, коли ми підключимо атом мітки до текстового введення. Нарешті, дослідимо атрибути ARIA та Role, щоб покращити доступність наших компонентів. ARIA

розшифровується як Accessible Rich Internet Applications. ARIA визначає спосіб зробити веб-контент і веб-додатки більш доступними для людей з обмеженими можливостями.

2.4.6. Розгляд об’єктів конфігурації

Принцип шостий: потрібно розглядати об’єкти конфігурації. Якщо ми проектуємо компонент, який матиме велику кількість опорних елементів або, ймовірно, буде додано кілька опорних елементів у майбутньому, подумайте про те, щоб прийняти об’єкт конфігурації на props. Найбільша перевага полягає в тому, що API компонента не змінюється з часом, коли ми додаємо нові властивості. Він продовжує приймати об’єкт. Об’єкт конфігурації також рятує людей від введення тексту. Якщо у нас уже є об’єкт клієнта, повернутий із виклику API, який має таку саму форму, ми можемо легко передати його без необхідності вручну підключати довгий список окремих реквізитів. Це також робить його менш схильним до помилок. Нам не доведеться турбуватися про те, що випадково введете неправильне значення для неправильного параметра. Якщо форма об’єкта правильна, то компонент повинен працювати належним чином.

2.4.7. Рендеринг на стороні сервера

Принцип сьомий: обов’язково потрібно звернути увагу на рендеринг на сервері. React підтримує серверний рендеринг, тому, якщо ми збираємося публікувати багаторазові компоненти, важливо вирішити, чи хочемо ми підтримувати серверний рендеринг зі своїми компонентами. Існує кілька поширених причин, через які наші користувачі можуть захотіти виконувати серверний рендеринг. Найпоширенішою причиною є пошукова оптимізація. Тепер, незважаючи на те, що Google підтримує синтаксичний аналіз JavaScript, є інші пошукові системи, які слід розглянути, а пошукові

системи не дуже прозорі щодо точних наслідків обслуговування програми, відтвореної на стороні клієнта, тому багато людей вважають за краще виконувати рендеринг на сервері, щоб бути в безпеці. Пам’ятаємо, що React більше не призначений лише для веб-програм. Зараз люди також створюють статичні веб-сайти за допомогою React. Якщо ми знаємо, що хтось захоче створювати статичні сайти з використанням наших компонентів, тоді рендеринг на сервері необхідний. Популярні інструменти для цього включають Gatsby і Phenomic. Початкова візуалізація на сервері також може підвищити продуктивність за рахунок скорочення часу до першого малювання. При відтворенні на сервері користувачам не потрібно чекати, поки JavaScript буде розібрано, перш ніж побачити щось на екрані, і, нарешті, якщо ми збираємося публікувати компоненти публічно, підтримка серверного рендерингу, очевидно, важлива, оскільки ми не знаємо, як люди будуть використовувати наші компоненти. Публіка буде цього очікувати. На щастя, підтримка відтворення сервера досить проста. Найбільше правило полягає в тому, що нам потрібно перестати припускати, що наші компоненти React запускаються у браузері, тому нам не потрібно виконувати будь-які виклики документів або вікон, а також не використовувати setTimeout. По суті, для підтримки відтворення сервера не можна вважати, що компонент запущено в API браузера.

2.4.8. Принцип єдиної відповідальності

Принцип восьмий: потрібно дотримуватися принципу єдиної відповідальності. Кожен компонент повинен нести чітку єдину відповідальність, але легко розробити компонент, який стає настільки конфігурованим, що його важко зрозуміти, використовувати та підтримувати. Реквізити дозволяють надати стільки точок конфігурації, скільки забажаєте, і якщо ми не проведемо лінію, ми можемо швидко створити компонент, який намагатиметься робити надто багато для багатьох

людей. Надмірно налаштовані компоненти можуть призвести до складних сценаріїв налагодження та можуть ускладнити подальшу впевнену підтримку змін. Виправлення помилки в одному варіанті використання може порушити два інших варіанти використання. Простіше кажучи, зі збільшенням кількості реквізитів зростає і складність. Потрібно слідкувати за тим, скільки реквізитів приймають наші компоненти. Чим більше реквізитів вони приймають, тим складніше буде розуміти і підтримувати компонент. З цієї причини потрібно віддавати перевагу окремим і простим компонентам перед одним складним і дуже налаштовними. Простіші компоненти легше зрозуміти, легше обслуговувати, і менше шансів викликати помилки. Ми завжди можемо складати компоненти разом для досягнення потужних сценаріїв повторного використання. Один компонент не обов'язково повинен бути героєм.

2.4.9. Атомний дизайн

Атомний дизайн - це мова для опису компонентів, написаних на кількох рівнях абстракції. Атоми є найменшими, найпростішими компонентами. Вони є основними будівельними блоками для більших компонентів. Молекули об'єднують два або більше атомів у більшу одиницю, а організми — це групи молекул, що функціонують разом. Ідея багаторазових компонентів, безумовно, не нова. Сучасні автомобілі складаються буквально з сотень багаторазових компонентів, які використовуються в багатьох моделях протягом багатьох років. Розглянемо купу автомобільних деталей. Пружина - це атом. Це найменша одиниця. Ми не можемо зруйнувати пружину далі, але пружина в поєднанні з циліндром і кріпленням створює повний амортизатор. У термінології атомного проектування ми можемо назвати цю збірку молекулою. Це компонент вищого рівня. З'єднавши амортизатор з продольним важелем і маточиною, ми отримаємо базовий рульовий механізм. У термінах атомного дизайну цю абстракцію вищого рівня можна назвати організмом. Все частіше виробники транспортних засобів збирають компоненти вищого

рівня. Це скорочує їхні витрати, зменшує ризик створення дизайну спиць і швидко прискорює доставку нових моделей. Якщо врахувати весь досвід, необхідний для створення навіть найдрібніших частин транспортних засобів, простих компонентів, таких як гвинтові пружини, гайки і ротори, стає зрозуміло, чому повторне використання на рівні компонентів є потужною концепцією, і те саме стосується програмного забезпечення. Компоненти дозволяють нам стати на плечі гігантів. Не так давно ми звикли вважати всю сторінку проблемою. При розробці на стороні сервера кожна сторінка вимагатиме повного завантаження, коли ми переходимо від сторінки до сторінки. Це змусило нас прийняти орієнтовані на сторінку шаблони, такі як модель, перегляд, контролер. З популярним шаблоном MVC ми оголошуємо три окремі файли. Модель оголошує дані для View, View оголошує розмітку для нашого View і HTML, у випадку веб-сторінки, а контролер міститиме логіку, яка пов'язує все це разом і реагує на події. Оскільки MVC — це шаблон, орієнтований на сторінку, коли ми вирішили створити сторінку продукту, ми зазвичай просто створюємо нову модель View and Controller для цієї нової сторінки, тому, хоча MVC є корисним шаблоном, він дуже відрізняється від компонента. орієнтований спосіб мислення, який пропонує React. За допомогою React ми навчилися думати про сторінку як про набір вкладених компонентів. Тож у розробці компонентів ми можемо почати створювати маленькі атоми, а потім ми можемо вкладати ці прості атоми в компоненти вищого рівня, які називаються молекулами та організмами, але малі сфокусовані атоми утворюють основу для все більш вражаючих проектів. Атомний дизайн — це підхід до дизайну компонентів, який популяризував Бред Фрост у його книзі «Атомний дизайн». Таким чином, атомний дизайн заохочує нас думати про свій інтерфейс як про ієрархію компонентів. Атоми – це будівельні блоки, молекули – це група атомів, а організми – це група молекул. Коли ми рухаємося далі, очевидно, складність зростає, а повторне використання зменшується. Легше зробити багаторазовий атом, ніж організм, який можна використовувати повторно. Чому? Тому що

площа поверхні тіла набагато більша. Ми можемо поглянути на це і подумати, що це просто модульний дизайн, і так, це також популярний спосіб описати цей шаблон. Атомний дизайн просто використовує хімічні терміни, щоб уникнути винаходу нового жаргону для рівнів і зробити ієрархію очевидною, але термін модульний дизайн популярний в інших галузях і, як правило, передає ту саму філософію.

2.4.10. Переваги атомного дизайну

Чому атомний дизайн корисний? Атомний дизайн корисний, оскільки ми розглядаємо проект на кількох рівнях деталізації. Ми можемо копнути глибоко і зосередитися на одному атомі, або ми можемо відступити і подивитися на нього на високому рівні, щоб побачити, як усі атоми, молекули та організми об'єднуються, щоб утворити корисне ціле. Це як почистити цибулю. Атомний дизайн заохочує нас створити бібліотеку компонентів, яка містить кілька шарів. Таке мислення заохочує повторне використання, оскільки ми розглядаємо свої компоненти на нижчому рівні і як будівельні блоки для компонентів вищого рівня. Термінологія корисна для передачі зв'язків між компонентами, а також вона корисна для передачі намірів, як і інші мови шаблонів програмування. Якщо ми прагнемо побудувати атом, то зрозуміло, що ми просимо простий базовий компонент, який буде складатися різними способами. Нам потрібно уважніше розглядати широту варіантів використання, ніж якщо ми створюємо компонент вищого рівня, який, як правило, більш пристосований до невеликого набору варіантів використання. Атомний дизайн — це мова, яка передає ієрархію повторного використання. Наші компоненти низького рівня утворюють основу для все більш високих рівнів абстракції.

2.5. Атоми

2.5.1. Атом

Атоми є найменшим будівельним блоком нашої програми, тому вони утворюють основу нашої бібліотеки компонентів, і, як справжні атоми, зазвичай їх не можна побачити поодинці. Атоми корисні, коли вони складаються разом, щоб створити більші, багаті та важливі компоненти. Це основний будівельний блок для створення більших компонентів, і оскільки вони є фундаментальними атомами, часто демонструють та інкапсулюють основні стилі нашого додатка. Звичайно, специфікація HTML включає багато атомів, але, ми можемо обгорнути елементи HTML і покращити їх поведінку.

2.5.2. Обернення примітивів HTML

Чи є сенс обертати примітиви HTML як наші атоми? У примітивах HTML не вистачає ряду вбудованих опцій, які, ймовірно, будуть у реальному додатку. Ми можемо інтегрувати мітку, яка належним чином застосовує необхідні класи та встановлює атрибут for для елемента, який відповідає елементу вхідного ідентифікатора. Ми можемо створити глобально унікальний ідентифікатор для введення, створивши централізовану функцію, яка просто збільшує число при виклику, а потім зіставляти цю мітку та введення разом за допомогою атрибута htmlFor. Ми можемо застосувати послідовний візуальний індикатор для обов’язкового поля, наприклад червону зірку праворуч від мітки. Ми можемо встановити значення заповнювача за умовою, використовуючи значення мітки, а також ми можемо послідовно встановити значення за замовчуванням у спадних меню. Ми можемо застосувати горизонтальний або вертикальний макет для кожного поля або зробити вибір таким же простим, як встановлення опори. Ми можемо інтегрувати повідомлення про помилки нижче або поруч із компонентом, залежно від

вибраного макета. Ми можемо вбудувати метадані про те, чи торкалися його чи забруднені, щоб батьківські компоненти могли перевірити наявність брудних дочірніх компонентів. Ми навіть можемо включити поліфіли для старіших браузерів, які не підтримують власні функції, такі як типи введення чисел. Усі ці елементи виконуються вручну, надлишкові та схильні до помилок без компонентів, які можна використовувати повторно, тому варто приділити час, щоб створити ці абстракції для кодифікації стандартів. Кожна думка, яку ми інкапсулюємо в компонент — це на одне рішення менше, яке має прийняти наша команда, і ще одна функція, якою ми можемо керувати в одному місці. Основні елементи HTML навіть не мають належних імен, тому ми можемо розглянути можливість абстрагування їх більш корисними іменами. Ми можемо обернути введення типом тексту і назвати його TextInput. Ми можемо зробити те ж саме з прапорцем і назвати його Checkbox. Ми можемо обернути Select і назвати його тим, що він є, випадним списком. Ми навіть можемо обгорнути набір перемикачів у список перемикачів, або ми можемо піти ще далі і створити один компонент, який обробляє списки перемикачів і випадні списки під назвою PickOneFromList.

2.5.3. Структура папки

Як ми повинні організувати наші компоненти? Розглянемо структури папок для п’яти найпопулярніших бібліотек компонентів React. React Bootstrap має найпростішу настройку. Усі компоненти розміщені разом в одній папці, а інші проблеми вирішуються в іншому місці. Blueprint розміщує стилі, написані на SASS, поряд з компонентом. Material-UI використовує вбудований стиль, тому немає окремого файлу стилю, але Material-UI поміщає тести компонентів у ту саму папку, що й компонент. Material-UI також надає index.js, щоб імпорт компонентів був коротким. React Toolbox також оголошує index.ts, який забезпечує розширену підтримку автозаповнення для редакторів. Це зручна умова для розгляду, і нам не потрібно писати свій компонент на TypeScript, щоб

користувачі могли користуватися цією перевагою. React Toolbox також розміщує документи поряд з цим компонентом, а також тему за замовчуванням, яку можна замінити. Нарешті, Ant Design — це єдина бібліотека в цьому списку, яка використовує TypeScript замість JavaScript плану. Команда Ant Design розміщує тести та демонстрації у підпапці для кожного компонента. Вони також зберігають документи та формат markdown у папці markdown. У таблиці легше побачити спільні риси та відмінності. Чотири з п'яти найпопулярніших бібліотек компонентів React розміщують кожен компонент у власній папці. React Bootstrap є диваком, оскільки він розміщує всі компоненти в одній папці. Ось чому ми взагалі не бачите X у стовпці React Bootstrap. Усі інші проекти створюють окрему папку для кожного компонента, тому ми будемо використовувати цю умову в нашій демонстрації. Більшість проектів також розміщують стилі компонентів у папці компонентів, але знову ж таки, варто зазначити, що Material-UI — це єдиний проект, який покладається на вбудовані стилі React, і вони вирішують не витягувати ці файли в окремий файл. Ant і Material-UI розміщують тести поряд з компонентом. Тільки React Toolbox оголошує окрему тему за замовчуванням у спеціальному файлі. Усі ці бібліотеки містять робочі демонстрації у своїх документах, але лише Ant Design вирішує зберігати ці демонстрації в самій папці компонента. Нарешті, багато проектів надають index.js у папці компонента, щоб імпортування було коротким та забезпечило простий спосіб імпортувати компонент разом із відповідними активами, такими як стилі.

2.6. Молекули

2.6.1 Молекула

У хімії молекули — це групи атомів, які з’єднані разом і набувають нових властивостей. У термінах атомного дизайну молекули — це кілька атомних компонентів, складених разом, щоб служити вищій меті. Вони все ще відносно

прості, але молекули складають атоми в один компонент. Склеювання атомів утворює простий, функціональний і багаторазовий компонент. Згадавши про принцип єдиної відповідальності, він також стосується молекул. Різниця полягає в тому, що, оскільки молекула є абстракцією вищого рівня, єдина відповідальність є відповідальністю вищого рівня, але добре сконструйована молекула все одно повинна прагнути робити щось добре. Незважаючи на те, що молекула складається з кількох атомів, вони повинні складатися разом стратегічно, щоб нести чітку, чітку відповідальність. Це допомагає гарантувати, що молекулу легко зрозуміти, спожити та створити з іншими молекулами організми вищого рівня. Як зазначає Бред Фрост у своїй книзі про атомний дизайн, молекули води та молекули перекису водню мають свої унікальні властивості і поводяться зовсім по-різному, навіть якщо вони складаються з однакових атомних елементів. Справа в тому, що дві молекули можуть виглядати і вести себе зовсім по-різному, навіть якщо вони складаються з багатьох однакових компонентів, але при об'єднанні атомів вони виконують певну мету.

2.7. Організми

2.7.1 Організм

Так само, як молекули складаються з атомів, організми складаються з молекул. Організм складніший, ніж молекула, оскільки він об'єднує молекули та інші атоми в абстракцію вищого рівня. Зазвичай це призводить до створення компонента інтерфейсу користувача, який обробляє частину екрана. Іноді організм або молекула можуть реалізувати фасадний візерунок. Шаблон фасаду абстрагує більш складний API, надаючи більш простий, більш детальний API, тому, наприклад, атом може прийняти дюжину опорних елементів для підтримки широкого спектру випадків використання.

2.7.2. Німі організми

Намагаймося, щоб наш організм був німим. Не потрібно перетворювати організми на міні-додатки. Потрібно створювати німі організми. Так само, як і наші атоми та молекули, організми зазвичай мають бути простими функціями, які беруть реквізит і виводять HTML. Потрібно уникати створення автономних міні-програм. Однак може бути корисно створювати компоненти контейнера, щоб заощадити час людей. Компоненти контейнера обробляють стан і логіку і зазвичай містять дуже мало розмітки. Вони делегують обробку розмітки німим дочірнім компонентам. Це допомагає розділити проблеми і гарантує, що наші тупі компоненти можна легко перевірити окремо, оскільки вони є простими, чистими компонентами, які приймають введення та повертають HTML. Наприклад, ми створили YouTube, використовуючи філософію атомного дизайну. Заголовок — це організм, а розділ коментарів — його власний організм. Звернемо увагу на те, як мій знімок відображається як у заголовку, так і в розділі коментарів. Кожен із цих організмів має бути німим, тож, наприклад, у них не повинно бути запечених викликів API. Натомість їх потрібно обробляти як і будь-яка інша функція багаторазового використання. Їм слід передати необхідні дані. Це дозволяє уникнути зайвих викликів API для отримання тих самих даних. Якщо кожен організм піклується лише про себе, тоді буде два виклики API, щоб отримати URL-адресу з фотографії. Не добре. Є й інші питання. Якщо ми змінимо свою фотографію, то обидва ці організми повинні відобразити зміну, але якщо кожен компонент піклується тільки про себе, як ми можемо гарантувати, що обидва організми оновлюються, коли змінили своє зображення? І якщо кожен організм володіє знаннями про те, як здійснювати виклики API, ми також ризикуємо надіслати дублікати логіки в двох різних файлах JavaScript, що витрачає пропускну здатність, сповільнює завантаження сторінок і збільшує навантаження на пам’ять. Під час створення компонентів для багаторазового використання може виникнути спокуса створити розумні багаторазові компоненти, які

знають, як обробляти аутентифікацію, виклики API, авторизацію тощо. Деякі люди називають шаблон міні-додатками. Натомість ми будемо тримати всі наші компоненти багаторазового використання, включно з організмами. Ось чому. Якщо ми створили кілька багатих організмів, які достатньо розумні, щоб повністю обробляти дві окремі частини однієї сторінки, що станеться, коли однакові дані відображаються в двох місцях? Якщо кожен компонент дбає лише про себе, то як ці два залишаються синхронізованими? Це робота додатків, тому не дозволяємо надмірно впевненому та складному організму скомпрометувати архітектуру наших програм. Розумні організми часто призводять до надмірного вилучення. Коли кожен компонент турбується лише про себе, ми, швидше за все, запитуватимемо одні й ті самі дані в кількох місцях. Це фундаментальна проблема створення розумних компонентів у стилі організму. Додаток має бути розроблений цілісно, але розумний настрій організму призводить до створення ряду егоцентричних компонентів разом, що призводить до низької продуктивності та марнотратного дизайну. Розумні організми можуть вбудовувати API, аутентифікацію та інші перехресні проблеми. Це може призвести до того, що наші користувачі завантажуватимуть той самий код у кількох пакетах JavaScript, що уповільнить завантаження сторінок і втратить пропускну здатність. За іронією долі, розумні організми часто призводять до тісного зв’язку, оскільки нам потрібні організми, щоб працювати як єдине ціле в реальному додатку, нам, можливо, доведеться тісно інтегрувати два окремих компонента, щоб змусити їх діяти як єдине ціле. Ця інтеграція є крихкою, і вона ускладнює подальшу зміну організму. Тож потрібно думати про атоми, молекули та організми так само. Як правило, це тупі, багаторазові компоненти, які приймають деякі параметри і повертають HTML. Зробимо їх простими. Думати про них потрібно як про багаторазовий інтерфейс і уникати перетворення організмів у мініатюрні програми. Організми по суті те саме, що й атом. Різниця полягає в тому, що організми повертають більші фрагменти HTML, складаючи багато дочірніх компонентів разом впевненим і корисним способом.

2.8. Стиль і тематика

2.8.1. Скомпільований CSS

Багато з найпопулярніших бібліотек компонентів React сьогодні вибирають компілювати свої CSS. Sass та Less є популярними засобами покращення CSS, і все частіше люди використовують PostCSS разом із cssnext, щоб насолоджуватися найновішими функціями CSS, перш ніж вони фактично підтримуються між браузерами. Всі ці інструменти мають спільну нитку. Вони компілюються до звичайного CSS. Вони популярні, оскільки пропонують вбудовану підтримку змінних, користувацькі функції, автоматичне встановлення префіксів постачальників, тому нам не потрібно самостійно відстежувати підтримку браузера, а Sass і Less також дозволяють оголошувати стилі за допомогою вкладених селекторів. Зауважимо, що PostCSS у поєднанні з cssnext є більш дружнім підходом, оскільки cssnext дозволяє використовувати завтрашній синтаксис CSS сьогодні. Це означає, що теоретично, якщо браузери реалізують усі передові функції, які підтримує cssnext, нам більше не потрібно буде компілювати свій CSS. Тепер ось приклад змінних Sass. Вони декларуються з доларом наперед. З бібліотеками компонентів для багаторазового використання зазвичай створюється файл, наповнений цими змінними, щоб ми могли уникнути повторення під час додавання стилів. Це також дає змогу створювати теми для компонентів, використовуючи інший файл змінної, і ось приклад вкладеності Sass. Він чутливий до пробілу, тому Saas зазначає, що ми вставили ul та li під nav. Результуючий CSS, який він створює, виглядає як висновок справа, тому Sass уникає необхідності повторювати батьківський селектор і робить вкладення дуже очевидним візуально. Тепер Sass, безумовно, має набагато більше і Less, але це пара основних функцій, які роблять їх привабливими. І коли ми говоримо про скомпільований CSS, ми також повинні обговорити PostCSS. PostCSS схожий

на Babel, але для CSS, тому що він транспілює наш CSS. Ми можемо використовувати PostCSS для автоматичного встановлення префіксу стилю, щоб нам не довелося вручну встановлювати префікси постачальника для підтримуваних браузерів. Ми можемо використовувати функції CSS4 сьогодні, а PostCSS може транспілювати їх. Ми можемо поліфілити нові функції, як Flexbox, які не підтримуються в старих браузерах. Ми можемо лінтувати свій CSS, щоб забезпечити дотримання умов і знайти поширені проблеми, а також ми можемо автоматично застосувати доступність.

2.8.2. Схеми найменування

Незалежно від того, яке компільоване рішення CSS ми вибираємо, у нас є проблема. Зрештою, він компілюється до звичайного CSS, а це означає, що велика проблема із простим CSS залишається, і це полягає в тому, що ніхто не буде відчувати себе комфортно, видаляючи будь-які стилі з таблиці стилів, оскільки відстеження залежностей відбувається вручну, займає багато часу та схильне до помилок. Люди турбуються про випадкове написання стилю, який стикається з іншою бібліотекою або програмою, тому на практиці люди часто пишуть довгі та крихкі селектори, і вони майже ніколи не видаляють файли, навіть якщо вони твердо вірять, що вони не використовуються. В результаті ми отримуємо файл CSS, який переповнений купою не послідовних і крихких селекторів, а також різноманітними стилями, які не потрібні, оскільки їх занадто ризиковано змінити або видалити. Щоб допомогти вирішити цю проблему, з’явилися різні схеми імен, включаючи BEM, OOCSS і SMACSS. Усі три схеми іменування мають однакову місію – пом’якшити глобальну та неявну природу CSS за допомогою послідовного стандарту імен. Згідно з опитуванням 2017 року на сайті sitepoint, 46% розробників використовували схему імен CSS. Найпопулярнішим був BEM: 40% розробників сказали, що ним користувалися. Оскільки BEM є найпопулярнішою методологією з цих трьох, швидко обговоримо її більш детально. BEM — це проста схема іменування. Ми

використовуєте класи для оголошення всіх стилів. BEM розбиває класи компонентів на три групи, які починаються з літер B, E та M; Блок, елемент і модифікатор. Тому він і називається БЕМ. Блок є коренем компонента, тому в React кожен компонент, як правило, є блоком, тож, наприклад, цей електронний лист із вбудованою міткою та повідомленням про помилку є цілісним компонентом. Це був би блок. Елемент є частиною блоку, тому в React елемент є частиною компонента, тому це може бути мітка, яка є частиною цього більшого компонента. Модифікатор - це варіант або розширення блоку, наприклад, стан помилки. Вбудовані стилі React застосовуються Inline, але їх не потрібно оголошувати Inline. Оскільки це просто JavaScript, ми можемо перемістити стилі в окремий файл, якщо хочемо. Розглядаючи вбудований стиль, нам потрібно задати одне велике запитання: наскільки ми хочемо, щоб мій стиль був впевненим? Якщо ми створюємо багаторазові компоненти React для внутрішнього використання, ми можемо переконатися, що їхні стилі застосовуються точно без їх легкого пере визначення. Вбудовані стилі забезпечують сильне виконання наших рішень щодо стилю. Навпаки, зовнішні стилі забезпечують більшу гнучкість. Зовнішнім CSS легко маніпулювати, перевизначати або повністю замінити, що може призвести до невідповідностей, але якщо ми розробляємо багаторазові компоненти для широкого загалу, то зовнішні стилі дають людям більше можливостей налаштувати наш компонент відповідно до своїх потреб. І вбудований, і зовнішній підходи до стилю продовжують мати переваги, тому подумайте про використання обох. Вибір використання одного з варіантів виключно обмежує нас від насолоди їх перевагами в належному контексті. Звичайно, мінусом змішування є те, що нам потрібно запустити програму, щоб побачити, як відтворюються остаточні стилі. Ми втрачаємо деякі детерміновані переваги вбудованих стилів, змішавши їх із традиційними зовнішніми стилями.

2.8.3. Модулі CSS

Модулі CSS – це цікавий підхід, який автоматично інкапсулює наші стилі. З модулями CSS імена наших класів за замовчуванням визначаються локально. Ось як це зазвичай працює. Спочатку ми пишемо свій CSS у традиційній окремій таблиці стилів. Ми також можемо використовувати Sass або Less, якщо хочемо. Потім ми використовуємо завантажувач CSS Webpack для імпорту таблиці стилів. Webpack підтримує імпорт стилів так само, як ми імпортуємо JavaScript, і коли ми імпортуємо стилі через Webpack, ми посилаємося на конкретні частини таблиці стилів, як на об’єкт. Наприклад, уявимо, що ми оголошуємо просту таблицю стилів, яка має клас під назвою error, який забарвлює текст у червоний колір. Потім нам потрібно налаштувати завантажувач CSS Webpack, щоб увімкнути модулі. Конфігурація модулів CSS для Webpack 1 і Webpack 2 відрізняється. Webpack 1 виглядає страхітливо, оскільки це одна довга лінія. Webpack 2 розбиває конфігурацію на кілька окремих, добре позначених властивостей під об’єктом параметрів, і так, обидва вони виглядають заплутаними, але в основному ми кажемо Webpack використовувати ім’я компонента та додати хеш до end, щоб безпечно інкапсулювати стиль, і використовуємо css-loader. Ми можемо імпортувати просту таблицю стилів, яку може обробляти Webpack. Знову ж таки, коли наші завантажувачі налаштовані, ми можемо імпортувати стилі так само, як імпортуємо JavaScript, і Webpack об’єднає це для нас, і посилаємося на класи CSS у своїй таблиці стилів як на об’єкти тут нижче. Тут ми говоримо про стилі .error, що означає, що ми говоримо, застосувати клас помилок, який ми визначили у таблиці стилів. Тепер ось що відбувається під час складання. Webpack замінює цей виклик унікальним ім'ям класу, об'єднуючи ім'я компонента з className і випадковим набором символів у кінці. Це запевняє, що стиль інкапсульований, хоча ми написали справді кульгавий і загальний селектор під назвою помилка, тому нам не доведеться турбуватися про те, що він був дуже, дуже загальним. Для нас він інкапсулюється, тому

модулі CSS автоматично виконували роботу BEM. За допомогою модулів CSS ми можемо писати простий CSS, Sass або Less. Це перемога, оскільки це означає, що ми можемо використовувати інструменти, пов'язані з CSS, і це доступне для дизайнерів. Стилі, які ми використовуємо, чітко посилаються у наших компонентах, тому зрозуміло, звідки беруться стилі. Ми можемо створювати стилі разом за бажанням, щоб створити добре названі стилі високого рівня, і, як вбудовані стилі в CSS і JavaScript, модулі CSS автоматично інкапсулюють наші стилі для певного компонента, тому нам не доведеться турбуватися про витік стилів. виходить із сторонніх таблиць стилів або зіткнутися з ними. Згенеровані назви класів надзвичайно унікальні завдяки хешам, розміщеним у кінці, тому нам не доведеться турбуватися про конфлікти імен, і нам більше не доведеться докучати людям, щоб вони вручну виконували правила, такі як BEM, або відповідали простору імен вашим класам, щоб уникнути конфліктів. Тепер він запроваджено програмно, тож ми можемо писати короткі, читабельні селектори CSS без турботи, і ми все ще можемо використовувати глобальний CSS, який продовжує бути корисним для каскадування, класів тіла, шрифтів тощо. І ще одна очевидна перевага модулів CSS полягає в тому, що, на відміну від більшості рішень CSS і JavaScript, нам не потрібно посилати з ним середовище виконання, оскільки модулі CSS компілюються до звичайного CSS під час створення. Звичайно, є деякі мінуси. По-перше, ми не можемо створювати динамічні стилі на основі стану, як ми робили раніше, використовуючи вбудовані стилі React, а також тематизація може бути досить складною з модулями CSS.

2.8.4. CSS в JS

Останній підхід до стилю CSS, який слід розглянути, - це CSS в JavaScript. За моїми останніми підрахунками, існує щонайменше 47 різних проектів із відкритим кодом для обробки стилів React через JavaScript. Враховуючи тепер, цікаво, що багато розробників відчувають огиду

до ідеї використання JavaScript для генерування стилів, але протягом багатьох років переважна більшість розробників прийняла ідею використання Sass або Less для додавання потужності CSS, але чомусь Ідея використання JavaScript для стилів відлякує багатьох людей. Якщо ми відступимо і подумаєте, ця реакція, можливо, нелогічна. Чому? Тому що Sass і Less— це просто вигадані мови, які компілюються в CSS. Тепер, як ми зрозуміли, багато хто вважає ідею використання JavaScript для створення CSS непривабливою, тому що здається, що ми використовуємо неправильний інструмент для цієї роботи. Sass і Less дуже схожі на CSS, що багато хто вважає за особливість, але якщо ми вважаємо, що Sass і Less насправді є просто створеними мовами, які компілюються до CSS, ми повинні визнати, що принаймні варто розглянути можливість використання Натомість JavaScript генерує CSS, і є кілька важливих причин використовувати JS для створення стилів замість використання Sass, Less або інших специфічних CSS альтернатив. Перша і найбільш очевидна перевага полягає в тому, що ми вже знаємо JS. Чому кожен у нашій команді повинен навчитися новому способу обробки змінних, функцій і циклів? JavaScript вже справляється з цим елегантним способом. JavaScript є кращою, більш зрілою мовою. Він більш ніж здатний вирішити цю проблему. Ми можемо насолоджуватися композиційною силою деструктуризації, розгортанням об’єктів, функціями стрілок тощо для створення своїх стилів, а коли ми використовуємо JavaScript для своїх стилів, ми можемо використовувати наявні інструменти JavaScript замість того, щоб створювати окремий ланцюжок інструментів для CSS. Це означає, що ми можемо використовувати ESLint, щоб перевірити свій код стилю, і ми можемо зменшити, використовуючи той самий мініфікатор, який ми використовуємо для коду програми. Нарешті, стилі, які ми не використовуємо, потенційно можуть бути усунені вашим пакетом; однак, безперечно, є деякі проблеми. За допомогою традиційного CSS ми можемо створити вихідну карту CSS, щоб допомогти налагодити наші стилі у виробництві, але ми упускаємо підтримку вихідної карти, коли пишемо стилі за допомогою JavaScript. Ми також

заблоковані у вибраній бібліотеці. Стилі, які ми пишемо, є специфічними для обраної нами бібліотеки JavaScript і CSS, тому їх не можна переносити в інші проекти, які використовують традиційний стандартизований CSS. Бібліотеки CSS та JS часто не мають повної підтримки всіх функцій CSS, таких як медіа-запити та псевдоелементи, і люди зазвичай пом'якшують це, звертаючись до традиційного CSS у цих випадках. Це означає, що ми в кінцевому підсумку змішуєте CSS і JavaScript із звичайним CSS, і виникає питання: де нам шукати стилі для даного компонента? Це часто стає сумішшю двох технологій, які визначаються в двох різних місцях. Нарешті, коли ми вибираємо CSS в JavaScript, споживачі нашого багаторазового компонента повинні прийняти ваше рішення. Вони можуть віддавати перевагу простому CSS, але замість цього вони змушені працювати з CSS у JavaScript. Тепер, якщо ми вирішили використовувати стилі на основі JavaScript в React, нам доведеться прийняти ще важче рішення, який JavaScript у бібліотеці CSS нам слід використовувати? Кілька популярних варіантів включають Aphrodite, Radium, JSS, jsxstyle, Styled-components, Glamour та ще понад 40. Кількість варіантів просто безглузда. Велика перевага стилізованих компонентів полягає в тому, що ми можемо писати простий CSS, на відміну від більшості інших CSS у Рішення JS, які ми бачимо тут, які вимагають від нас написання стилів та об'єктних літералів.

2.9. Тестування

2.9.1. Фреймоврк

Існує принаймні шість важливих рішень, які нам потрібно прийняти, коли ми налаштовуємо автоматичне тестування в JavaScript. Ми повинні вибрати фреймворк, бібліотеку тверджень, допоміжні бібліотеки, вирішити, в якому середовищі запускати тести, куди розмістити файли і, нарешті, коли запускати тести. Перше рішення, яке нам потрібно прийняти, — це те, яку систему

тестування використовувати. Є широкий вибір на вибір, включаючи мокко, жасмин, стрічку, QUnit, AVA та Jest. Ми будемо використовувати Jest, але підходить будь-яка з них.

Ми будемо використовувати Jest для проведення модульного тестування. Тож коли слід використовувати модульне тестування? Коли ми перевіряємо бізнес-логіку, такі речі, як перевірка введених даних, перетворення даних, сортування, фільтрація тощо. Модульне тестування фокусується на окремих модулях або функціях і гарантує, що вони працюють належним чином. Другим типом тестування інтерфейсу користувача є тестування взаємодії. Це стосується перевірки того, що взаємодії з користувачем обробляються належним чином. Приклади тестування взаємодії включають твердження, що функція викликається, коли натискається певний елемент, або що повідомлення відображається, коли користувач надсилає форму. З React нам не потрібно відкривати справжній браузер. Замість цього ми можемо використовувати такі інструменти, як Enzyme, щоб написати ці тести, тож коли тести взаємодії корисні? Ну, очевидно, коли у нас є компонент, який є інтерактивним. Якщо наш компонент реагує на внесок клієнта, тести взаємодії можуть допомогти переконатися, що він відповідає очікуванням. Ми можемо перевірити, чи наш компонент реагує на події, або, можливо, наш компонент генерує випадковий вихід для заданого входу. Це нечасто, але коли це відбувається, тоді тест взаємодії може запевнити, що дані відображаються в бажаному положенні, коли відбувається взаємодія, або що функція викликається, як очікувалося. Третій вид тестування — структурне тестування. Структурне тестування запевняє, що вихідний HTML-код даного компонента React відповідає очікуванням, тому структурне тестування схоже на фотографування виводу компонента. Jest пропонує гладкий і автоматизований спосіб створення структурних тестів, який називається тестуванням моментальних знімків. Тестування моментальних знімків Jest зберігає копію того, що виводить даний компонент із заданим введенням. Таким чином, ми відразу отримуємо сповіщення, коли вихідні дані

компонента змінюються. Більше ніяких сюрпризів. Тестування моментальних знімків – це часто все, що нам потрібно для функціональних компонентів без стану. Ми встановлюєте деякі параметри, і знімок відстежує, як повинен виглядати компонент із поданими даними. Легко. Знімки зберігають текстове зображення виводу нашого компонента для заданого введення. Це робить знімки дійсно корисними для не інтерактивних компонентів або для тестування детермінованого виходу будь-якого з наших компонентів. Наприклад, ми можемо підтвердити, що дані приховані, якщо певний параметр false або що деяка розмітка відображається лише тоді, коли дані доступні. ми навіть можемо поєднувати тестування моментальних знімків із тестуванням взаємодії. Наприклад, ми можемо використовувати Enzyme, щоб створити клацання кнопки, а потім створити тест знімка, який фіксує очікуване відображення інтерфейсу користувача після цієї взаємодії. Як ми побачимо, тести моментальних знімків настільки прості у використанні, що ми, ймовірно, будемо використовувати їх у багатьох місцях. Так чому ж тестування знімків? Тестування моментальних знімків легко налаштувати. Для створення нових тестів потрібно всього кілька рядків коду, і натомість ми отримуємо сповіщення під час відтворення змін. Коли відображення змінюється, ми переглядаємо зміни, і якщо це зміна, яку ми збиралися зробити, натисніть U, щоб автоматично оновити відповідні знімки, і це швидко та легко налаштувати, оскільки нам не потрібно взаємодіяти з браузером. Як і інші тести, які ми збираємося написати, тести виконуються в пам'яті. Нарешті, ми можемо налагоджувати тести знімків, як і код. Вихідні дані можна читати людиною, оскільки це просто результат візуалізації компонентів, однак знімки не призначені для тестової розробки. У традиційному TDD ми дотримуємося червоного зеленого циклу рефакторингу. Ми пишемо невдалий тест, пишемо код для проходження тесту, а потім реорганізуємо. Тестування моментальних знімків не призначене для цього потоку. Тестування знімків — це те, що ми додаємо після завершення, тому тестування знімків не для TDD, але воно чудово підходить для тестування після розробки. Подумаємо про тестування

знімків як ще один корисний спосіб перевірити свій інтерфейс, але це просто ще один інструмент. Нарешті, тестування стилів підтверджує, що стилі застосовуються належним чином. За допомогою традиційного тестування стилю ми порівнюємо буквальні знімки екрана поточного виводу в порівнянні з історичним результатом, тому тестування стилю зазвичай порівнює зображення, а тестування знімків Jest порівнює вихідний код. Якщо ми використовуємо вбудовані стилі, то структурного тестування за допомогою знімків дійсно достатньо, оскільки стилі є детермінованими, але якщо ми використовуємо традиційні зовнішні каскадні таблиці стилів, нам потрібно буде відкрити фактичний браузер, щоб перевірити, чи застосовуються стилі. як і очікувалося, і ось тут у нагоді стає тестування стилю.

2.9.2. Допоміжні бібліотеки

Є ще одне запитання, на яке потрібно відповісти, перш ніж ми почнемо писати тести. Чи варто використовувати якісь допоміжні бібліотеки? Ми збираємося використовувати Enzyme як нашу основну допоміжну бібліотеку для тестування взаємодії. Enzyme — це бібліотека від Airbnb, яка дозволяє легко виконувати тести взаємодії, такі як моделювання кліків і запити DOM. Ми могли б використовувати звичайні тестові утиліти React, але Enzyme пропонує набагато зручніший API. Насправді, це настільки добре, що документація React навіть рекомендує використовувати Enzyme замість їхніх власних тестових утиліт React. Ми також будемо використовувати react-test-renderer, щоб відтворити наші знімки Jest.

2.10. Рішення про розподіл

2.10.1. Відкрите, закрите чи внутрішнє джерело?

Перш ніж публікувати свої компоненти для використання іншими, нам потрібно вирішити, як ми збираємося керувати базою коду. Наша модель має бути закритим, внутрішнім чи відкритим кодом? Якщо ми створюємо компонент, який, на нашу думку, буде корисним для широкої публіки, подумайте про його відкритий код. Розглянемо переваги відкритого джерела перед закритим. Існує ряд причин, щоб розглянути можливість використання відкритих джерел для компонентів React. Найочевидніша перевага полягає в тому, що це сприяє розвитку спільноти React. Чим більше онлайн-ресурсів для розробників React, тим більше людей охопить React, і якщо Наша компанія прагне React, то розвиток спільноти React також допоможе нам. Публікація популярного проекту з відкритим кодом також допомагає з підбором персоналу. Наша компанія може бути зазначена як спонсор проекту. Цей ефект ореолу може допомогти підвищити інтерес розробників до роботи у нашій компанії і навіть працювати як дошка вакансій, коли є вакансії. Відкритий код означає, що громадськість допоможе попередити нас про будь-які помилки. Люди швидко повідомляють про проблеми з відкритим вихідним кодом. Ми можемо опублікувати зміни в загальнодоступному репозиторії і залишити їх на кілька тижнів, перш ніж використовувати зміни у нашому власному програмному забезпеченні, чекаючи, щоб побачити, чи не виникнуть якісь непередбачені проблеми. Це, безсумнівно, краще дізнаватися про помилки від наших клієнтів, і ми можемо насолоджуватися безкоштовною розробкою, оскільки люди надсилають запити на виправлення помилок та покращення нашого компонента, тому відкрите джерело має багато переваг, але є кілька важливих причин, що компанії ухиляються від відкритих джерел будь-якого свого програмного забезпечення. Якщо у нас є проблеми з конфіденційністю, то відкрите кодування не для нас, хоча загалом для компонентів React це навряд

чи буде проблемою. Закрите джерело також дає нам повну свободу радикально змінювати свій дизайн у будь-який час, не впливаючи на широку публіку, і ми також можемо включати специфічні функції компанії; однак, ми часто можемо додати особливі функції компанії, просто загорнувши загальнодоступну версію нашого компонента. Ми щойно обговорювали переваги відкритого та закритого коду, але Walmart використовує свого роду золоту середину, яку вони називають внутрішнім джерелом для розробки компонентів React.

2.10.2. Хостинг пакетів

Де ми повинні розміщувати свої пакунки? Тепер ми можемо бути здивовані, виявивши, що існує безліч способів розміщення наших пакетів, щоб інші люди могли використовувати їх. ми можемо подумати: «Ей, який тут вибір?» Очевидно, ми повинні розмістити свій пакет на npm, чи не так? Так роблять усі. Ну, не зовсім. Хоча npm, безумовно, є найпопулярнішим підходом, насправді npm є найбільшим у світі менеджером пакетів з понад 400 000 пакетів, а також менеджером пакетів, що швидко розвивається; однак ми можемо розмістити свій пакет компонентів для багаторазового використання в іншому місці. Якщо хостинг на загальнодоступному npm не є варіантом, ось кілька альтернатив, які варто розглянути. Sonatype пропонує Nexus Repository Pro, JFrog пропонує JFrog Artifactory, a inedo пропонує ProGet. То чому б ми розміщували свій пакунок npm за допомогою одного з цих універсальних менеджерів пакетів замість npm? Ну, по-перше, у нас повний контроль. нам не потрібно довіряти сторонній стороні для розміщення наших компонентів, і якщо у нас є інші типи пакетів у нашій компанії, ми можемо розмістити їх всі в одному місці. Наприклад, репозиторії, які щойно перерахували, також підтримують розміщення популярних пакетів, таких як NuGet, Maven, Docker, Python, RubyGems тощо. Розміщення всіх наших пакетів в одному місці може спростити встановлення та керування доступом, оскільки у нас є єдине місце для надання та скасування доступу. Якщо у нашому середовищі є багато різних

типів пакетів, може бути зручно керувати ними всіма в одному місці, і якщо ми вирішили зберегти свій пакет конфіденційним, нам доведеться заплатити за розміщення нашого пакета на npm. Це також може стосуватися інших типів пакетів, тому, розміщуючи всі свої пакети всередині, ми можемо уникнути сплати окремих зборів за кожен тип пакета. Звісно, у цього підходу є деякі негативні сторони. Найважливішим є те, що наш пакет стає недоступним для широкого загалу. Тепер, якщо припустити, що ми вирішили зберегти свій пакет конфіденційним, це функція, але якщо ми хочемо оприлюднити широку публіку, тоді, очевидно, нам підходить хостинг на npm, і він також менш доступний для нових працівників. На відміну від загальнодоступного npm, щоб нові співробітники отримали доступ до нашого пакету, ми повинні надати їм доступ, і щоб будь-який з них фактично встановив і використав наш пакет, їм доведеться налаштувати npm на своїй локальній машині, щоб він вказував на наш реєстр замість npm за замовчуванням реєстру, тому розміщення в універсальному менеджері пакунків всередині створює деякі початкові тертя, але найпростіший спосіб зберегти наш пакет npm приватним — це за допомогою npmjs.com. Ми можемо створити організацію, а потім прив’язати всі свої пакети до організації. Таким чином, наші пакунки не будуть доступні для всіх, але будь-хто, кого ми призначите своїй організації, може встановити наші пакунки. Наразі npm стягує невелику плату з кожного розробника за цю послугу, але це дійсно низька тертя. Коли ми розміщуєте приватно на npm, ми оголошуємо назву організації. Після того як ми надаємо людям доступ, вони зможуть встановити наш пакет, вказавши організацію перед назвою пакета. Ще одна приємна перевага приватного хостингу на npm полягає в тому, що наша організація працює як приватний простір імен, тож ми можемо вибрати будь-яке ім’я для свого пакета, навіть якщо існує якийсь загальнодоступний пакет з цим ім’ям, ми можемо створити внутрішній пакет з тим самим ім’ям тому що він фактично розміщений у нашій організації. Ми опублікуємо нашу бібліотеку компонентів публічно в npm.

2.10.3. Імпортні підходи

Перш ніж публікувати наші документи та налаштовувати процес складання, нам потрібно вирішити, як нашим користувачам імпортувати наші компоненти. Існує три популярних підходи до імпорту, які ми можемо розглянути як підтримку. Перший – це імпортований підхід. Цей підхід використовує іменовану функцію імпорту в JavaScript, щоб зробити оператор імпорту красивим і коротким, тому перевага цього підходу полягає в тому, що він найбільш стислий. Його легко читати та вводити, оскільки нам потрібно лише один раз вказати назву компонента; однак у такого підходу є і мінуси. По-перше, і найголовніше, цей стиль фактично імпортує всю бібліотеку. Це поширена помилка. ми можемо подумати, що використання іменованого імпорту тут означає, що лише цей один компонент імпортується у нашу програму, але це не так. Фактично вся бібліотека компонентів імпортується. Ця лінія насправді виконує дві речі. Він імпортує всю бібліотеку компонентів, а по-друге, посилається на компонент Label як Label, тому імпорт таких компонентів має певні наслідки для продуктивності. Він роздуває пакет нашого користувача, імпортуючи весь пакет, тому це може призвести до роздутої програми, якщо наша бібліотека велика і вони використовують лише невелику частину наших компонентів. Іншим недоліком є те, що нам потрібно створювати і підтримувати index.js, який експортує посилання на всі наші компоненти з кореня папки компонентів, тому цей підхід потребує додаткового обслуговування. Другий підхід полягає в тому, щоб записати свої компоненти в папку під назвою lib і оновити документи, щоб безпосередньо посилатися на компоненти. Основна перевага цього підходу полягає в тому, що він імпортує лише певний компонент, який ми вказали. Це допомагає зменшити розмір пачки. нам також не потрібно підтримувати index.js у корені нашого пакунка, оскільки він не використовується цим підходом до імпорту, а фактично, пропускаючи індекс. js, ми можемо захистити людей від випадкового імпорту всієї нашої бібліотеки. Це також найпоширеніший підхід, і збірка проста в

налаштуванні та обслуговуванні. ми можемо використовувати традиційні підходи для включення та виключення файлів, наприклад. npmignore або масив файлів у package.json. Очевидним недоліком є трохи довший оператор імпорту, оскільки користувачі повинні вказати підкаталог lib і шлях імпорту, а також двічі ввести ім'я компонента. Третій підхід — опублікувати наші компоненти в корені пакета, щоб ми могли імпортувати компоненти з кореня нашого пакета npm. Подібно до імпорту з каталогу lib, це імпортує лише певний компонент, і, як і підхід номер два, нам не потрібно створювати та підтримувати index.js у корені пакета, тому тут багато помітних переваг, але у цього підходу є один істотний недолік. Щоб підтримувати цей стиль імпорту, нам потрібно налаштувати спеціальний процес збірки, який публікує наші компоненти в кореневому каталозі нашого пакета. Тепер npm не дуже зручний для цього підходу, тому для цього потрібно виконати кілька кроків. нам потрібно динамічно згенерувати package.json, який копіює відповідні налаштування з нашого поточного package.json. Нам потрібно скопіювати такі ресурси, як readme, до папки, яку ми збираєтеся опублікувати в npm, і оскільки цей підхід використовує користувацький скрипт для копіювання файлів, ми не використовуємо традиційний .npmignore або масив файлів для оголошення файлів, які будуть існувати у нашому пакеті. Натомість ми запускаємо команду publish з папки lib, яку ми створюємо. Тож нам може бути цікаво, як різні популярні бібліотеки компонентів React справляються з цим? Що ж, Ant Design і Blueprint використовують названий стиль імпорту у своїх документах. React Toolbox визначає повний явний шлях до папки lib у своїй документації, а Material-UI використовує короткий стиль прямого імпорту. Вони підтримують цей підхід, використовуючи спеціальну конфігурацію збірки, подібну до тієї, яку ми щойно описали на попередньому слайді. React Bootstrap взагалі не відображає імпорт у своїх прикладах коду. Замість цього вони вирішили централізувати загальну документацію імпорту, але вони документують як підхід імпорту з імпортом, так і підхід імпорту з бібліотеки. Вони також показують, як імпортувати за допомогою імпорту в стилі CommonJS і ES, тому

подумаємо про те, щоб додати примітку до документів. Ось три підходи пліч-о-пліч. Як ми щойно побачили, кожен з цих підходів має переваги. Наші документи будуть використовують підхід третій.

2.10.4. Формат виведення

Наступне рішення – які вихідні формати ми повинні опублікувати? На даний момент існує три варіанти форматів виводу збірки: ES5 з CommonJS, який сьогодні є стандартом для пакетів npm, ES5 з модулями ES або Universal Module Definition. Зараз ми знаходимося в цікавому періоді для JavaScript та Node, хоча Node вже деякий час підтримує ES6, більшість людей продовжують публікувати пакунки виключно в ES5, використовуючи формат CommonJS, тому нам обов'язково слід це зробити. Сьогодні це залишається стандартною практикою для npm. Ми створимо нашу бібліотеку компонентів, використовуючи цей формат.

2.10.5. Збірка модуля ES

Обговоримо варіант створення збірки модуля ES і чому він потенційно привабливий. Модулі ES6 дозволяють модулювати код JavaScript, явно оголошуючи імпорт та експорт. Цю функцію JavaScript зазвичай називають модулями ES6, тому що це була версія JavaScript, де ця функція була представлена, однак ця версія також називається ES2015 після року, в якому вона була представлена, оскільки всі майбутні версії JavaScript будуть називатися за роком, в якому вони були. повторно звільнений. Цю функцію також називають модулями TC-39, оскільки TC-39 — це назва комітету стандартів JavaScript. ми можемо просто назвати це модулем JavaScript. Деякі називають це модулем EcmaScript, оскільки EcmaScript є офіційною назвою для JavaScript, оскільки ECMA була назвою початкового комітету стандартів JavaScript. Хтось скорочує це і називає його модулем ES, а хтось скорочує це

ще більше і просто називає це ESM, що означає модуль EcmaScript. Нарешті, ми можемо просто назвати це модулем, оскільки це єдиний формат модуля, який насправді є частиною стандарту JavaScript. Усі інші модульні підходи — це лише нестандартні шаблони, які люди популяризували. Зі збіркою модуля ES ми все одно транспілюємо свій код до ES5, як зазвичай, щоб гарантувати, що він може працювати скрізь, але є один елемент, який ми не транспілюємо, це модулі ES. CommonJS - це стандарт модуля, який Node JS популяризував і продовжує використовувати сьогодні. Ми імпортуємо за допомогою ключового слова require, а експортуємо через module.export. Ось як він порівнюється з синтаксисом модуля ES. Тож у нашій збірці ми будемо використовувати Babel для транспіляції модуля ES у стиль CommonJS, щоб він міг використовуватися Node. Чому ми повинні публікувати наш пакет npm, використовуючи синтаксис модуля ES? Ось чому. Модулі ES статично аналізуються. Це звучить дуже заплутано, але насправді це означає, що код, на який посилається наш компонент, має бути детермінованим. Він не може змінитися під час виконання. Це важливо, оскільки воно забезпечує кращу підтримку автозаповнення в сучасних редакторах, оскільки чітко і чітко визначено, який код знаходиться в області дії в даному файлі. Це обмеження також важливе, оскільки воно означає, що сучасні пакети, такі як Webpack 2 і Rollup, можуть виконувати струшування дерева, також відоме як видалення мертвого коду. Сучасні пакети можуть читати модулі ES і розумно пропускати не викликаний код. Струшування дерева бажано, оскільки воно зменшує розмір пакета, усуваючи код, який ми все одно не запускали, тож, наприклад, якщо ми публікуєте компонент із 200 рядками коду, і хтось не використовує всі функції, деякі з цих невикористаних рядків коду можна виключити з остаточного пакета, щоб заощадити пропускну здатність і скоротити час розбору.

2.10.6. Збірка UMD

Іншим популярним альтернативним типом виводу, який слід розглянути, є UMD. UMD означає універсальне визначення модуля. Його називають універсальним, тому що це визначення модуля, яке може працювати в будь-якому місці і може інтегруватися з іншими популярними визначеннями модулів. Збірка UMD дозволить людям використовувати нашу бібліотеку компонентів, просто додавши тег сценарію у верхній частині сторінки. Зі збіркою UMD ми надаємо бібліотеку компонентів як глобальну змінну. UMD — це лише один із багатьох стилів модулів JavaScript. Коротко обговоримо інші. Існує IIFE, що розшифровується як вираз функції, що викликається негайно, є AMD, що означає визначення асинхронного модуля. Цей стиль був популяризований RequireJS, і сьогодні він значною мірою втратив популярність. Є CommonJS, який популяризував Node. Є модулі ES, які ми обговорювали, і, нарешті, є UMD, універсальне визначення модуля. UMD, як випливає з його назви, є універсальним форматом модуля, який взаємодіє з усіма наведеними вище альтернативними стилями модулів, то чому ж UMD важливий для компонентів, які можна використовувати повторно? Ну, просто. Якщо ми публікуємо свій компонент у форматі UMD, люди, які зараз використовують будь-який із популярних стилів модулів, можуть використовувати наш компонент. Є й інші переваги збірки UMD. Основна перевага збірки UMD полягає в тому, що вона дає можливість використовувати наш код, просто вставивши тег сценарію у верхній частині сторінки. UMD надає глобальну змінну, на яку наші користувачі можуть посилатися, тому будь-кому, хто хоче спробувати наш компонент, не потрібно налаштовувати процес збірки або працювати над його інтеграцією в існуючий процес. UMD підтримує просто додавання тегу сценарію, щоб це зробити. Цей підхід також є дружнім до експериментів і повстанців JavaScript в Інтернеті, таких як JSFiddle, JS Bin та різних альтернатив, але для внутрішніх компонентів UMD, ймовірно, непотрібний, а для загальнодоступних компонентів UMD є дружнім способом

гарантувати, що кожен може використовувати наші компоненти, навіть якщо вони вирішили залишитися за межами екосистеми npm і Node. Якщо ми вирішили використовувати UMD, нам також потрібно визначити назву глобальної змінної, яку ми надаєте. Найпопулярнішою умовою для компонентів React є назва компонента PascalCased. До речі, лише одна з п’яти найкращих бібліотек компонентів React пропонує збірку UMD, тому це, безумовно, не є обов’язковою умовою.

Висновок до розділу 2

У цьому розділі ми розглянули два ключових рішення щодо налаштування середовища розробки компонентів, які можна використовувати повторно. Перше рішення — створити бібліотеку чи окремі компоненти. Ми створюємо бібліотеку. Друге рішення складніше: вибрати основу середовища розробки. Ми побачили, що існує понад 100 шаблонів React, що є багато цікавих бібліотек, орієнтованих на документацію, як React Storybook, і, звичайно, ми завжди можемо вирішити створити власну для повного контролю над процесом, але ми вирішили використовувати create-react-app, оскільки це потужна основа для швидкого початку роботи.

Ми створили багату документацію з вихідного коду, читаючи реквізити та коментарі. Усі наші документи автоматично генеруються та перебудовуються щоразу, коли ми натискаємо Зберегти. Документи надають увесь необхідний контекст для наших користувачів, включаючи опис, робочі інтерактивні приклади, виділений синтаксис код, який використовується для кожного прикладу, а також список реквізитів, включаючи опис того, для чого призначений кожен реквізит, чи потрібний він, і його значення за замовчуванням.

Ми обговорили багато ключів до міцного, багаторазового дизайну компонентів, наприклад, уникати слабких елементів обгортки, вказувати PropType та значення за замовчуванням, не кодувати HTML-ідентифікатори

жорстко, розглянути об’єкти конфігурації та переконатися, що кожен із наших компонентів має чіткі, єдина відповідальність. Потім представили концепцію атомного дизайну. Ми дізналися, що атомний дизайн використовує наукові терміни для опису дизайну компонентів. Ми побачили, що цей словник корисний, оскільки він допомагає нам думати про нашу бібліотеку компонентів у кількох шарах абстракції. Таке мислення заохочує повторне використання, тому що ми спонукаємо думати про компоненти нижчого рівня, які могли б складатися в молекули та організми вищого рівня. Він чітко передає взаємозв’язки між компонентами та допомагає передати намір, надаючи кожному єдине слово для опису базового рівня компонента, який ми створюємо.

Атоми — це найменші, найосновніші компоненти, які ми створимо. Вони є будівельними блоками для великих абстракцій вищого рівня, таких як молекули та організми, які ми розглянемо в наступному модулі. Продуманий дизайн атома має вирішальне значення, оскільки атоми становлять основу нашої бібліотеки компонентів для багаторазового використання. Вони часто складаються з інших компонентів для створення абстракцій вищого рівня, а атоми можуть бути корисними для інкапсуляції основних думок про зовнішній вигляд, відчуття та поведінку. Ми згадав кілька важливих порад при проектуванні атомів.

Молекула — це група атомів. Він упорядковує атоми, наші основні будівельні блоки, у корисну абстракцію вищого рівня. Добре розроблені молекули підвищують узгодженість, і вони допомагають зменшити втому від прийняття рішень, інкапсулюючи довгий список думок. У цьому модулі ми створили дві повторно використовувані молекули, TextInput і PasswordInput, і ми фактично використали TextInput за лаштунками для створення введення пароля. Це був приклад композиції над успадкуванням. PasswordInput обернув TextInput, тому PasswordInput є спеціалізацією компонента TextInput, і тепер ми готові використати ці молекули для створення компонентів для повторного використання з ще більш високим рівнем абстракції.

Ми використали атоми та молекули, які ми створили в попередніх модулях, щоб створити багаторазовий реєстраційний організм. Ми розглянули кілька ключових порад. Сильні думки – це особливість. Вони допомагають узгодженості, зменшують помилки та допомагають людям зрозуміти наш компонент. Не потрібно виставляти всі реквізити компонентів дитини. Потрібно робити вибір і намагатися тримати свій організм німім. Не потрібно створювати міні-програми, які містять виклики API, перевірки аутентифікації, пісок тощо. Це призводить до втрати пропускної спроможності та зменшує можливість повторного використання, впроваджуючи проблеми, пов’язані з конкретними програмами.

Ми розглянули чотири популярні підходи до стилізації React. Як ми бачили, правильної відповіді немає. Наше рішення зводиться до пріоритетів і цінностей. Існують компроміси, які слід враховувати при кожному підході. Ми також обговорили деякі основні рішення щодо стилів, такі як схеми найменування, які корисні, якщо ми робимо скомпільований CSS або звичайний CSS, і ми завершили коротке обговорення тем. Розглядаємо можливість доставки наших компонентів без стилів або принаймні з можливістю імпортування компонента без стилів. Це особливо корисно, якщо ми створюємо компоненти для громадськості, але пам’ятаємо, що стандарту тематики не існує, тому технічний підхід, який ми використовуємо для створення тем, буде відрізнятися залежно від обраної нами технології стилізації.

РОЗДІЛ 3. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ

3.1. Створення репозиторію GitHub

Нам потрібно створити репозиторій на GitHub, щоб ми могли присвятити свою роботу на цьому шляху. Якщо у нас немає облікового запису GitHub, ми можемо зареєструватися безкоштовно, а після входу натиснути плюс у верхній частині екрана, щоб створити репозиторій. Назвемо наш репозиторій rrc-diploma. На gitignore, ми виберемо Node, оскільки ми працюємо з Node, і це гарантує, що файли, пов'язані з Node, ігноруються GitHub. Потім нам потрібно вибрати ліцензію MIT (рис. 3.1).

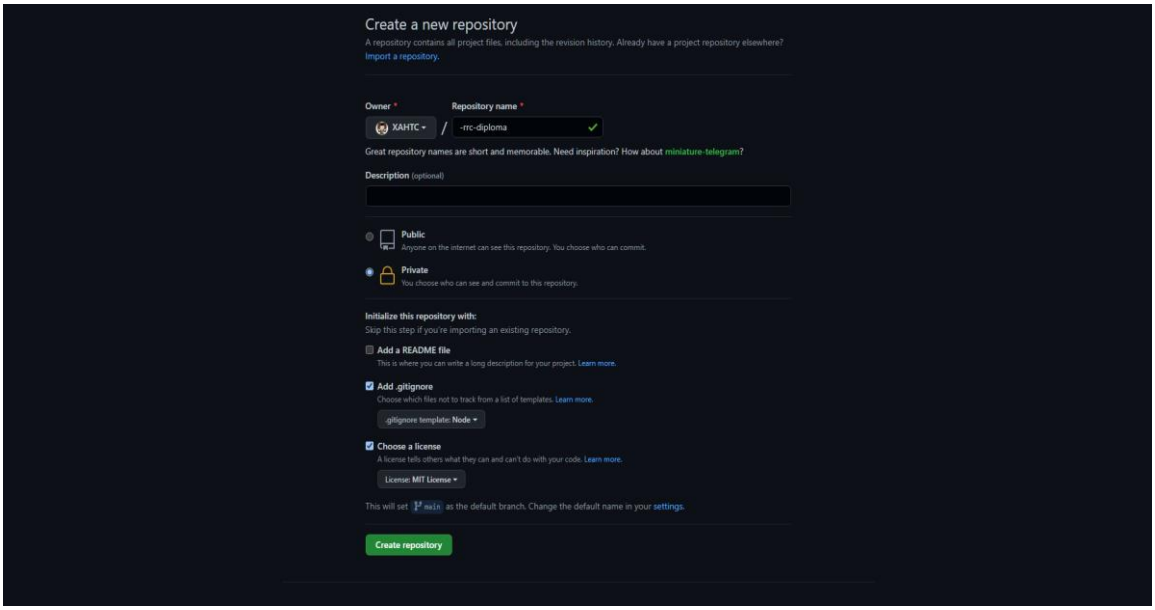


Рис. 3.1. Створення репозиторію GitHub

3.2. Створення проекту за допомогою create-react-app

Тепер, створимо основу для нашого середовища розробки за допомогою create-react-app. Ми будемо використовувати VSCode, він має вбудований термінал, інтеграцію git та швидку продуктивність. Далі в терміналі ми використовуємо команду git clone, а потім нам потрібно перейти на GitHub, натиснути зелену кнопку «Клонувати або завантажити», скопіювати цю URL-адресу та вставити її в термінал (рис. 3.2).

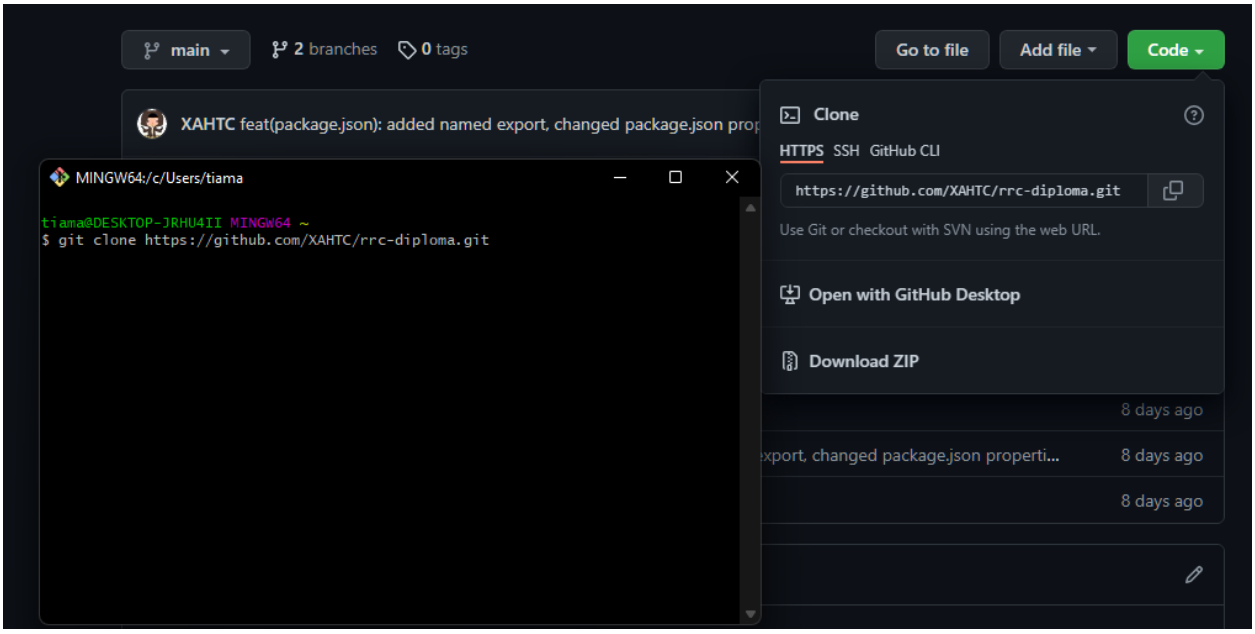


Рис. 3.2. Клонування та завантаження репозиторію на ПК

Як тільки ми це зробимо, ми зможемо перейти до цього каталогу і клонувати репозиторій GitHub на наш ПК. Тепер, коли у нас відкрито сховище в редакторі, встановимо наші залежності. Ми почнемо з встановлення create-react-app, тому ми використовуємо команду “npx create-react-app ./” , де “./” повідомляє йому створити програму React прямо тут, у моєму поточному каталозі. Далі нам потрібно відкрити папку, яку ми щойно створили у VSCode. Ще одна річ, яка нам подобається у VSCode, — це те, що після того, як відкриваємо проект, термінал досить розумний, щоб відкрити правильний каталог, тому нам не потрібно вести cd. Якщо ми використовуємо якийсь

окремий термінал, то кожен раз, коли ми відкриваємо термінал, нам потрібно буде обов'язково відкривати його у відповідному каталозі, лише на один крок менше, щоб турбуватися тут. Подивимося навколо і подивимося, що створив для нас додаток create-react-app. У вихідному каталозі ми знаходимо наш вихідний код. У нас є файл індексу, який є нашою точкою входу, деякий CSS, фактична програма, яка є лише не великим прикладом програми, логотипом, і насправді тут немає нічого особливо цікавого. Тепер у додатку create-react-app є те, що воно об'єднує все для нас - наш package.json. У нас є посилання на react і react-dom. Це було досить дружньо, щоб встановити назву на нашій package.json, а потім у нас є кілька скриптів для запуску програми, створення її для виробництва, запуску наших тестів і видалення. Додаток Create-react-app також має конвенцію щодо розміщення загальнодоступних файлів у папці під назвою public, тому index.html знаходиться там. Тут теж не так багато, щоб побачити. Є лише корінь програми, а також іконка favicon.ico, тож вони автоматично копіюються в виробничу збірку, оскільки вони знаходяться в загальнодоступному каталозі. Тож повернемося до package.json. Ми повинні мати можливість сказати, npm start і подивитися, як запускаються наші програми. Чудово, і ось воно. Тепер, якщо ми повернемося до цього, ми помітимо, що список залежностей тут дуже малий, тому що create-react-app зараз володіє всіма залежностями, але якщо ми відкриємо node_modules, ми побачимо величезну кількість node_modules, які були встановлені та на які посилаються позаду сцени за допомогою програми create-react-app. Це те, що так чудово в цій установці: вона надійна, і вона абстрагує велику частину тієї складності, яку нам не довелося налаштовувати самим. Він налаштував Webpack, Babel, ESLint і Jest для тестування, і все це за лаштунками для нас, і приємна річ у створенні-реагуванні-додатку, що володіє цими пакетами, це те, що ми можемо легко оновлюватися з часом. Ми також повинні мати можливість запускати наші тести. Якщо ввести команду "npm t" або "npm test", будь-який з них спрацює, він визначить, який тест потрібно запустити, заглянувши в каталог джерел і знайшовши файл з розишренням ".spec.js" або

“.test.js”, тож ми можемо побачити, що у нас є вбудована автоматична настройка тестування, яка просто працює (рис. 3.3).

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Рис. 3.3. Результат створення проекту за допомогою create-react-app

3.3. Розпакування create-react-app

Перш ніж ми завершимо налаштування середовища розробки, розпаковуємо наш проект для більш гнучкого налаштування пізніше. Приємна особливість програми create-react-app полягає в тому, що вона абстрагує

складність Webpack, Babel, ESLint, Jest та десятків інших пакетів. Ця абстракція означає, що ми можемо легко оновити create-react-app, оскільки всі ці базові пакунки, конфігурації та сценарії будуть виправлені, оновлені та покращені. Це великий плюс але, оскільки вони абстраговані, ми також втрачаємо можливість повністю налаштовувати їх конфігурацію. Ось чому create-react-app забезпечує зручну функцію під назвою eject. Коли ми використовуємо eject, ми отримуємо повний доступ до основних інструментів, але ми також беремо на себе повну відповідальність за керування оновленнями з часом, тому це компроміс. Цю команду не можна скасувати. Для цього ми виконаємо команду “npm run eject”. Ми отримаємо запит, що ми впевнені в цьому і підтвердимо його. Після виконання, ми помітимо, що package.json став набагато більшим, тому що тепер ми можемо безпосередньо бачити всі пакети npm, від яких залежали. Ми також бачимо, що наші сценарії змінилися. Тепер вони викликають node і вказують певні сценарії, які знаходяться в папці “scripts”, які ми раніше не могли бачити. Також ми бачимо конфігурацію Jest, конфігурацію Babel і конфігурацію ESLint (рис. 3.4).

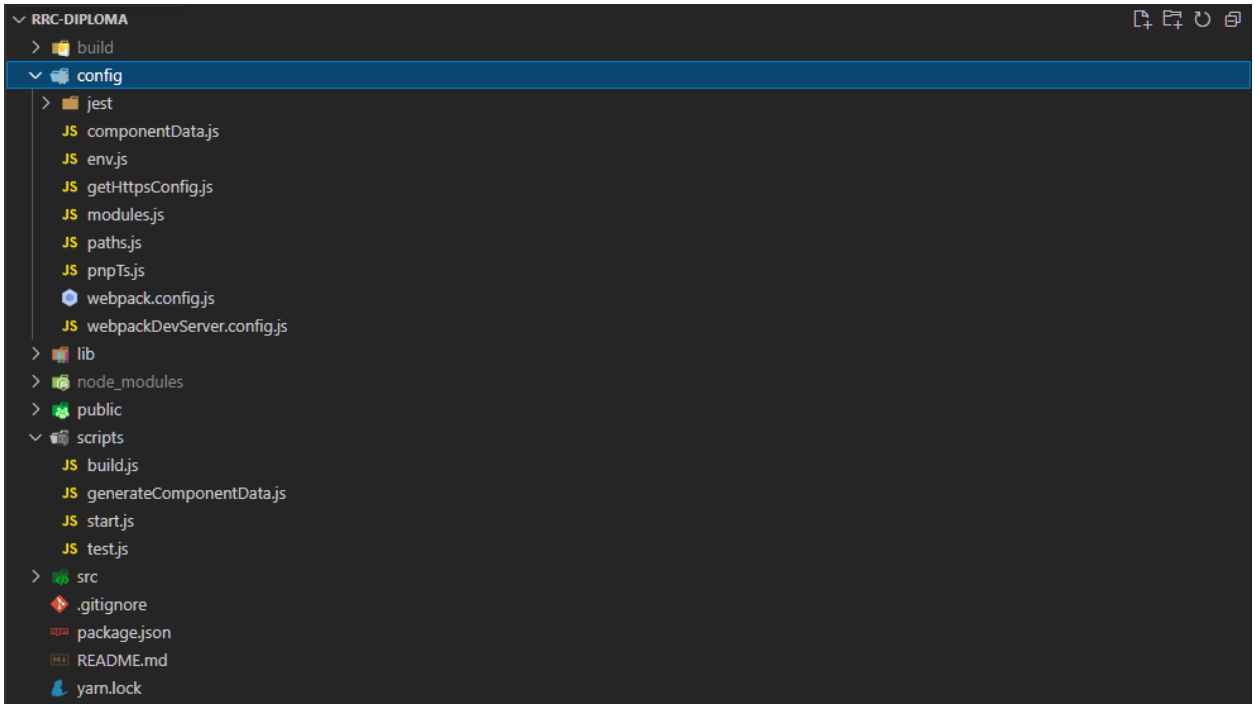


Рис. 3.4. Результат виконання команди “npm run eject”

Якщо ми перейдемо сюди до папки scripts, ми побачимо сценарії, які насправді запущені в програмі create-react-app. Тут є сценарій запуску, сценарій, який запускає тести, сценарій, який запускає збірку. Тож це сценарії, які запускаються, коли ми виконуємо необхідні команди: “npm start”, “npm run build”, щоб фактично виконати виробничу збірку, і “npm t” або “npm test” для запуску наших тестів. Ще одна важлива папка - це config. Ось де знаходиться конфігурація для create-react-app, тож тепер ми можемо побачити, що ми насправді маємо доступ до базової конфігурації Webpack, до налаштувань середовища, до конфігурації Jest, до шляхів і поліфілів, які завантажуються поза лаштунками, тож ми маємо всю необхідну силу, і хороша новина полягає в тому, що ми повинні мати можливість прийти сюди і сказати, “npm start”, і він повинен продовжувати працювати так само, як і раніше, за винятком того, що тепер ми маємо повний доступ до конфігурації, так, ми можемо продовжувати успішно бачити завантаження нашої програми. Тепер ми не створюємо справжню програму. Ми просто хочемо створити бібліотеку компонентів для повторного використання.

3.4. Налаштування платформи документації

Створимо багату користувацьку документацію компонента React. Існує ряд цікавих і зручних інструментів для створення документації. Наша програма буде сайтом документації, тому створимо дві папки верхнього рівня під джерелом. Один ми назвемо docs. Тут буде розміщено наш сайт з документацією, а друга папка, яку ми створимо, буде називатися components (рис. 3.5).

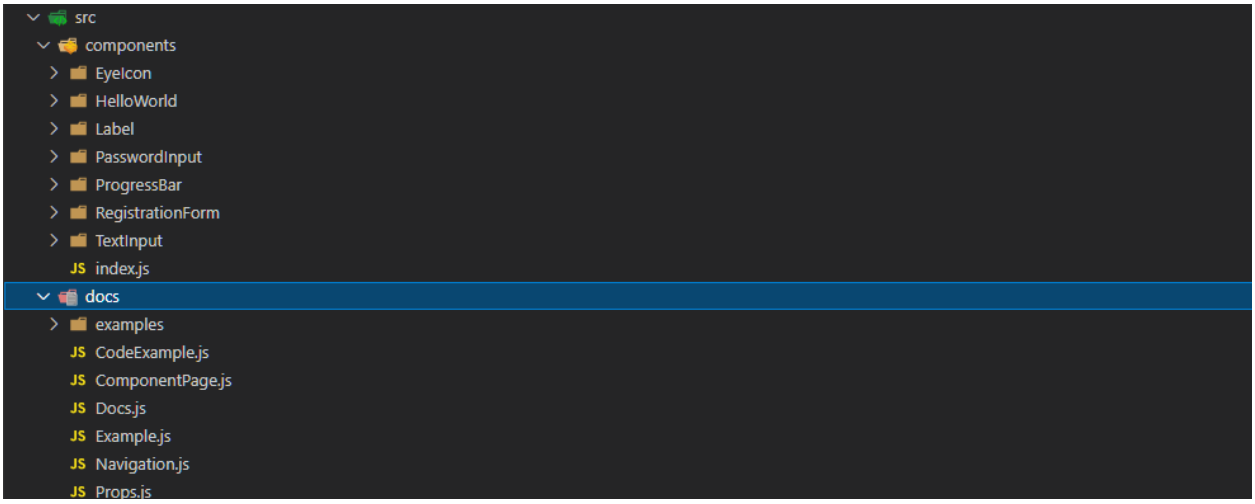


Рис. 3.5. Створення папок docs та components

Тут будуть зберігатися багаторазові компоненти, які ми збираємося створити незабаром у майбутньому модулі. У папку docs ми збираємося додати всі компоненти, необхідні для відображення документації нашого компонента.

3.5. Створення метаданих компонентів

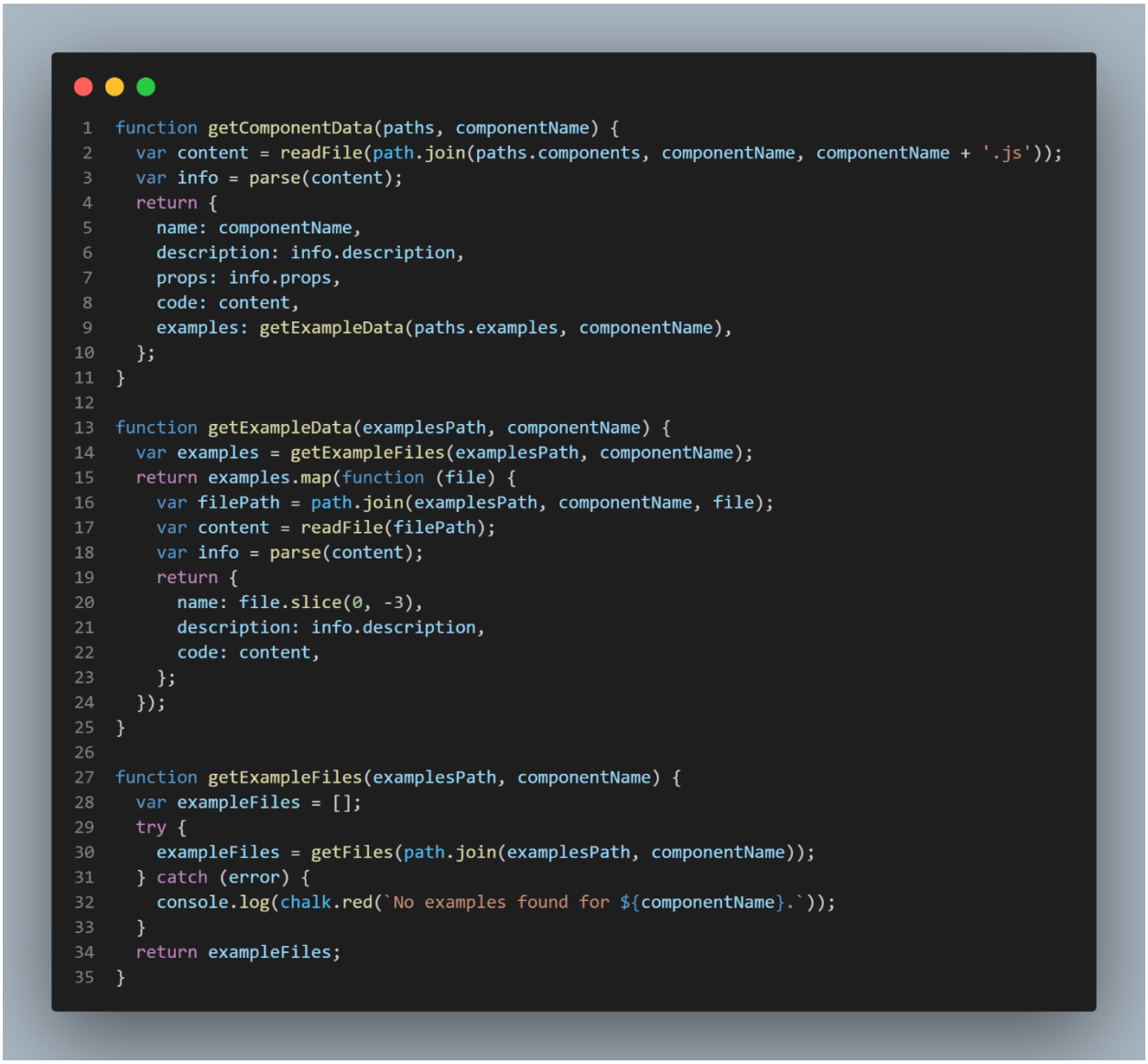
Щоб почати генерувати та відображати нашу документацію, нам потрібно встановити чотири пакунки npm. Перші три – це react-docgen, Chokidar і npm-run-all. Встановлюємо їх як залежності для розробки, тому використаємо флаг -D. Потім ми також встановлюємо React highlight і зберігаємо це як виробничу залежність. Після встановлення, перейдемо до папки скриптів, і створимо новий файл під назвою generateComponentData.js (рис 3.6).

```
var fs = require('fs');
var path = require('path');
var chalk = require('chalk');
var parse = require('react-docgen').parse;
var chokidar = require('chokidar');

var paths = {
  examples: path.join(__dirname, '../src/', 'docs', 'examples'),
  components: path.join(__dirname, '../src/', 'components'),
  output: path.join(__dirname, '../config', 'componentData.js'),
};
const enableWatchMode = process.argv.slice(2) == '--watch';

if (enableWatchMode) {
  chokidar.watch([paths.examples, paths.components]).on('change', function (event, path) {
    generate(paths);
  });
} else {
  generate(paths);
}

function generate(paths) {
  var errors = [];
  var componentData = getDirectories(paths.components).map(function (componentName) {
    try {
      return getComponentData(paths, componentName);
    } catch (error) {
      errors.push(
        'An error occurred while attempting to generate metadata for ' +
          componentName +
          '. ' +
          error,
      );
    }
  });
  writeFile(
    paths.output,
    'module.exports = ' + JSON.stringify(errors.length ? errors : componentData),
  );
}
```

Рис. 3.6. Файл generateComponentData.js

Угорі ми імпортуємо fs і path, які є двома пакетами, які постачаються з Node. Це для читання файлової системи, шлях для роботи з шляхами. Ми будемо використовувати highlight для фарбування результатів командного рядка, а потім react-docgen – це те, що насправді привносить магію в цей процес. Це буде бібліотека, яка переглядатиме наші компоненти та витягуватиме метадані з коду нашого компонента. Нарешті, ми використовуємо пакет npm під назвою Chokidar, який дозволяє нам переглядати файли, а потім виконувати функцію міжплатформенним способом. Угорі ми оголошуємо три

важливі шляхи: шлях до папки з прикладами, шлях до вихідного коду компонентів і, нарешті, шлях, за яким ми хотіли б вивести наш файл метаданих. Тепер саме тут, у рядку 13, ми вникаємо в суть справи. Ось де розташована бізнес-логіка. Те, що ми робимо, це перевіряємо наявність прапора перегляду, оскільки цей процес за бажанням буде стежити за змінами в наших файлах, і якщо режим перегляду увімкнено, ми будемо використовувати Chokidar для перегляду шляху наших прикладів і наших компонентів. Щоразу, коли вони змінюються, ми відновимо наші метадані. Якщо режим годинника не ввімкнено, ми просто продовжимо і згенеруємо виклик лише один раз. Тож прокрутимо вниз і подивимося, як працює генерація. Функція генерування відстежує масив помилок, якщо такі трапляються, і що ми робимо, так це отримуємо список каталогів, які знаходяться в нашій папці компонентів, ми відображаємо ці каталоги та отримуємо дані компонентів для кожної папки, і ми обертаємо це в блок try/catch на випадок, якщо ми допустимо будь-яку синтаксичну помилку. Тоді ми просто виведемо це в наш вихідний файл. Таким чином ми знаємо, коли ми зробили помилку, а також загортаючи це в спробу лову, дозволяє обробці помилок create-react-app обробляти будь-які проблеми, які виникають, тож таким чином наш процес продовжуватиме виконуватися, навіть якщо ми зробимо помилка, коли ми створюємо наші компоненти. Як тільки ми отримаємо цей масив даних компонента, ми запишемо цей масив тут, у рядку 33, у наш вихідний файл. Ми використовуємо JSON.stringify, щоб серіалізувати ці дані в JSON, щоб ми могли прочитати їх за допомогою React. Якщо є якісь помилки, ми поміщаємо помилки в цей файл, інакше ми розміщуємо фактичні дані компонента. Тепер перейдемо на рівень глибше (рис. 3.7), тому що тут ми отримуємо каталоги, ми відображаємо їх, і ми отримуємо дані компонентів для кожного з компонентів, які ми знаходимо в нашому каталозі компонентів, тож подивимося, як дані компонента працюють.



```
function getComponentData(paths, componentName) {
  var content = readFile(path.join(paths.components, componentName, componentName + '.js'));
  var info = parse(content);
  return {
    name: componentName,
    description: info.description,
    props: info.props,
    code: content,
    examples: getExampleData(paths.examples, componentName),
  };
}

function getExampleData(examplesPath, componentName) {
  var examples = getExampleFiles(examplesPath, componentName);
  return examples.map(function (file) {
    var filePath = path.join(examplesPath, componentName, file);
    var content = readFile(filePath);
    var info = parse(content);
    return {
      name: file.slice(0, -3),
      description: info.description,
      code: content,
    };
  });
}

function getExampleFiles(examplesPath, componentName) {
  var exampleFiles = [];
  try {
    exampleFiles = getFiles(path.join(examplesPath, componentName));
  } catch (error) {
    console.log(chalk.red(`No examples found for ${componentName}.`));
  }
  return exampleFiles;
}
```

Рис. 3.7. Продовження файлу generateComponentData.js.

Він читає файл і отримує вміст із цього файлу, а потім викликає аналіз. Тепер синтаксичний аналіз — це функція, яка постачається з react-docgen, тому ми справді говоримо: привіт, react-docgen, ідіть прочитайте вміст мого вихідного коду, а потім ми отримуємо метадані. Ми отримуємо метадані опису компонента, його реквізитів, а потім також беремо вміст цього файлу і поміщаємо його у властивість, яка називається кодом, тож тепер у нас є метадані про компонент, включаючи не лише опис та реквизити, а й сам вихідний код. Нарешті, ми зберігаємо список прикладів, і для цього ми викликаємо функцію getExampleData, щоб ми могли деталізувати на один рівень глибше, щоб побачити, як це працює. GetExampleData працює дуже

схоже на функцію, яку ми щойно розглянули тут, getComponentData. Основна відмінність полягає в тому, що в прикладах ми не маємо всіх однакових реквізитів, тому що приклад не має доданого масиву прикладів, він має лише назву, опис, а потім код. Знову ж таки, ми використовуємо react-docgen, щоб отримати метадані для прикладів, які ми створюємо, і таким чином ми матимемо всі метадані, які нам потрібні як для нашого вихідного коду наших компонентів, так і для прикладів, які ми будемо відображати для наших компонентів. Решта функцій – це лише базові обгортки функціональності Node (рис. 3.8).

```
function getDirectories(filePath) {
  return fs.readdirSync(filePath).filter(function (file) {
    return fs.statSync(path.join(filePath, file)).isDirectory();
  });
}

function getFiles(filePath) {
  return fs.readdirSync(filePath).filter(function (file) {
    return fs.statSync(path.join(filePath, file)).isFile();
  });
}

function writeFile(filePath, content) {
  fs.writeFile(filePath, content, function (err) {
    err ? console.log(chalk.red(err)) : console.log(chalk.green('Component data saved'));
  });
}

function readFile(filePath) {
  return fs.readFileSync(filePath, 'utf-8');
}
```

Рис. 3.8. Допоміжні функції в файлі generateComponentData.js.

Одна для отримання списку файлів із шляху, який передається, одна функція для отримання каталогів із шляху до файлу, інша функція для отримання списку файлів для шляху до файлу, функція для запису файлу до шляху до файлу, а потім функція для читання файлу, тож це насправді лише наші утиліти.

3.6. Скрипти npm

Кожного разу, коли ми вводимо npm, починаємо відкривати наші документи, нам потрібно генерувати дані компонента перед запуском нашого сайту документації, тому нам потрібно створити кілька сценаріїв npm, щоб автоматизувати цей процес. NPM-скріпти, є легкою альтернативою Gulp або Grunt, і вони є популярним способом автоматизації завдань під час роботи з Node. create-react-app постачається з трьома вбудованими сценаріями npm, start, який запускає програму, build, який створює його для виробництва, і test, який запускає наші тести.

Створимо нові скрипти (рис. 3.9):

"prestart": "npm run gen:docs",

"start:docs": "node scripts/start.js",

"gen:docs": "node scripts/generateComponentData.js",

"gen:docs-watch": "npm run gen:docs - --watch",

"build:docs": "node scripts/build.js",

"build:css": "cpx \"./src/components/**/*.css\" ./lib",

"build:lib": "npm run build:commonjs && npm run build:css"

"build:commonjs": "cross-env NODE_ENV=production babel ./src/components --out-dir ./lib --ignore spec.js",

prestart - виконує запуск скірпта, що генерує документацію компонентів

start:docs - запускає наш проект

gen:docs - генерує документацію

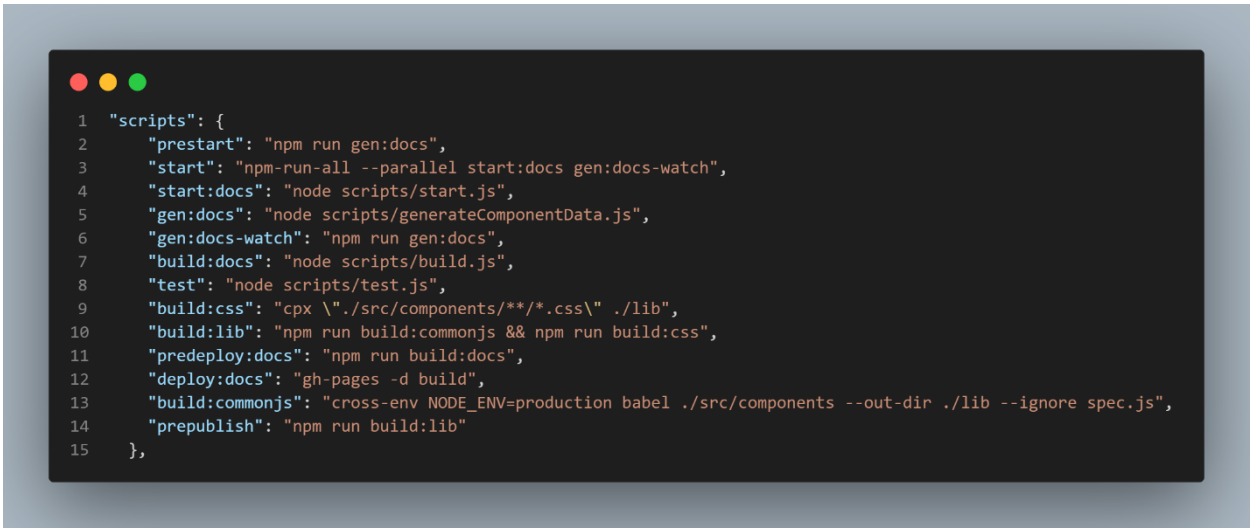
gen:docs-watch - запускає генерацію документі з прапором --watch

build:docs - створює білд

build:css - створює білд CSS файлів

build:lib - виконує створення бібліотеки та стилів до неї

build:commonjs - створює білд бібліотеки з використанням транспілятора babel до ES2015



```
"scripts": {
    "prestart": "npm run gen:docs",
    "start": "npm-run-all --parallel start:docs gen:docs-watch",
    "start:docs": "node scripts/start.js",
    "gen:docs": "node scripts/generateComponentData.js",
    "gen:docs-watch": "npm run gen:docs",
    "build:docs": "node scripts/build.js",
    "test": "node scripts/test.js",
    "build:css": "cpx \"./src/components/**/*.css\" ./lib",
    "build:lib": "npm run build:commonjs && npm run build:css",
    "predeploy:docs": "npm run build:docs",
    "deploy:docs": "gh-pages -d build",
    "build:commonjs": "cross-env NODE_ENV=production babel ./src/components --out-dir ./lib --ignore spec.js",
    "prepublish": "npm run build:lib"
  },
```

Рис. 3.9. Створення нових скриптів у файлі package.json

3.7. Створення компонента Docs

Щоб відобразити метадані наших компонентів, почнемо зі створення компонента Docs, оскільки це компонент верхнього рівня для нашої документації. Перейдемо до нашої папки документів і створимо новий файл, і ми назвемо його Docs.js (рис. 3.10). Ми викликали два дочірніх компоненти, Navigation і ComponentPage. Остаточний імпорт – це посилання на файл даних компонента, який ми генеруємо щоразу, коли запускаємо програму або зберігаємо. Нижче видно, як ми його використовуємо. Тепер нам потрібна маршрутизація для наших документів, але наші потреби дуже прості, тому ми не використовували бібліотеку маршрутизації. Натомість ми просто утримуємо перший сегмент URL-адреси в стані. Ми використовуємо URL-адреси на основі, щоб це було просто. Щоразу, коли URL-адреса змінюється, ми будемо оновлювати стан. Як ми можемо бачити тут, у componentDidMount, ми додаємо слухач події hashchange. Внизу, в рендері, ми збираємося використовувати маршрут, який зберігаємо в стані, щоб отримати відповідний компонент. За домовленістю маршрут і URL-адреса повинні відповідати нашій компонента. Ця умова зробить наш бруд маршрутизації простим, і якщо маршрут не вказано, ми просто відображаємо перший компонент у списку. Таким чином, ми обробили маршрутизацію на стороні клієнта менш

ніж за 10 рядків коду. Тепер у цьому компоненті дуже мало jsx, оскільки він просто викликає дочірні компоненти. Ми передаємо список імен компонентів до компонента Navigation, а поточний вибраний компонент до компонента ComponentPage, тому в основному цей компонент містить 10 рядків маршрутизації та делегування декільком дочірнім компонентам.

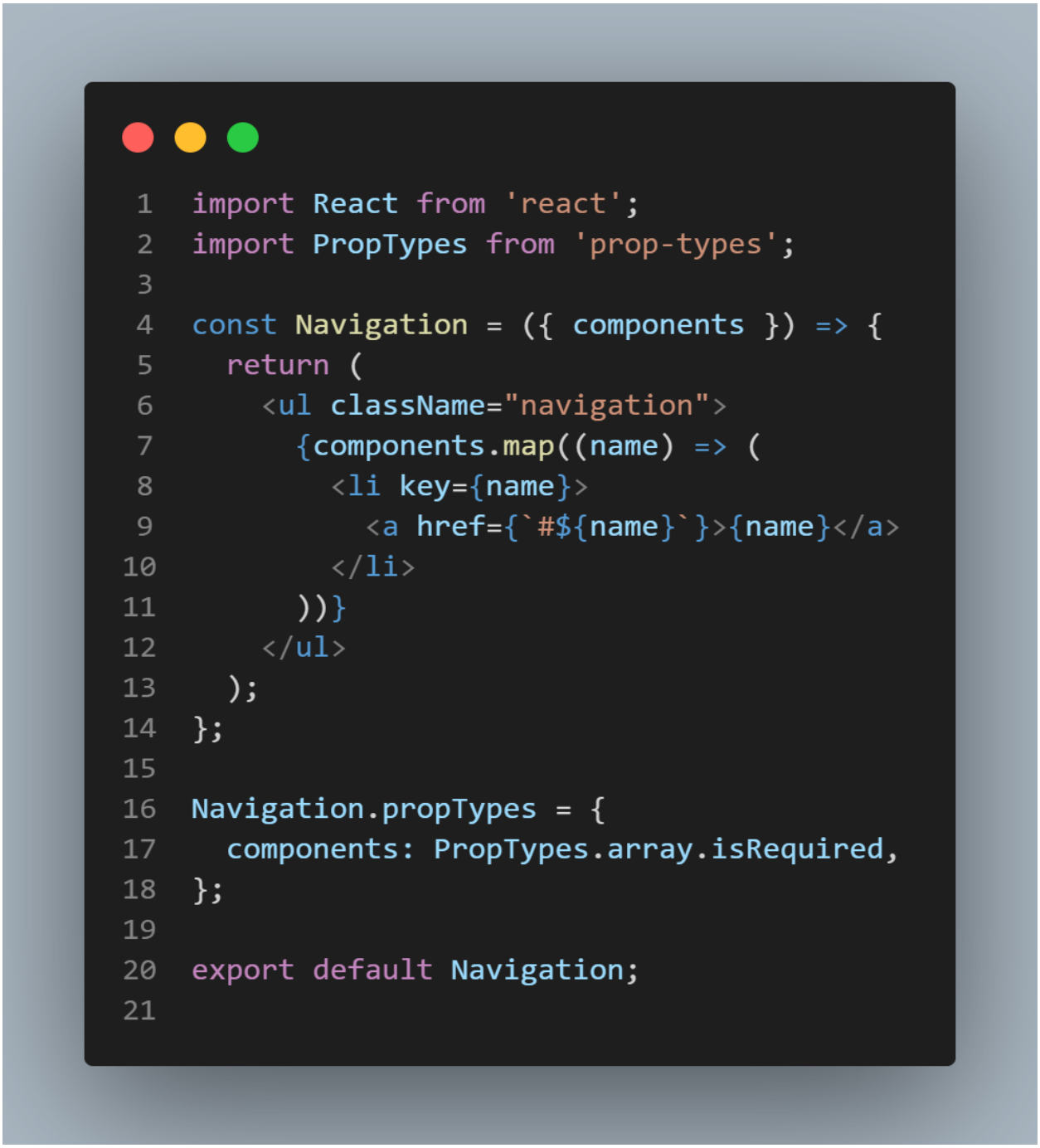
```
import { Component } from 'react';
import Navigation from './Navigation';
import ComponentPage from './ComponentPage';
import componentData from '../../config/componentData';

export default class Docs extends Component {
  constructor(props) {
    super(props);
    this.state = {
      route: window.location.hash.substr(1),
    };
  }

  componentDidMount() {
    window.addEventListener('hashchange', () => {
      this.setState({ route: window.location.hash.substr(1) });
    });
  }

  render() {
    const { route } = this.state;
    const component = route
      ? componentData.filter((component) => component.name === route)[0]
      : componentData[0];

    return (
      <div>
        <Navigation components={componentData.map((component) => component.name)} />
        <ComponentPage component={component} />
      </div>
    );
  }
}
```

Рис. 3.10. Компонент Docs

3.8. Створення компонента Navigation

Додамо компонент Navigation (рис. 3.11). Всередині тут дуже мало логіки. У нас невпорядкований список, який відображатиме список наших

компонентів, тому ми будемо відображати масив компонентів, який передано, і просто відобразимо прив'язку для кожного з них. За умовою, ім'я компонента буде тим, що використовується в URL-адресі, тому ми просто вставлю хеш в URL-адресу, а потім назву компонента. Насправді більше тут нема чого дивитися.



```
1  import React from 'react';
2  import PropTypes from 'prop-types';
3
4  const Navigation = ({ components }) => {
5    return (
6      <ul className="navigation">
7        {components.map((name) => (
8          <li key={name}>
9            <a href={`#${name}`}>{name}</a>
10         </li>
11       ))}
12     </ul>
13   );
14 };
15
16 Navigation.propTypes = {
17   components: PropTypes.array.isRequired,
18 };
19
20 export default Navigation;
21
```

Рис. 3.11. Компонент Navigation

3.9. Створення компонента ComponentPage

Створимо компонент ComponentPage (рис. 3.12). Можемо бачити, що ми імпортуємо приклад і реквізит, а потім всередині отримуємо об'єкт компонента, який передається вниз, а потім ми деструктуруємо, лише щоб виклики були акуратними та короткими. Ми збираємося працювати з назвою компонентів, описом, масивом реквізитів і масивом прикладів тут, на сторінці компонента. Тому ми показуємо назву, опис, а потім заголовок для будь-яких прикладів. Якщо якісь приклади існують, ми продовжуємо показувати їх. Якщо ні, ми просто показуємо дружнє повідомлення, а потім показуємо таблицю реквізитів. Якщо є реквізити, ми їх показуємо, а якщо ні, то просто показуємо повідомлення про те, що компонент не приймає реквізитів.

```
import PropTypes from 'prop-types';
import Example from './Example';
import Props from './Props';

const ComponentPage = ({ component }) => {
  const { name, description, props, examples } = component;
  return (
    <div className="componentpage">
      <h2>{name}</h2>
      <p>{description}</p>
      <h3>Example{examples.length > 1 && 's'}</h3>
      {examples.length > 0
        ? examples.map((example) => (
            <Example key={example.code} examples={example} componentName={name} />
          ))
        : 'No examples exist'}
      <h3>Props</h3>
      {props ? <Props props={props} /> : 'This component accepts no props.'}
    </div>
  );
};

ComponentPage.propTypes = {
  component: PropTypes.object.isRequired,
};

export default ComponentPage;

```

Рис. 3.12. Компонент ComponentPage

3.10. Створення компонента Example

Створимо компонент Example (рис. 3.13). Цей компонент відображатиме приклади наших компонентів у дії. Тут відбувається трохи більше, тому що цей компонент має невелику частину стану, який відстежує, чи маємо ми показувати приклад коду чи ні, і коли ми клацнемо прив'язку внизу під час візуалізації, він перемикає setState, і це просто логічне значення, яке відстежує, чи повинен відображатися код чи ні. Тепер ми повинні зазначити, що ми використовуємо статичну властивість класу, яка дуже схожа на стрілочну функцію, але суть цього в тому, що нам не потрібно робити явне прив'язування тут, у конструкторі. Це не призведе до зміни значення this, тому значення this тут все одно буде прив'язано до відповідного екземпляра, оскільки ми використовуємо тут статичний клас. Це експериментальна функція, але вона буде перенесена Babel для нас. Тепер найпримітніша функція в цьому файлі знаходиться тут, у рядку 21. Саме тут відбувається магія. нам динамічно потрібен компонент Example. Динамічне використання компонента корисно, тому нам не потрібно вручну додавати імпорт для кожного прикладу компонента, який ми створюємо. Однак динамічне імпортування компонентів вимагає іншого підходу, ніж імпорт ES, який ми досі використовували у верхній частині цих файлів. Ми використовуємо тут функцію вимоги стилю CommonJS замість імпорту ES цілком навмисно. Це необхідно, оскільки нам динамічно потрібен файл. Імпорт ES статично аналізується за конструкцією, тому ми не можемо використовувати імпорт для динамічного імпорту компонентів прикладу, і за замовчуванням в кінці, оскільки ми експортуємо всі наші компоненти як експорт за замовчуванням, тому для доступу до них через CommonJS нам потрібно явно вказати. за замовчуванням тут, і без цієї примхи решта цього компонента досить проста. Ми відображаємо опис прикладу, якщо він є, і показуємо сам компонент прикладу. Також ми маємо обробник кліків toggleCode. Якщо showCode має значення true, мітка на нашому прив'язці

змінюється залежно від того, чи є showCode true чи false, а потім якщо showCode відповідає дійсності, ми відображаємо код.

```
import { Component } from 'react';
import PropTypes from 'prop-types';
import CodeExample from './CodeExample';
export default class Example extends Component {
  constructor(props) {
    super(props);
    this.state = { showCode: false };
  }

  toggleCode = (event) => {
    event.preventDefault();
    this.setState((prevState) => ({ showCode: !prevState.showCode }));
  };

  render() {
    const { showCode } = this.state;
    const { code, description, name } = this.props.examples;
    const ExampleComponent = require(`./examples/${this.props.componentName}/${name}`).default;
    return (
      <div className="example">
        {description && <h4>{description}</h4>}
        <ExampleComponent />
        <p>
          <a href="#" onClick={this.toggleCode}>
            {showCode ? 'Hide' : 'Show'} Code
          </a>
        </p>

        {showCode && <CodeExample>{code}</CodeExample>}
      </div>
    );
  }
}

Example.propTypes = {
  examples: PropTypes.object.isRequired,
  componentName: PropTypes.string.isRequired,
};

```

Рис. 3.13. Компонент Example

3.11. Створення компонента Props

Створимо компонент під назвою Props (рис. 3.14). Таким чином, цей компонент буде відображати інформацію про реквізити на наших компонентах. Таким чином, ми можемо побачити, що це таблиця, яка має заголовки «Ім’я», «Опис», «Тип», «За замовчуванням» і «Обов’язковий», і що

вона робить, так це відображає ключі на реквізитах, які ми знаходимо, тож це ключі з нашого файлу метаданих, а потім відображає ці дані в табличній сітці. Тут немає нічого складного. Одна річ, яка трохи спантеличує, це те, що єдиний реквізит у цьому компоненті називається props, оскільки він є масивом реквізитів. Ми також повинні створити жодних прикладів використання для нашого одного компонента, але ми створюємо папку, де будуть ці приклади, яка буде в документах, і ця папка буде називатися прикладами, тому, як ми додамо приклади для наших компонентів ми розмістимо їх тут у docs/examples. Перш ніж зробити це, нам потрібно зробити ще одне налаштування конфігурації. Create-react-app не дозволяє імпортувати файли за межі вихідного каталогу за замовчуванням. Це допоможе захистити нас від випадкового імпорту файлів за межі вихідного коду, але саме це ми хочемо зробити в даному випадку, оскільки ми імпортуємо згенерований componentData.js, який зберігається в папці конфігурації. Тож відкриємо файли конфігурації Webpack і внесемо зміни. Спочатку ми перейдемо до webpack.config.js. Тут всередині знайдемо moduleScopePlugin. Закоментуємо цей рядок, щоб вимкнути цю перевірку. Таким чином ми можемо імпортувати файли з-за меж вихідної папки. Тепер, коли ми все це налаштували, ми зможемо запустити термінал і ввести npm start і побачити це в дії. Ми бачимо, що він створив метадані наших компонентів, і вони з’являються, і наша програма успішно завантажується.

```
import PropTypes from 'prop-types';

const Props = ({ props }) => {
  return (
    <table className="props">
      <thead>
        <tr>
          <th>Name</th>
          <th>Description</th>
          <th>Type</th>
          <th>Default</th>
          <th>Required</th>
        </tr>
      </thead>
      <tbody>
        {Object.keys(props).map((key) => (
          <tr key={key}>
            <td>{key}</td>
            <td>{props[key].description}</td>
            <td>{props[key].type.name}</td>
            <td>{props[key].defaultValue && props[key].defaultValue.value}</td>
            <td>{props[key].required && 'X'}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};

Props.propTypes = {
  props: PropTypes.object.isRequired,
};

export default Props;

```

Рис 3.14. Компонент Props

3.12. Аліас Webpack

Зараз ми вводимо довгий, потворний імпорт у наших прикладах. Це дратує введення, і, що більш важливо, це проблема, оскільки наша документація повинна відображати фактичний шлях, який хтось повинен ввести для імпорту наших компонентів. Таким чином наші приклади коду дійсно можна скопіювати та вставити в робочий додаток. Зараз, звісно, ми ще не опублікували у виробництві на npm, тому важко здогадатися, яким саме буде

цей шлях. Щоб це сталося, нам потрібно налаштувати псевдонім Webpack, тож відкриємо папку config і відкриємо webpack.config.js. У розділі resolve, нам потрібно додати невеликий псевдонім прямо тут, який значно полегшить наше життя. Ми добавимо, @rrc-diploma (рис 3.15).

Рис. 3.15. Створення аліасу webpack

Тепер ми можемо відкрити наші компоненти с прикладами і змінити шлях, щоб він був набагато більш дружнім, ніж був раніше. Замість цього ми скажемо @rrc-diploma/... Також зробимо ще одне налаштування. Щоб уникнути повторень {ComponentName}, перейдемо до нашої папки компонентів, і ми додамо новий файл під назвою index.js, а всередині зробимо експорт за замовчуванням з {ComponentName}. Корисна річ полягає в тому, що тепер ми можемо перейти до {ComponentName}, і наш імпорт стане гарним і коротким. Ми просто імпортуємо з rrc-diploma і говорим, що нам потрібен компонент {ComponentName}, тож це той тип шляху, який ми дійсно хотіли б спроектувати у світ, тому важливо, щоб наші приклади в кінцевому підсумку відображали шлях, який люди могли б фактично використовувати, коли вони взаємодіють з нашими компонентами, і тепер ми, нарешті, зробили це, тож це шаблон, якого ми будемо дотримуватися. Коли ми створюємо компонент, у нас буде папка, у якій є компонент з іменем, яке відповідає папці компонента, а потім файл індексу, який просто експортує компонент, щоб ми могли добре

імпортувати з наших прикладів. і для всіх, хто вирішить використовувати наші компоненти в реальному світі.

3.13. Підсвічування синтаксису

Додамо підсвічування синтаксису до нашого прикладу фрагментів коду за допомогою highlight.js. Щоб додати форматування та підсвічування синтаксису для наших прикладів коду, повернемося до коду та створимо новий компонент у папці docs (рис. 3.16).

```
1  import React, { Component } from 'react';
2  import PropTypes from 'prop-types';
3  import hljs from 'highlight.js/lib/common';
4  import javascript from 'highlight.js/lib/languages/javascript';
5
6  export default class CodeExample extends Component {
7    componentDidMount() {
8      hljs.registerLanguage('javascript', javascript);
9      hljs.highlightBlock(this.element);
10   }
11   render() {
12     return (
13       <pre style={{ padding: 10, fontSize: 16 }} ref={(ref) => (this.element = ref)}>
14         <code>{this.props.children}</code>
15       </pre>
16     );
17   }
18 }
19
20 CodeExample.propTypes = {
21   children: PropTypes.string.isRequired,
22 };
23
```

Рис. 3.16. Компонент CodeExample

Щоб застосувати підсвічування, ми будемо використовувати highlight.js. Highlight.js – це бібліотека JavaScript, яка забезпечує підсвічування синтаксису для довгого списку мов. У componentDidMount ми ініціюємо highlight.js, а також вказуємо, що ми збираємося працювати з JavaScript, і застосовуємо

виділення до попереднього блоку, використовуючи посилання тут, у рядку 17. Щоб реально застосувати це, відкриємо батьківський компонент і додаємо імпорт. Ми переходимо до прикладу Example, а потім угорі імпортуємо CodeExample, а внизу ми обернемо виклик коду за допомогою CodeExample, оскільки CodeExample очікує, що код буде дочірнім, і, звісно, нам потрібно загорнути код у фігурні дужки. Є ще один останній фрагмент, який нам потрібно додати. Нам потрібно посилатися на highlight.js файл CSS. Highlight.js пропонує десятки різних колірних схем. Ми виберемо темну тему. Відкриємо index.js, який є точкою входу нашої програми, а потім додаємо імпорт. Тепер перейдемо до нашого результату та натиснемо “Show code”. Це набагато легше для читання, і тепер у нас є розфарбовування коду, і все в окремих рядках. Виглядає гарно. Тож тепер будь-хто, хто читає цей код, може підійти і сказати: о, це корисна відправна точка. Я міг би скопіювати та вставити це в свою програму, і поки ми встановили rrc-diploma, це буде працювати. Це дійсно важливий спосіб переконатися, що люди можуть легко використовувати наші компоненти, і це все. Тепер у нас є багаті налаштування автоматичного створення документації, і ми можемо використовувати це як середовище розробки і ніколи не турбуватися про те, що наш код і документи не синхронізуються.

3.14. Атом ProgressBar

Створимо ProgressBar, який відображає прогрес і змінює кольори на основі відсотка завершення, мітку, яка містить необов’язковий маркер поля, і піктограму за допомогою SVG. Створимо папку ProgressBar під джерелом, компонентами, а потім всередині цієї папки ми створимо файл під назвою ProgressBar.js. Цей компонент відображатиме барвисту панель, яка передає прогрес. Пізніше ми використаємо цей атом, щоб допомогти нам створити нашу першу молекулу. Цей компонент концептуально дуже простий (рис. 3.17).

```jsx
import React, { Component } from 'react';
import PropTypes from 'prop-types';

export default class ProgressBar extends Component {
  getColor = () => {
    if (this.props.percent === 100) return 'green';
    return this.props.percent > 50 ? 'lightgreen' : 'red';
  };

  getWidthAsPercentOfTotalWidth = () => {
    return parseInt(this.props.width * (this.props.percent / 100), 10);
  };

  render() {
    const { percent, width, height } = this.props;
    return (
      <div style={{ border: '1px solid lightgray', width }}>
        <div
          style={{
            width: this.getWidthAsPercentOfTotalWidth(),
            height,
            backgroundColor: this.getColor(percent),
          }}
        />
      </div>
    );
  }
}

ProgressBar.propTypes = {
  /** Percent of progress completed */
  percent: PropTypes.number.isRequired,

  /** Bar width */
  width: PropTypes.number.isRequired,

  /** Bar height */
  height: PropTypes.number,
};

ProgressBar.defaultProps = {
  height: 5,
};
```

Рис. 3.17. Атом ProgressBar

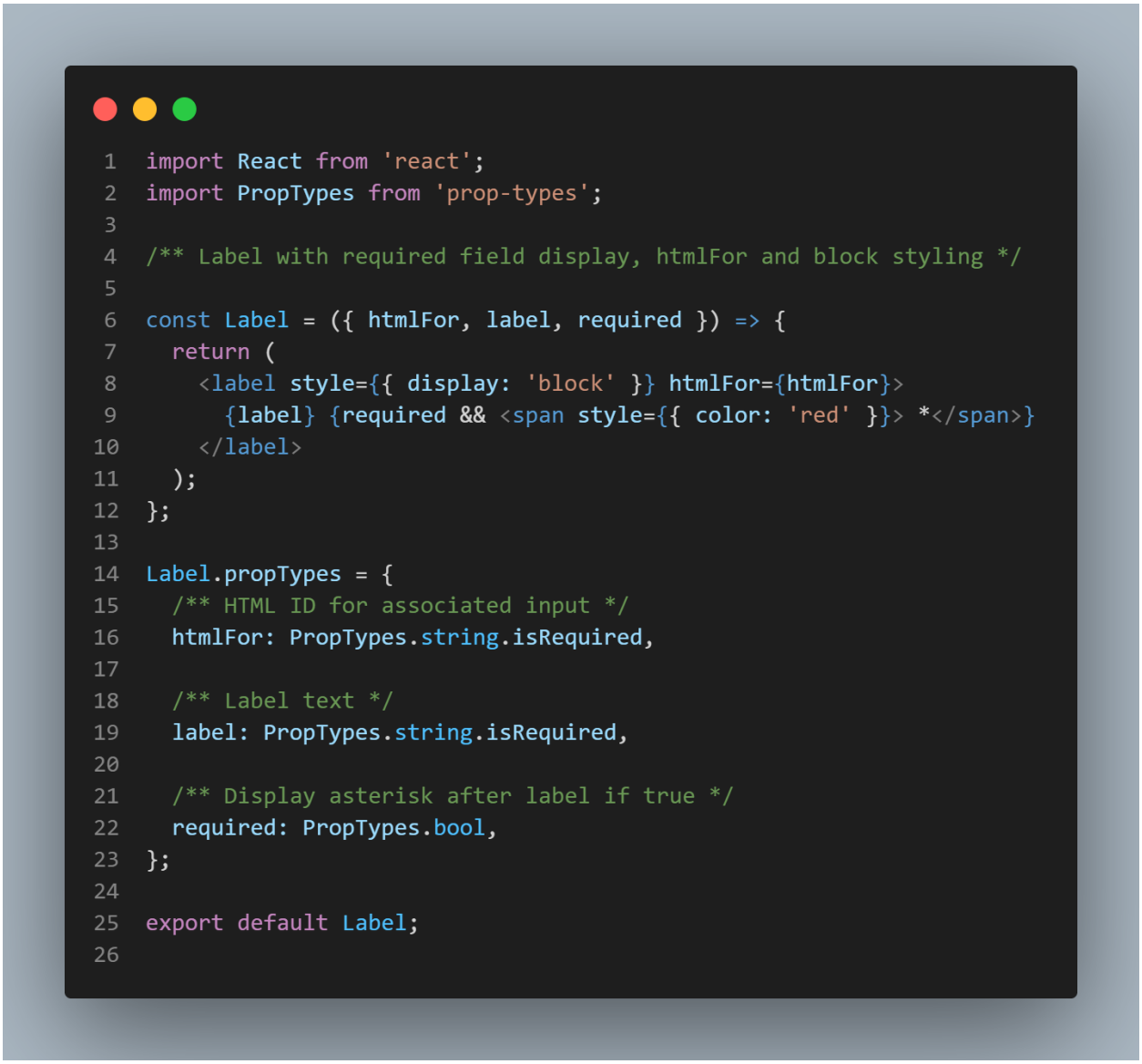
Він прийматиме відсоток від 0 до 100% і відображатиме прогрес у вигляді горизонтальної смуги, і ось як це виглядає. Ми імпортуємо типи react і prop, а потім маємо метод візуалізації, який спирається на два вищенаведені

методи. Перший з них називається getColor. Якщо відсоток передачі становить 100%, то смуга буде зеленою. Якщо він нижче 50%, то смуга буде червоною, а якщо вона десь посередині, то ми зафарбуємо смугу в світло-зелений. Тут досить проста логіка. Цей метод обчислюватиме, наскільки широкою має бути наша смуга на основі переданого відсотка, а наш метод візуалізації тут є div, загорнутим в інший div. Одне, що може привернути нашу увагу, це те, що ми маємо тут. Тепер ми опустимося вниз і експортуємо ProgressBar, а інше, що ми хочемо зробити для узгодженості, — це створити index.js також у папці ProgressBar. Оголосимо деякі реквізити прямо тут: відсотки, ширину та висоту в props. Додамо коментарі над кожним, щоб описати їх, а також оголосимо defaultProps; ми встановимо для висоти 5, якщо це не буде передано. Перейшовши до документів, ми побачимо, що наша таблиця опорних елементів відображається, як очікувалося, і наше значення за замовчуванням було проаналізовано також. Ми також можемо побачити, які реквізити потрібні, але у нас поки що немає прикладів, тому повернемося до редактора та створимо приклад. Ми перейдемо до папки прикладів, і ми створимо папку з назвою ProgressBar, оскільки назва папки в прикладах має відповідати назві компонента, створимо файл з назвою Example10Percent, тому що ми збираємося створити приклад індикатора прогресу, встановленого на 10%. Обов’язково потрібно поставити розширення .js в кінці, тому що компонент сам імпортує ProgressBar, ми додамо коментар, який описує наш приклад, переконайтеся, що ім'я нашого компонента відповідає інам файлу, це просто гарна умова, яку потрібно дотримуватися , а потім ми встановимо відсоток на 10%, а ширину на 150. Якщо ми подивимося у браузері, чудово, ми побачимо наш приклад, наш опис відображається, панель перебігу сам тут відображається з 10% прогресом, і якщо ми натиснемо «Show code», то побачимо код, який потрібен для відтворення цього компонента. Створимо другий приклад Example70Percent.js. Встановлюємо для цього значення 70%, і тепер ми можемо побачити другий приклад, щоб показати людям, як працюють різні налаштування. Створимо Example100Percent.js. Встановимо іншу висоту - 20 пікселів, тому ми

продовжимо і додамо ще одну опору сюди, щоб побачити, як це працює. Ми встановимо відсоток на 100, змінимо це так, щоб воно відповідало інам файлу. Тепер ми маємо третій приклад, 100% прогрес із висотою 20 пікселів, і бачимо інший код, який знадобився для цього. Чудово, тож тепер у нас є три приклади роботи ProgressBar з різними переданими властивостями. Ми створили наш перший компонент для повторного використання. Ми не можемо змінити кольори, які використовуються, або межі смужок, або точки зрізу, коли змінюються кольори, або форму самої смуги. Наприклад, це не може бути коло. Це має бути прямокутна планка, тому це залишає багато можливостей зробити цей компонент більш надійним і придатним для повторного використання в більшому діапазоні ситуацій, але жорсткі компоненти мають багато переваг. Їх легше створювати, розуміти, підтримувати, тестувати, а пізніше легко додати гнучкість.

3.15. Атом Label

Для нашого другого атома створимо абстракцію над рідною label HTML. Щоб зрозуміти, чому уявимо, що ми сідали з командою дизайнерів і вирішили, що всі форми в компанії повинні виглядати однаково. Ми вирішили, що форми мають розташовуватися вертикально, а не горизонтально, щоб мітки розташовувалися над вводом, а не біля нього. Ми вирішили, що обов’язкові поля мають бути позначені червоною зірочкою, а для доступності кожна мітка має бути пов’язана з вводом за допомогою атрибута for. Створюючи власний компонент Label, ми можемо програмно запровадити ці стандарти та заощадити час людей на кожну створену мітку. Оскільки компоненти React починаються з великої літери, ми можемо викликати мітку нашого компонента без конфлікту з компонентом власного HTML-мітки, тому створимо папку під назвою Label під компонентами, а всередині створимо Label.js. Також створимо файл індексу, поки ми будемо зайняті створенням файлів, і помістимо всередину відповідний експорт. Для цього компоненту дуже мало коду, що не дивно (рис. 3.18).



```jsx
import React from 'react';
import PropTypes from 'prop-types';

/** Label with required field display, htmlFor and block styling */

const Label = ({ htmlFor, label, required }) => {
  return (
    <label style={{ display: 'block' }} htmlFor={htmlFor}>
      {label} {required && <span style={{ color: 'red' }}> *</span>}
    </label>
  );
};

Label.propTypes = {
  /** HTML ID for associated input */
  htmlFor: PropTypes.string.isRequired,

  /** Label text */
  label: PropTypes.string.isRequired,

  /** Display asterisk after label if true */
  required: PropTypes.bool,
};

export default Label;

```

Рис. 3.18. Атом Label

Це мітка, яку ми використовуємо у стилях рядків, щоб налаштувати відображення на блокування, а потім ми беремо htmlFor, який передається на props, і оголошуємо його як частину мітки. Ми маємо саму етикетку, яка відображається тут, і якщо вона потрібна, ми додаємо зірочку прямо тут і оформляємо її червоним кольором. Таким чином, це інкапсулює ті основні вимоги. Ми оголосили свої PropTypes тут нижче, і продовжимо і створимо приклад для нашої етикетки. Ми встановимо htmlFor для тестування та мітку для перевірки.

3.16. Атом Icon

SVG є найпопулярнішим способом роботи з піктограмами в наші дні, оскільки вони чудово масштабуються, ефективні з пропускною здатністю і навіть підтримують кілька кольорів, на відміну від альтернатив, як значки шрифтів. Як видно, це обгортка для тегу SVG (рис. 3.19).

```
import React from 'react';

const EyeIcon = () => {
  return (
    <svg width="16" height="16" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 22 22">
      <g transform="matrix(.02146 0 0 .02146 1 1)" fill="#4d4d4d">
        <path d="m466.07 161.53c-205.6 0-382.8 121.2-464.2 296.1-2.5 5.3-2.5 11.5 0 16.9 81.4 174.9 258.6 296.1 464.2
        296.1 205.60 382.8-121.2 464.2-296.1 2.5-5.3 2.5-11.5 0-16.9-81.4-174.9-258.6-296.1-464.2-296.1m0 514.7c-116.1
        0-210.1-94.1-210.1-210.1 0-116.1 94.1-210.1 210.1-210.1 116.1 0 210.1 94.1 210.1 210.1 0 116-94.1 210.1-210.1 210.1" />
        <circle cx="466.08" cy="466.02" r="134.5" />
      </g>
    </svg>
  );
};

export default EyeIcon;
```

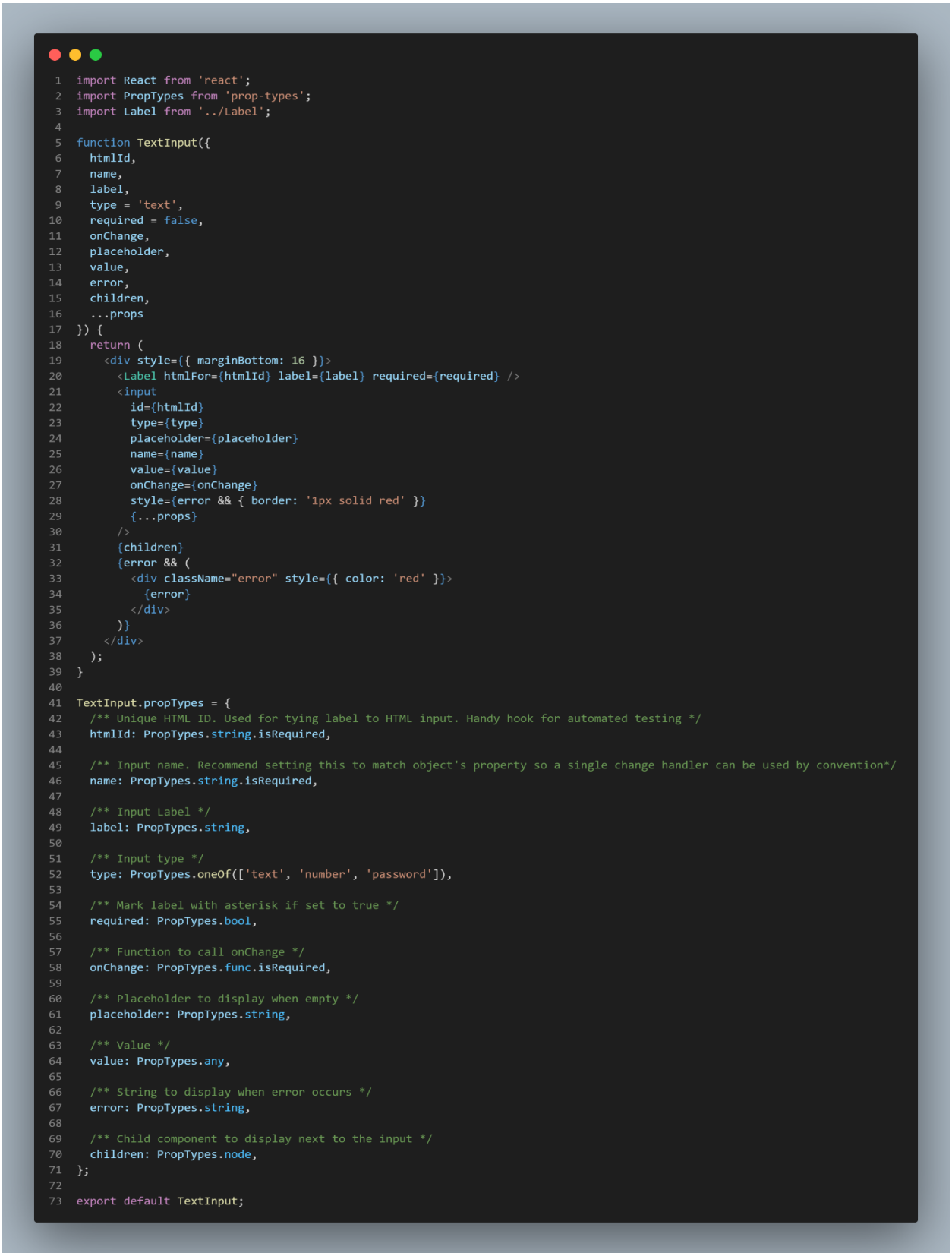
Рис. 3.19. Атом Icon

Тепер додамо короткий приклад, щоб ми могли побачити цю роботу. Зайдемо в розділ examples та створимо нову папку і назвемо її EyeIcon, а потім всередині створимо файл з назвою ExampleEyeIcon.js, і приклад дійсно не може бути набагато простішим. Ми імпортуємо EyeIcon, а потім повертаємо його прямо сюди. Ми бачимо, що піктограма ока відображається тут, як очікувалося.

3.17. Молекула TextInput

Подумаємо про форми. Вони звучать легко, але є багато складнощів і рішень, які потрібно враховувати, коли ми вникнемо в деталі. Макет форми має бути горизонтальним чи вертикальним? Іншими словами, мітка має бути ліворуч від входу чи над ним? Чи слід показувати помилки вгорі, біля поля чи в обох місцях? Чи слід показувати помилки під час розмиття чи надсилати? Чи

потрібно позначати обов’язкові поля, і якщо так, то як? Чи слід пов’язувати мітку з вводом за допомогою htmlFor для цілей доступності? Чи має бути відступ під вводом, і якщо так, то скільки? Усі ці рішення уповільнюють роботу розробників і, що ще гірше, можуть призвести до не постійної взаємодії з користувачем, коли різні розробники впроваджують різні частини додатка. Завдяки компонентам багаторазового використання ми маємо можливість втілити всі ці рішення та багато іншого прямо в компонент. Результат цієї роботи? Узгодженість зростає, оскільки вона забезпечується програмно, а втома розробників від прийняття рішень зменшується. Добре розроблені компоненти інкапсулюють рішення. У наступних двох компонентах ми використаємо Label, ProgressBar та EyeIcon атоми, які ми вже створили. Створимо TextInput, доповнений деякими думками щодо розміщення пов’язаної етикетки, прив’язування її до входу, відображення необхідного маркера поля та відображення помилок перевірки. Знову зробимо як завжди. Ми перейдемо до папки компонентів, створимо TextInput, створимо всередині файл з таким же ім’ям, створимо наш index.js, додамо експорт, і ось компонент TextInput (рис. 3.20).



```jsx
import React from 'react';
import PropTypes from 'prop-types';
import Label from '../Label';

function TextInput({
  htmlId,
  name,
  label,
  type = 'text',
  required = false,
  onChange,
  placeholder,
  value,
  error,
  children,
  ...props
}) {
  return (
    <div style={{ marginBottom: 16 }}>
      <Label htmlFor={htmlId} label={label} required={required} />
      <input
        id={htmlId}
        type={type}
        placeholder={placeholder}
        name={name}
        value={value}
        onChange={onChange}
        style={error && { border: '1px solid red' }}
        {...props}
      />
      {children}
      {error && (
        <div className="error" style={{ color: 'red' }}>
          {error}
        </div>
      )}
    </div>
  );
}

TextInput.propTypes = {
  /** Unique HTML ID. Used for tying label to HTML input. Handy hook for automated testing */
  htmlId: PropTypes.string.isRequired,

  /** Input name. Recommend setting this to match object's property so a single change handler can be used by convention*/
  name: PropTypes.string.isRequired,

  /** Input Label */
  label: PropTypes.string,

  /** Input type */
  type: PropTypes.oneOf(['text', 'number', 'password']),

  /** Mark label with asterisk if set to true */
  required: PropTypes.bool,

  /** Function to call onChange */
  onChange: PropTypes.func.isRequired,

  /** Placeholder to display when empty */
  placeholder: PropTypes.string,

  /** Value */
  value: PropTypes.any,

  /** String to display when error occurs */
  error: PropTypes.string,

  /** Child component to display next to the input */
  children: PropTypes.node,
};

export default TextInput;
```

Рис. 3.20. Молекула TextInput

Насправді не так багато коду. Наші propType знаходяться тут на 25, так що насправді код, на якому потрібно зосередитися, знаходиться саме тут. Ми використовуємо компонент Label, який ми створили раніше, і деструктуруємо

ряд реквізитів тут, угорі. Спочатку ми створюємо функціональний компонент без стану, оскільки цей компонент не потребує локального стану. Далі деструктуруємо props, щоб виклики, наведені нижче, були короткими, і використовуємо тут розповсюдження об'єктів, щоб захопити будь-які інші реквізити, передані в цей компонент, і застосувати їх до входу тут, у рядку 18. Ми використовуємо параметри за замовчуванням для встановлення значення за замовчуванням для типу та обов'язкового. Цей підхід відрізняється від компонентів на основі класів, де ми оголошуємо параметри за замовчуванням під оголошенням класу або через статичні властивості. Сама розмітка досить проста. Ми загортаємо компонент у div, щоб ми могли забезпечити постійне поле під кожним введенням. Ми використовуємо вбудовані стилі React для встановлення поля, і встановлюємо межу на вході, коли передається помилка. Важлива примітка: для простоти ми не централізуємо жоден зі своїх стилів, але завжди можемо перемістити свої оголошення стилів в окремий файл JavaScript, якщо потрібно. Вбудовані стилі React – це просто JavaScript, тому ми, безумовно, можемо витягти їх в окремий файл і централізувати будь-які загальні стилі, які використовуються на кількох компонентах. Якщо помилка перевірки була передана на props, ми відображаємо її червоним кольором під вводом. Решта цього файлу – це лише оголошення типу prop. Нам потрібен ідентифікатор HTML, щоб у нас був унікальний ідентифікатор для зв'язування мітки та введення разом з метою забезпечення доступності. Іншим примітним пунктом є те, що ми приймаємо деякі конкретні типи введення, тому цей вхід можна використовувати як власне число або, можливо, також як пароль. В папці прикладів створимо наш перший приклад, щоб ми могли побачити це в дії. Створимо TextInput, а потім всередині ми створимо ExampleOptional.js. Ми можемо бачити, що ми просто оголошуємо порожній обробник onChange, тому що нам насправді нічого не потрібно тут. Ми просто хочу, щоб він відображався, також встановлюємо інші обов'язкові поля з основною інформацією. Клацнемо на TextInput, і ми можемо побачити код, як очікувалося, і опис, що відображається вгорі. Ми створимо другий приклад,

який ми назвемо ExampleError.js, тому ми можемо показати, що виникає помилка, і це дуже схоже на першу, але ми просто жорстко кодуємо помилку і встановлюємо його як обов’язковий, а потім просто передаємо повідомлення про помилку для відображення.

3.18. Молекула PasswordInput

Створимо нашу другу молекулу, компонент PasswordInput. Для нашої другої молекули помістимо наші атоми, а також молекулу TextInput, яку ми створили для використання. Ми збираємося створити компонент для введення пароля, тож повернемося до каталогу компонентів і створимо PasswordInput, а також файл всередині, а також папку індексу. нам потрібно перейменувати це в PasswordInput.js (рис. 3.21), і добавимо експорт до index.js.

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import ProgressBar from '../ProgressBar';
import EyeIcon from '../EyeIcon';
import TextInput from '../TextInput';

/** Password input integrated label, quality tips and show password toggled */

export default class PasswordInput extends Component {
  constructor(props) {
    super(props);
    this.state = { showPassword: false };
  }

  toggleShowPassword = (event) => {
    this.setState((prevState) => ({ showPassword: !prevState.showPassword }));
    if (event) event.preventDefault();
  };

  render() {
    const {
      htmlId,
      value,
      onChange,
      label,
      error,
      placeholder,
      maxLength,
      showVisibilityToggle,
      quality,
      ...props
    } = this.props;
    const { showPassword } = this.state;

    return (
      <TextInput
        htmlId={htmlId}
        value={value}
        onChange={onChange}
        label={label}
        type={showPassword ? 'text' : 'password'}
        placeholder={placeholder}
        maxLength={maxLength}
        error={error}
        required
        {...props}>
        {showVisibilityToggle && (
          <a href="#" onClick={this.toggleShowPassword} style={{ marginLeft: 5 }}>
            <EyeIcon />
          </a>
        )}
        {value.length > 0 && quality && <ProgressBar percent={quality} width={130} />}
      </TextInput>
    );
  }
}

PasswordInput.propTypes = {
  /** Unique HTML ID. Used for tying label to HTML input. Handy hook for automated testing */
  htmlId: PropTypes.string.isRequired,

  /** Input name. Recommend setting this to match object's property so a single change handler can be used by convention*/
  name: PropTypes.string.isRequired,

  /** Value */
  value: PropTypes.any,

  /** Input Label */
  label: PropTypes.string,

  /** Function to call onChange */
  onChange: PropTypes.func.isRequired,

  /** Max password length accepted*/
  maxLength: PropTypes.number,

  /** Placeholder to display when empty */
  placeholder: PropTypes.string,

  /** Set to true to show the toggle for displaying the currently entered password */
  showVisibilityToggle: PropTypes.bool,

  /** Display password quality visually via ProgressBarm, accepts a number between 0 and 100 */
  quality: PropTypes.number,

  /** String to display when error occurs */
  error: PropTypes.string,

  /** Child component to display next to the input */
  children: PropTypes.node,
};

PasswordInput.defaultProps = {
  maxLength: 50,
  showVisibilityToggle: false,
  label: 'Password',
};
```

Рис. 3.21. Молекула PasswordInput

Ми імпортуємо інші компоненти, які ми створили, ProgressBar, EyeIcon і TextInput будуть використовуватися для введення пароля. Це компонент стилю класу, тому що ми маємо стан. Ми збираємося відстежувати, чи є пароль

видимим, тому що цей компонент дозволить нам перемикати видимість пароля, натиснувши значок EyeIcon, і тому у нас є невеликий метод, який буде перемикати showPassword кожен раз, коли натискаєте цю піктограму. Ми використовуємо стиль зворотного виклику стану set, щоб можна було перемикати showPassword, посилаючись на попередній стан. Сама візуалізація в кінцевому підсумку використовує наявні компоненти, які ми створили, тому ми деструктуруємо тут, угорі, лише щоб наші виклики були короткими внизу, і ми викликаємо TextInput, а потім використовуємо дочірню властивість, яка надається на введенні тексту. Ми хочемо розмістити значок EyeIcon поруч із TextInput, і ми хочемо, щоб він знаходився всередині div. Введення тексту, якщо ми його відкрили, використовує div для інкапсуляції мітки та рідного введення HTML, тому нам потрібно використовувати властивість дочірнього елемента тут, щоб наш значок EyeIcon знаходився в цьому div, і ми показуємо лише це, якщо showVisibilityToggle відповідає дійсності. Ми використовуємо логічний && тут, щоб виконати це, і перемикаємо тип введення залежно від того, чи встановлено showPassword. Якщо для showPassword встановлено значення true, ми робимо його як “text”. В іншому випадку ми робимо “password”. Ми дозволили компоненту підтримувати як стиль введення пароля, так і введення тексту. Потім під нашим компонентом ми використовуємо наш ProgressBar для відображення якості нашого пароля, і встановлюємо для нього статичну ширину 130 пікселів, але ми відображаємо цю якість, лише якщо є значення для пароль. Якщо хтось ще не ввів пароль, то ми не будемо показувати якість, а якість - це відсоток, тому ми використовуємо логічне і кажемо, якщо у нас немає якості, то навіть немає показувати індикатор прогресу, оскільки якість передається реквізитам. Потім, щоб отримати трохи більше інформації, ми можемо зайти сюди і ознайомитися з типами реквізитів, тому що ми додали коментарі, щоб були зрозумілі наміри всього цього. Ми також оголосили декілька параметрів defaultProps, maxLength 50, showVisibilityToggle за замовчуванням буде false, тому ми повинні явно вирішити увімкнути цю функцію, а потім встановили мітку за замовчуванням

на “Password”. Створимо приклад, щоб ми могли побачити, як ця річ працює. Ми перейдемо до прикладів, створимо нашу папку, а потім створимо новий файл і назвемо його ExampleAllFeatures.js. Звичайно, ми імпортуємо PasswordInput, а потім нам потрібен стан, щоб відстежувати пароль, тому що PasswordInput потребує обробника onChange, щоб ми могли показувати поведінку панелі якості, яка змінюється під час введення, і ми створили алгоритм, який дуже спрощено обчислює якість пароля. Кожне натискання клавіш коштує 10%, тож коли ми досягнемо 10 клавіш, ми досягнете 100%. Тут ми викликаю PasswordInput, посилаюся на обробник onChange, і щоразу, коли відбувається подія, ми просто використовуємо функцію стрілки тут у рядку, щоб все було просто, і встановлюємо пароль для e.target.value. Ми показуємо VisibilityToggle і отримуємо якість прямо тут, і передаємо його в PasswordInput, але важливою частиною є те, що нам потрібен обробник onChange, щоб ми могли бачити цей компонент у дії. Ми бачимо, як він відображається, ми можемо вводити текст, і з кожним натисканням клавіші ми бачимо, що індикатор прогресу продовжується, поки ми не досягнемо 100%, коли ми натиснемо цей десятий символ. Ми також можем натиснути значок ока, і він перемикає видимість нашого PasswordInput.

3.19. Організм RegistrationForm

Створимо організм RegistrationForm, який використовує наші молекули. Щоб створити приклад організму, ми збираємося помістити все, що ми створили до цього часу, щоб використовувати для створення реєстраційної форми. Для цього не знадобиться багато коду, оскільки ми будемо використовувати так багато насичених компонентів, які ми вже зібрали. Спочатку виконаємо той самий підхід, який ми робили досі. Ми створимо папку RegistrationForm і створимо RegistrationForm.js всередині. Також створимо index.js і експортуємо за замовчуванням звідти. Тепер створимо форму RegistrationForm.js (рис. 3.22).

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import TextInput from '../TextInput';
import PasswordInput from '../PasswordInput';

export default class RegistrationForm extends Component {
  constructor(props) {
    super(props);
    this.state = {
      user: {
        email: '',
        password: '',
      },
      errors: {},
      submitted: false,
    };
  }

  onChange = (event) => {
    const user = this.state.user;
    user[event.target.name] = event.target.value;
    this.setState({ user });
  };

  passwordQuality(password) {
    if (!password) return null;
    if (password.length >= this.props.minPasswordLength) return 100;
    const percentOfMinLength = parseInt((password.length / this.props.minPasswordLength) * 100, 10);
    return percentOfMinLength;
  }

  validate({ email, password }) {
    const errors = {};
    const { minPasswordLength } = this.props;

    if (!email) errors.email = 'Email required.';
    if (password.length < minPasswordLength)
      errors.password = `Password must be at least ${minPasswordLength} characters.`;

    this.setState({ errors });
    const formIsValid = Object.getOwnPropertyNames(errors).length === 0;
    return formIsValid;
  }

  onSubmit() {
    console.log('####: this.state', this.state);
    const { user } = this.state;
    console.log('####: user', user);
    const formIsValid = this.validate(user);
    if (formIsValid) {
      this.props.onSubmit(user);
      this.setState({ submitted: true });
    }
  }

  render() {
    const { errors, submitted } = this.state;
    const { email, password } = this.state.user;

    return submitted ? (
      <h2>{this.props.confirmationMessage}</h2>
    ) : (
      <div>
        <TextInput
          htmlId="registration-form-email"
          name="email"
          onChange={this.onChange}
          label="Email"
          value={email}
          error={errors.email}
          required
        />

        <PasswordInput
          htmlId="registration-form-password"
          name="password"
          value={password}
          onChange={this.onChange}
          quality={this.passwordQuality(password)}
          showVisibilityToggle
          maxLength={50}
          error={errors.password}
        />

        <input type="submit" value="Register" onClick={this.onSubmit.bind(this)} />
      </div>
    );
  }
}

RegistrationForm.propTypes = {
  /** Message displayed upon successful submission */
  confirmationMessage: PropTypes.string,

  /** Called when form is submitted */
  onSubmit: PropTypes.func.isRequired,

  /** Minimum password length */
  minPasswordLength: PropTypes.number,
};

RegistrationForm.defaultProps = {
  confirmationMessage: 'Thank for registering!',
  minPasswordLength: 8,
};

```

Рис. 3.22. Організм RegistrationForm

Сама реєстраційна форма містить близько 100 рядків коду, включаючи наші перевірки PropType, тому розглянемо це. Ми імпортуємо молекули, які ми

створили у верхній частині, і цей організм обробляє свій власний стан форми, включаючи електронну пошту, пароль, будь-які помилки та логічне значення, яке відстежує, чи була форма надіслана, використовуючи єдиний обробник змін для обох полів, і він використовує властивість name для встановлення пов'язаного значення в стані. Потім ми переходимо до функції, де обчислюється якість пароля. Ця функція повертає число від 0 до 100, що відображає якість пароля. Тепер ми можемо створити для цього дійсно складний алгоритм, але для демонстрації ми залишили його простим і зосередилися виключно на дотриманні вимог щодо мінімальної довжини пароля, переданої в props, і якщо він відповідає цим вимогам, ми повертаємо 100; в іншому випадку ми повертаємо відсоток, який представляє частину мінімальної довжини пароля, яка була введена на даний момент. Тепер ми можемо задатися питанням, чому код обчислення пароля сидить тут. Чи не могли б ми замість цього вбудувати цю логіку в компонент PasswordInput? Так, ми могли б, але були б певні наслідки. По-перше, це зробило б введення пароля менш гнучким. Ми мали б дуже конкретні думки, а по-друге, батьківські компоненти, такі як ця реєстраційна форма, також повинні знати, чи пароль пройшов перевірку, а це означає, що батьківській формі потрібно буде отримати інформацію від дочірнього PasswordInput, щоб визначити, чи пароль відповідає мінімальним вимогам якості. Функція validate встановлює помилки в стані, якщо поля електронної пошти або пароля не відповідають вимогам, і повертає логічне значення, щоб уявити, чи є форма дійсною. Функція onSubmit обробляє відправку форми. IT викликає перевірку вище, яка обробляє подачу форми та відповідно встановлює об'єкт помилки. Якщо форма дійсна, ми продовжуємо і викликаємо функцію onSubmit, яка була передана в props, а потім ми викликаємо setState, щоб помітити, що форму було надіслано, задавши значення true. Переходячи до функції візуалізації, це досить просто, тому що ми викликаємо існуючі компоненти, які ми вже створили. Якщо форма надіслана, вона відображає повідомлення підтвердження, інакше ми відображаємо нашу форму, яка включає наші компоненти TextInput і PasswordInput. Потім внизу

кнопка «Надіслати», яка викликає нашу функцію onSubmit, тому тут немає нічого дивного, і хоча це дуже складний компонент, він приймає лише три опори, необов'язкове повідомлення підтвердження для відображення onSubmit, функцію для виклику під час відправки та мінімальний довжина пароля. Організми абстрагують компоненти, що знаходяться під ними, тому ми мали вибір вибрати, які реквізити виставити на цьому компоненті. Щоб все було просто, ми показуємо лише основні елементи, які нам потрібні для роботи. Наприклад, ми не відкриваємо жодних перемикачів базових функцій при введенні пароля. Звичайно, ми можемо розширити цю форму реєстрації, щоб підтримувати налаштування базових компонентів різними способами, якщо хочемо, але, як правило, для організмів думки, які вони запекли, є особливістю, тому намагаємося строго ставитися до конфігурацій, які підтримуєте. Це допоможе зберегти API організмів простим і забезпечити узгодженість, і, як ми вже робили раніше, створимо приклад, щоб ми могли побачити це в дії. RegistrationForm - це ім'я папки, а потім всередині ми створимо ExampleRegistrationForm.js, а для прикладу ми викликаємо форму реєстрації і передаємо їй опору для onSubmit. Ми бачимо, що наш код відображається. Ми мали би мати можливість вводити електронну пошту та пароль, і коли ми вводим цей пароль, ми повинні бачити, як він росте. Коли ми натискаємо Registration - бачимо дякую за реєстрацію. Ми бачимо, що він записав на консоль, і що ми отримали нашу електронну пошту та пароль, які були передані цій функції onSubmit. RegistrationForm працює так само, як ми цього очікували, і останнє зауваження, ми могли б назвати цей RegistrationFormContainer замість цього, оскільки, якщо ми подивимося тут на компонент, ми могли б створити лише форму реєстрації, яка містила цю розмітку, фактично нудну компонент, дехто назвав би його компонентом презентації, і тоді могли б назвати весь цей компонент з нашим управлінням тут, RegistrationFormContainer, але немає особливих причин для цього, оскільки в компоненті RegistrationForm було б так мало нової розмітки , але при цьому ми можемо розглянути можливість використання контейнера суфіксів для

компонентів, які керують станом. Це може допомогти людям швидше зрозуміти, що робить компонент, і все.

3.20. Модульний тест

Почнемо зі створення автоматизованого модульного тесту. Create-react-app вже постачається з налаштованим тестуванням Jest, але є три інші бібліотеки, пов'язані з тестуванням, які ми будемо використовувати; react-test-renderer, Enzyme та react-addons-test-utils. React-test-renderer корисний для створення тестів знімків у Jest. Enzyme — це зручний спосіб перевірити взаємодію компонентів за допомогою синтаксису jQuery для маніпуляцій DOM і запитів, а react-addons-test-utils — це набір основних утиліт для тестування, наданих Facebook для React. Ми не будемо використовувати це безпосередньо, але Enzyme використовує його за лаштунками, оскільки Enzyme, в основному, є дружньою обгорткою для react-addons-test-utils. У наступних кількох кліпах ми використаємо кожен із них разом із Jest. Для нашого першого модульного тесту напишемо тест для функції всередині ProgressBar (рис. 3.23).

```
import React from 'react';
import Adapter from '@wojtekmaj/enzyme-adapter-react-17';
import { shallow, configure } from 'enzyme';
import ProgressBar from './ProgressBar';

configure({ adapter: new Adapter() });

describe('ProgressBar', () => {
  test('getWidthAsPercentOfTotalWidth should return 250 with total width of 500 and percent of 50', () => {
    const wrapper = shallow(<ProgressBar percent={50} width={500} />);
    const width = wrapper.instance().getWidthAsPercentOfTotalWidth();
    expect(width).toEqual(250);
  });
});

```

Рис. 3.23. Тест для функції getWidthAsPercentOfTotalWidth

Ця функція має деяку логіку для надання ширини як цілого числа з урахуванням загальної ширини та відсотка. Ця логіка використовується для

визначення ширини ProgressBar для даного поточного налаштування відсотка, тому є два способи, якими ми можемо перевірити цю функцію. Перший підхід є найпростішим, і він передбачає використання Enzyme, тому перейдемо до нашого файлу специфікації, і знову ж таки, Jest достатньо розумний, щоб знайти цей файл автоматично, оскільки він закінчується в spec.js, ми також могли б добавити в інам файлу .test.js - це також працювало б. Для початку додамо необхідний імпорт. Звісно, ми будемо працювати з React. Ми збираємося використовувати не глибокий рендерер Enzyme для роботи з цим тестом, і тоді, звісно, нам знадобиться посилання на систему, що тестується, якою в даному випадку буде ProgressBar. Під час написання тестів у Jest ми можемо додатково створити блок опису, а блок опису — це лише спосіб згрупувати наші тести разом, щоб упорядкувати їх. Абсолютно необов'язковий. Деякі люди вважають це корисним. Всередині напишемо наш перший тест, і назва тесту буде досить довгою. Ми збираємося перевірити, що getWidthAsPercentOfTotalWidth має повертати 250 із загальною шириною 500 і відсотком 50, тому що логіка полягає в тому, що ми хочемо отримати назад ширину для заданого відсотка та загальної ширини.

3.21. Тест знімків

Напишемо кілька структурних тестів для компонента PasswordInput за допомогою знімків Jest. Для початку створимо файл .spec.js в папці PasswordInput і назвемо його PasswordInput.spec.js.

```
test('hides password quality by default', () => {
  const tree = renderer
    .create(<PasswordInput htmlId="test" name="test" onChange={() => {}} value="Uisi38#8iad" />)
    .toJSON();
  expect(tree).toMatchSnapshot();
});

```

Рис. 3.24. Тест знімків для молекули PasswordInput

Для початку нам потрібні три імпорти. Нам потрібно імпортувати react, нам потрібно імпортувати react-test-renderer, який буде використовуватися для підтримки наших тестів знімків, і, нарешті, PasswordInput, який є тестованою системою. Ми збираємося перевірити, чи введений пароль приховує панель якості пароля за замовчуванням, тому що ми закодували так, що нам потрібно явно увімкнути панель якості пароля, щоб вона відображалася, і ми можемо побачити, що ми не увімкнули якість пароля тут. Ми лише вказуємо необхідні реквізити. Встановлюємо для пароля значення, панель якості пароля взагалі не відображається, поки ми не вводимо пароль. create створює дерево, яке ми можемо серіалізувати в JSON, тому ми беремо те, який засіб візуалізації. create products, ми серіалізуємо його в JSON, а потім стверджуємо, що це дерево відповідає знімку. Тепер, звичайно, коли ми вперше запускаємо цей тест, моментальний знімок не існуватиме, тому тест напевно пройде цей перший раз, але виконаємо npm t, ми побачимо, що зараз у нас є три тести замість двох, і він говорить, що один знімок був написаний в одному тестовому наборі. Ми бачимо, що тут створена нова папка під назвою __snapshots__. У цьому файлі є наш знімок цього конкретного компонента з цими входами, і ми бачимо, що він зберігає те, що ми назвали нашим тестом, а потім він зберігає вихід нашого тесту (рис 3.25).

Рис. 3.25. Знімок молекули PasswordInput

Тож ми передали ці реквізити, і це був результат нашого компонента з урахуванням його вхідних даних. Ось що цікаво, наприклад, ми маємо тут вбудовані стилі. Якби вбудовані стилі змінилися в будь-який момент, тоді наш тест знімків порушився б, тому що ми б сказали: «Привіт, наші стилі змінилися, і нам доведеться вирішити, чи це була навмисна зміна, чи це була випадкова зміна. Таким чином, ми можемо використовувати тестування моментальних знімків, щоб робити знімки різних перестановок наших реквізитів, але слід пам’ятати про одну очевидну примху з тестами знімків. Вони завжди проходять

під час першого запуску, оскільки вони просто роблять знімок виводу компонента, тому корисно переглянути вихід і переконатися, що він відповідає очікуванням, і останнє: ми також можемо використовувати моментальний знімок тестує разом із Enzyme, щоб перевірити конкретні частини виходу нашого компонента. Ми можемо використовувати Enzyme, щоб написати селектори для певних частин, створити взаємодії, а потім зробити знімок конкретних частин цієї DOM до і після.

3.22. Тест взаємодії

Для цього ми будемо використовувати Enzyme. Enzyme пропонує два способи перевірити компонент: не глибокий і монтувати. Shallow відображає компонент, який ми тестуємо ізольовано без дітей. Це робить його швидким і допомагає ізолювати наш тест, тому почніть з неглибокого, якщо можемо, але якщо нам потрібно перевірити методи життєвого циклу або діти, замість цього використовуємо mount. Підключення трохи повільніше, але воно відображає всіх дітей і підтримує тестування методів життєвого циклу. Отже, створимо автоматизований тест взаємодії за допомогою Enzyme (рис. 3.26).

```
import React from 'react';
import renderer from 'react-test-renderer';
import PasswordInput from './PasswordInput';
import Adapter from '@wojtekmaj/enzyme-adapter-react-17';
import { shallow, configure } from 'enzyme';

configure({ adapter: new Adapter() });

test('toggles input type when show/hide password clicked', () => {
  const wrapper = shallow(
    <PasswordInput htmlId="test" name="test" onChange={() => {}} value="" showVisibilityToggle />,
  );

  // Password input should have a type of password initially
  expect(wrapper.find({ type: 'password' })).toHaveLength(1);
  expect(wrapper.find({ type: 'text' })).toHaveLength(0);

  wrapper.find('a').simulate('click');

  // Password input should have a type of text after clicking toggle
  expect(wrapper.find({ type: 'password' })).toHaveLength(0);
  expect(wrapper.find({ type: 'text' })).toHaveLength(1);
});
```

Рис. 3.26. Тест взаємодії молекули PasswordInput

Використовувати Enzyme разом із Jest гарна річ, оскільки це полегшує тестування поведінки та дозволяє нам тестувати один компонент ізольовано за допомогою функції, яка називається не глибоким рендерингом. Щоб використовувати Enzyme, імпортуємо його у верхній частині spec.js. Тепер просто потрібно використовувати Shallow для нашого першого тесту. Shallow відтворює один компонент. Тому його називають не глибоким. Він не відображає жодного дочірнього компонента. Це корисно, оскільки робить тест швидшим і часто полегшує його налаштування, оскільки дозволяє проводити модульне тестування окремого компонента. Ми також можемо використовувати функцію монтування Enzyme, яка повністю монтує компонент і всі його дочірні елементи, але нам це не потрібно для тесту, який ми тут пишемо. Enzyme особливо корисний для тестування поведінки. Передбачається, що наш PasswordInput перемикає видимість пароля, коли ми клацаємо по значку ока, змінюючи тип введення за кадром, тому напишемо тест для цього. Таким чином, наш тест буде полягати в тому, що він перемикає тип введення, коли натискається показати/приховати пароль. Для

початку викличемо shallow і передаємо йому наш компонент, який ми хотіли б відобразити з реквізитами, які ми хотіли б передати. Зверніть увагу, що ми встановили showVisibilityToggle, щоб ми знали, що він буде там, і ми звичайно, потрібно, щоб він був присутній у нашому тесті, тому що весь цей тест стосується тестування, щоб ми могли натиснути на нього та побачити, як він перемикає тип введення, але неглибоке повернення є обгорткою цього компонента, щоб ми могли робити твердження щодо цієї обгортки. Ми можемо запитувати цю обгортку так само, як ми сидимо в реальному браузері, тому що за лаштунками Enzyme використовує JSDOM, який є віртуальним DOM в пам'яті, тому ми хочемо перевірити, що введення пароля має мати тип пароля спочатку, і для цього ми скажемо, що очікуємо обгортку, і ми збираємося використати find — ми шукаємо введення з типом пароля. Ми знаємо, що тут лише один, і ми очікуємо, що він матиме одну довжину. Тепер ми також можемо скопіювати це, тому що є ще один тест, який ми можемо виконати тут. Ми можемо стверджувати, що введення типу тексту повинно мати довжину 0, оскільки, оскільки в цьому компоненті є лише 1 вхід, ми просто тестуємо, щоб переконатися, що спочатку єдиним введенням є пароль і що там немає введення тексту.

3.23. Публікація документації через GitHub Pages

Це займе лише мить. Завершимо наші документи, подивимося, як легко створити автоматичне розгортання наших документів на сторінках GitHub. Щоб опублікувати документи на GitHub Pages, використаємо пакет npm, який називається gh-pages. Gh-pages достатньо розумний, щоб читати наш package.json і завантажити наш проект для розміщення через GitHub Pages у нашому репозиторії GitHub, тож перейдемо до нашого package.json до розділу скриптів. Ми викличемо це, додавши новий скрипт npm. Create-react-app записує створену програму в каталог збірки, тому ми просто кажемо gh-pages розгортати вбудовані документи з каталогу збірки. Звичайно, має сенс створити

програму перед її розгортанням. Наразі сценарій для створення наших документів для виробництва називається просто build:docs, для створення документів. Сценарій predeploy:docs, щоб переконатися, що наші документи створені, перш ніж ми їх розгортаємо. Таким чином ми можемо просто запустити цей один сценарій, і наші документи будуть створені, а потім розгорнуті через gh-сторінки. Оскільки ми створюємо для сторінок GitHub, нам потрібно встановити налаштування домашньої сторінки в package.json, і це повідомить create-react-app, що ми розміщуємо в підкаталогі.

3.24. Налаштування README

Звичайно, README GitHub також має значення. Це перше, що, швидше за все, побачать наші користувачі, тому переконуємося, що він містить резюме проекту, інструкції зі встановлення та, звичайно, посилання на щойно створені документи. Налаштуємо базове читання GitHub, яке містить найнеобхідніше. Npm відображатиме Readme GitHub, коли ми переглядаємо пакет на npmjs.com також, тому важливо налаштувати readme перед публікацією пакета. Тепер ми бачимо, що наш README заповнений.

3.25. Підготовка package.json для публікації

Перш ніж опублікувати наш пакет, нам також потрібно переглянути package.json, щоб переконатися, що він містить всю необхідну інформацію для публікації. Package.json — це маніфест для пакетів npm. Він містить різноманітну інформацію про наш пакунок, включаючи список усіх пакетів npm, які використовує наш проект. Важливим рішенням тут є те, як оголошувати свої залежності, оскільки ми оголошуємо залежності в package.json впливає на наших користувачів. Існує чотири різних типи залежностей, які ми можемо оголосити в package.json. Найбільш очевидна залежність — залежність від болю. Це означає, що наша залежність завжди

потрібна, тому вона встановлюється щоразу, коли користувачі запускають npm install, оскільки вона завжди потрібна для запуску нашого проекту. ми встановлюємо цей тип залежності, запустивши npm install -S. devDependencies потрібні лише для розробки. Ці залежності встановлюються, якщо ми запускаємо npm install у каталозі, де знаходиться package.json існує. Ця відмінність корисна, оскільки означає, що під час публікації пакунка будь-які споживачі не встановлюватимуть залежності, які ми перерахували в devDependencies, оскільки користувач не запускатиме команду npm install у каталозі, де знаходиться наш пакунок npm після встановлення. ми встановлюємо цей тип залежності, запустивши npm install -D. PeerDependencies – це спосіб сказати, що ми очікуємо, що залежність буде встановлена споживачем, тому, якщо ми оголошуємо залежність від однорангового користувача, ми стверджуємо, що користувач повинен вручну встановити peerDependency, а щоб оголосити peerDependencies, ми повинні вручну оголосити її у своєму package.json. Нарешті, встановлюються optionalDependencies, якщо вони доступні на платформі. Це корисно, якщо є пакет, який ми хочемо встановити, якщо він сумісний із платформою. Наприклад, FSEvents недоступний у Windows, тому його часто вказують як додаткову залежність. ми можемо встановити додаткову залежність за допомогою npm install -O, тому потрібно розміщувати залежності лише для розробки в розділі devDependencies нашого package.json. Таким чином, коли люди встановлюють наші компоненти, їм не потрібно встановлювати всі наші інструменти розробки. Npm встановить лише ті пакунки, які ми перерахуємо під залежностями. Підготовка package.json для Publish має ряд інших полів, які нам потрібно додати та налаштувати, тож перейдемо до package.json та вкажемо деякі додаткові параметри для підготовки до публікації. Перш ніж публікувати нашу бібліотеку компонентів багаторазового використання, нам потрібно додати кілька полів до package.json. Нам потрібно вказати вузлу точку входу для нашого пакета. Ми робимо це, оголошуючи основний запис. Ми налаштували наш пакет для

запису в папку lib, тому цей шлях відображає це рішення. Іменування цієї папки lib є популярною конвенцією. Оскільки ми пишемо нашу збірку до папки lib, цей параметр підтримує імпорт компонентів користувача безпосередньо з папки lib за допомогою цього синтаксису. Ми збираємося оголосити мінімальну версію вузла, яку підтримує наш пакет. Ми будемо підтримувати вузол 4.0 і вище. Теоретично ми могли б підтримати і далі. Оновлення безкоштовні. Тоді настав час надати дані про авторів. Далі ми заявляємо, що папка lib — це єдине, що має бути опубліковано в нашому пакеті npm. Інші важливі файли, наприклад package.json і readme.md буде опубліковано автоматично. Це перевага файлового підходу. Ми оголосили кілька ключових слів. Це допоможе людям знайти бібліотеку компонентів, коли вони шукають у npm. Npm.com використовує ці ключові слова, щоб допомогти розширити пошук. Є кілька інших налаштувань, які нам потрібно внести в package.json, створений додатком create-react. По-перше, тут версія 0.1.0, тому, коли ми опублікуємо наш пакет, буде опубліковано як 0.1.0. Далі нам потрібно видалити приватне налаштування. Ми також можемо просто встановити для нього значення false. Це невелика функція безпеки, яку налаштовує create-react-app, щоб уберегти нас від випадкової публікації пакунка npm, але оскільки ми хочемо зробити саме так, цей рядок потрібно видалити або встановити на false. Також потрібно перевірити, чи всі наші залежності розробника розміщено в розділі devDependencies. Це важливо, оскільки все, що перелічено тут у розділі залежностей, буде встановлено споживачами наших пакетів. Примушувати своїх споживачів встановлювати залежності лише для розробників, такі як Babel, Webpack тощо, є лише марною тратою їх пропускної спроможності та місця на жорсткому диску, тому будемо гарним та розмістимо лише фактичні виробничі залежності в розділі залежностей.

3.26. Опублікування пакету npm

Настав час розгорнути наш код у всьому світі, щоб вони могли насолоджуватися нашими захоплюючими компонентами. Для публікації нашого пакета потрібна одна команда, npm publish. Коли ми це зробимо, ми бачимо, що наша збірка виконується як частина попередньої публікації, а потім наш пакет публікується npm. Тепер ми зможемо перейти до npmjs.com, ввести назву нашого пакета та переглянути його.

Висновок до розділу 3

Ми спроектували та реалізували бібліотеку компонентів React для повторного використання. Розглянули створення кожного с компонентів та детального розглянули їх функціонал. Виконали транспіляцію через Babel та написали автоматизовані тести, підготували бібліотеку для опублікування та опублікували це в npm і задокументували всю нашу роботу.

ВИСНОВКИ

Програмне забезпечення проникає в наше повсякденне життя, можливо, немає іншого створеного людиною матеріалу, який був би більш повсюдним, ніж програмне забезпечення в нашому сучасному житті. Він став невід'ємною частиною багатьох частин суспільства, магазинів, телекомунікацій, побутової техніки, літаків, особистих розваг, аудиту, автомобілів тощо. В результаті все більше розробників програмного забезпечення використовують програмне забезпечення не тільки як комплексну систему, але і як модульну частину більшої системи. Повторне використання коду не означає, що ми зможемо копіювати та вставляти той самий код у багатьох частинах системи. Насправді це означає зовсім протилежне.

Існує велика кількість різноманітних патернів і каркасів програмних архітектур, які можуть вирішувати поставлені задачі, але для більш ефективної роботи з ними краще створити власну бібліотеку компонентів для їх перевикористання.

Розроблена в дипломній роботі технологія дає можливість запровадити повторне використання компонентів, що полегшує подальшу розробку проектів і збільшую швидкість розробки та тестування.

Розроблена бібліотека є актуальною, оскільки надає можливість на основі створеного проекту будувати архітектури різних типів програмних додатків і підтримувати їх.

Підсумовуючи виконану роботу, можемо стверджувати, що в результаті створення концепції, аналізу вимог, проектування та реалізації побудовано багаторазову бібліотеку компонентів React, яка вирішує більшість проблем сучасного програмування.

Застосування запропонованої бібліотеки під час проектування ПЗ дозволить за рахунок використання бібліотеки готових компонентів підвищити якість програмних продуктів, скоротити терміни розробки та зменшити сукупні витрати.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ReactJS [Електронний ресурс]. Режим доступу: ru.reactjs.org/docs/getting-started.html (дата звернення 05.12.2021) – Назва з екрана.
2. Javascript [Електронний ресурс]. Режим доступу: devdocs.io/javascript/ (дата звернення 06.12.2021) – Назва з екрана.
3. HTML [Електронний ресурс]. Режим доступу: devdocs.io/html/ (дата звернення 06.12.2021) – Назва з екрана.
4. Browser [Електронний ресурс]. Режим доступу: https://www.computerhope.com/jargon/b/browser.htm (дата звернення 07.12.2021) – Назва з екрана.
5. TypeScript [Електронний ресурс]. Режим доступу: https://www.typescriptlang.org/ (дата звернення 09.12.2021) – Назва з екрана.
6. JS vs TS [Електронний ресурс]. Режим доступу: https://www.guru99.com/typescript-vs-javascript.html (дата звернення 09.12.2021) – Назва з екрана.
7. Framework [Електронний ресурс]. Режим доступу: https://techmonitor.ai/what-is/what-is-a-framework-4945801 (дата звернення 09.12.2021) – Назва з екрана.
8. Bootstrap [Електронний ресурс]. Режим доступу: https://getbootstrap.com/ (дата звернення 09.12.2021) – Назва з екрана.