hillel IT school

Lesson 30

План заняття

- Види компонентів;
- Props;
- State в класовому компоненті;
- State batching;
- Події у react;

Render arrays

Щоб відобразити списки елементів. Вам потрібно перетворити список до валідних react emennets які react зможе відобразити на сторінці. Для цього потрібно викликати метод масивів map() та повернути валідні елементи.

Наприклад:

[1, 2, 3].map(el => <li key={el}>{el});

[1, 2, 3].map(el => <CustomComponent key={el} >el</CustomComponent>);

[1, 2, 3].map(el => el);

Ви повинні надати кожному елементу масиву ключ — рядок або число, яке унікально ідентифікує його серед інших елементів цього масиву.

Ключі повідомляють React, якому елементу масиву відповідає кожен компонент, щоб він міг зіставити їх пізніше. Це стає важливим, якщо ваші елементи масиву можуть переміщатися (наприклад, через сортування), вставлятися або видалятися. Правильно підібраний ключ допомагає React визначити, що саме сталося, і правильно оновити дерево DOM.

Докладніше тут https://react.dev/learn/rendering-lists

Render by conditions

React буде ігнорувати false, null, udefined. Ці значення не відрендаряться. Це. Надає нам чудовий механізм рендорігну за умовами.

{false && "Some text"} // no render

{null && "Some text"} // no render

{undefined && "Some text"} // no render

Для простих логічних умов ми можемо використовувати терні оператори { isTrue ? "Hello world!" : "Good by world!" }.

Для більш складних умов ми можемо написати свою влану функцію, яка вже поверне нам тещо ми хочемо відрендорити.

У середені jsx виразу {} не використовують else if або switch case. Це можна зробити у функції render(). Не міксуйте усі яйця у одній корзині.

Render by conditions

React буде ігнорувати false, null, udefined. Ці значення не відрендаряться. Це. Надає нам чудовий механізм рендорігну за умовами.

{false && "Some text"} // no render

{null && "Some text"} // no render

{undefined && "Some text"} // no render

Для простих логічних умов ми можемо використовувати терні оператори { isTrue ? "Hello world!" : "Good by world!" }.

Для більш складних умов ми можемо написати свою влану функцію, яка вже поверне нам тещо ми хочемо відрендорити.

У середені jsx виразу {} не використовують else if або switch case. Це можна зробити у функції render(). Не міксуйте усі яйця у одній корзині.

Render Fragment

Щоб відрендерити кілька елементів, не використовуючи DOM-елемент, що обертає, використовуйте React.Fragment.

Також можна використовувати <></> це теж саме що і <React.Fragment></React.Fragment>.

Render HTML

Бувають випадки, коли необхідно відрендерити готовий HTML-рядок. Наприклад, якщо прийшли дані із сервера, і вам потрібно відрендерити контент як є. Якщо ви передасте такий рядок у проп children - React просто відрендерить його як рядок.

Це потенційно небезпечна операція, тому React не допускає цього та рендерит HTML-рядок як рядок. Щоб виправити це використовуйте проп dangerouslySetInnerHTML.

Наприклад:

<div dangerouslySetInnerHTML={{ __html: someHTML }}/>

Він приймає об’єкт з властивістю __html, значення якої мусить бути html для рендорінга.

Portals

Трапляються випадки, коли необхідно відрендерити елемент за межами кореневого DOM-елемента. Наприклад, при рендері модальних вікон. Для цього використовуються портали. Для цього імпортуємо функцію createPortal.

```
import { createPortal } from 'react-dom';
```

Першим параметром функції createPortal є елемент. Це React-елемент.

Другий параметр - container. Це DOM-елемент, у якому потрібно рендерити переданий element.

Наприклад:

```
import { createPortal } from 'react-dom';
// в функции рендера
return createPortal(( <h1>Hello!</h1>
), document.getElementById('modals'));
```

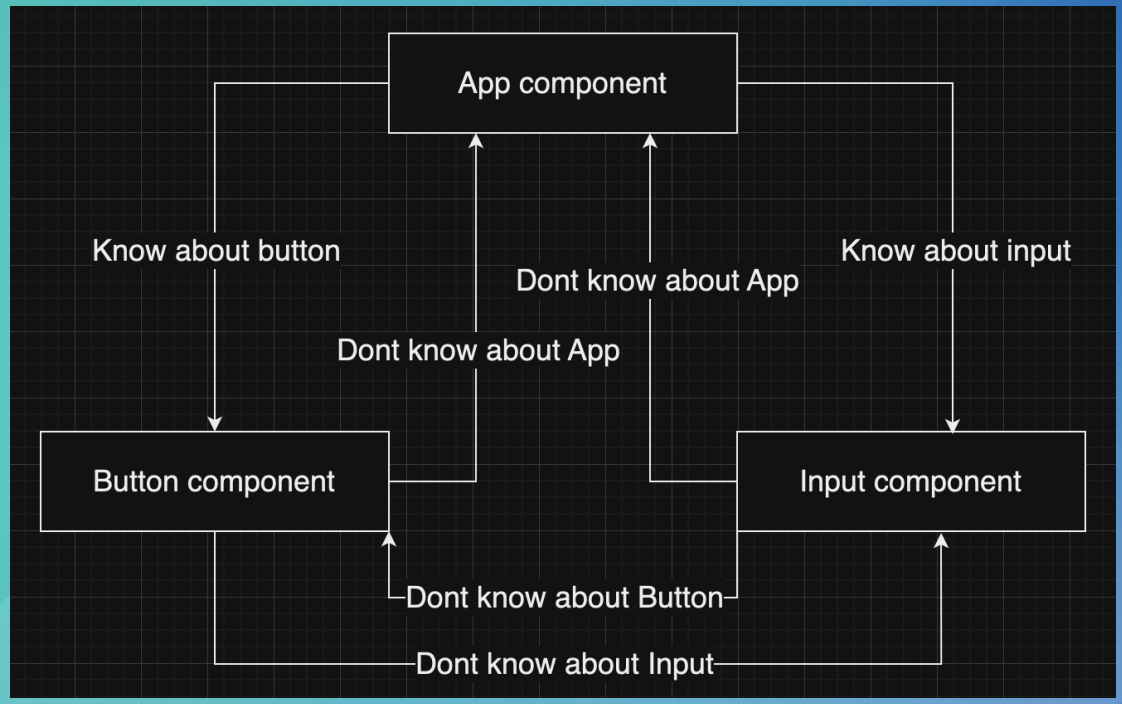
Components

Кожен компонент нічого не знає ні про додаток, ні про інші компоненти - він ізольований. Єдине про що знає компонент – це вхідні дані, пропси.

У середину компонента ми можемо передати:

- дочірній контент
- колббеки
- будь-які вхідні дані
- спец. пропси: key, ref, ...

Компоненти схожі на функції: у них можна передати набір параметрів, пропсів, а у відповідь отримати React-елемент.



Class Components

Клас повинен успадковувати від React.Component.

За рендер відповідає метод render – він має повернути React-елемент. React може викликати рендеринг у будь-який момент, тому ви не повинні вважати, що він запускається в певний час. Зазвичай метод render має повертати частину JSX.

Ви повинні написати метод render як чисту функцію, тобто він повинен повертати той самий результат, якщо властивості, стан і контекст однакові. Він також не повинен містити побічних ефектів (наприклад, налаштування підписок) або взаємодіяти з API браузера.

Коли строгий режим увімкнено, React двічі викличе рендер під час розробки, а потім викине один із результатів. Це допоможе вам помітити випадкові побічні ефекти, які потрібно усунути з методу візуалізації.

Пропси записуються як об'єкт props. До них можна звернутись через this.props у методі render.

Дочірні елементи доступні props.children.

У класових компонентах ми маємо доступ до життевих циклів react. Про них трохи з часом.

Більш детально тут https://react.dev/reference/react/Component

Props

Пропси дуже схожі на аргументи функції:

- кожен пропс схож на окремий іменований аргумент;
- можна надавати значення за замовчуванням;
- можна вказувати тип аргументу, майже як у TypeScript.

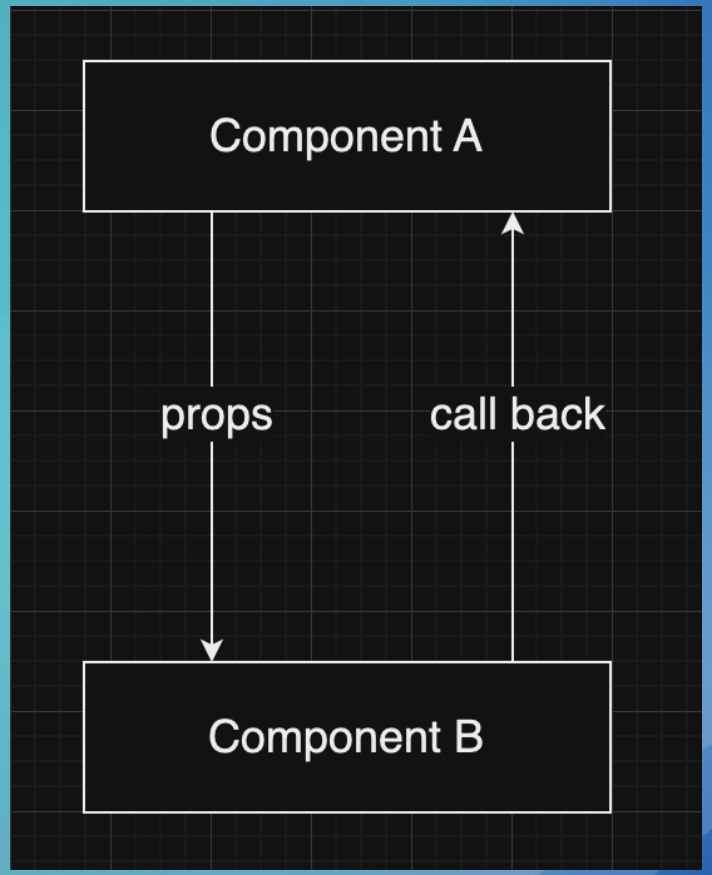
Пропси умовно можна поділити на 3 види:

- React-пропси: key, ref, dangerouslySetInnerHTML;
- пропси користувача (що завгодно);
- дочірній контент (children).

Ідея передачі даних:

Дані йдуть від кореневого вузла до його цільового нащадка, з верху до низу. Це називається бурінням пропсів.

Якщо нащадок хоче змінити данні то він мусить викликати callback кореневого елементу, знизу до гори. Так корінь буде знати що потрібно змінити дані і відрендерити новий дочірній елемент.



Props validation

Оскільки JS - це мова з динамічною типізацією, ми не можемо вказати пропси якого типу очікуємо отримати. А дуже часто робити валідацію потрібно, наприклад при розробці UI-бібліотеки.

Для цього у React-стеку є пакет prop-types.

Для того, щоб вказати типи пропсів, потрібно класу або функції компонента присвоїти статичну властивість propTypes.

У нього потрібно присвоїти об'єкт, де ключ - це назва пропса, а значення - це функція, яка буде викликана для валідації.

Дока тут https://react.dev/reference/react/Component#static-proptypes

Бібліотека тут https://www.npmjs.com/package/prop-types

Default props

Для того, щоб вказати значення пропсів за умовчанням, потрібно класу або функції компонента присвоїти статичну властивість defaultProps.

У нього потрібно присвоїти об'єкт, де ключ - це назва пропсу, а значення - це власне дефолтне значення пропсу.

Якщо компонент не буде передано якийсь пропс, React спробує встановити значення пропсу за замовчуванням (якщо воно задано). Після цього буде запущено перевірку типів пропсів.

Дефолтні значення теж валідуватимуться.

State in class component

Для роботи зі станом потрібно визначитися, за чим ми стежитимемо і які є стани.

Наприклад:

- Відкрито / закрито - для меню, списків, що випадають
- Вибрана опція - для списку опцій - Вибрані категорії - для списку категорій
- Текстове значення - для текстових полів
- ...

Після цього стан можна описати як об'єкт із певними властивостями.

Це можна зробити або у конструкторі, або використовуючи сучасний синтаксис JS як властивість класу.

Важливо!

State завжди є об’єктом у середині якого може бути що завгодно.

Взяти щось з state можна через this.state

Змінити стан можна за допомогою this.setState()

State in class component

Для роботи зі станом потрібно визначитися, за чим ми стежитимемо і які є стани.

Наприклад:

- Відкрито / закрито - для меню, списків, що випадають
- Вибрана опція - для списку опцій - Вибрані категорії - для списку категорій
- Текстове значення - для текстових полів
- ...

Після цього стан можна описати як об'єкт із певними властивостями.

Це можна зробити або у конструкторі, або використовуючи сучасний синтаксис JS як властивість класу.

Важливо!

State завжди є об’єктом у середині якого може бути що завгодно.

Взяти щось з state можна через this.state

Змінити стан можна за допомогою this.setState()

setState

setState буде оновлювати стан вашого компонента React. Не викликайте setState у середині render() воно визову нескінченний цикл оновлення інтерфейсу.

Синтаксис:

setState(nextState, callback?)

setState ставить у чергу зміни стану компонента. Він повідомляє React, що цей компонент і його дочірні компоненти мають повторно відобразити з новим станом. Це основний спосіб оновлення інтерфейсу користувача у відповідь на взаємодію.

Ви також можете передати функцію setState. Це дозволяє оновлювати стан на основі попереднього стану.

Якщо ви передаєте об'єкт як nextState, він буде об'єднаний у this.state.

Якщо ви передаєте функцію як nextState, вона розглядатиметься як функція оновлення. Віна має бути чистою, має приймати стан очікування та властивості як аргументи та має повертати об'єкт, який буде об'єднано в this.state. React поставить вашу функцію оновлення в чергу та повторно візуалізує ваш компонент. Під час наступного рендерингу React обчислить наступний стан, застосовуючи всі програми оновлення в черзі до попереднього стану.

State batching

Бувають випадки коли вам потрібно викликати декілька разів setState() у одному колбєку. Це може спричинити не очевидну поведінку. Це трапляється тому що React чекає, доки весь код в обробниках подій не буде виконано, перш ніж обробити ваші оновлення стану. Ось чому повторна візуалізація відбувається лише після всіх цих викликів incrementNumber().

Для вирішення цієї проблеми вам потрібно використовувати колбек у setState() де перший параметр буде поточний стан компоненту.

Тому у даному контексту потрібно використовувати setState так

```
setState( state => ({number: state.number + 1}) )
```

Це гарантує що ви отримаєте один рендерінг з правильними даними.

setState with obj and arr

State доступер тільки для читання тому ми не можемо просто взяти та додати нову власивість або елемент за індексом до стану. Для цього потрібно використовувати деструктурізацію.

Приклад з об’єктом

```
setState( state => ( { user: { ...state.user, key: value } } ) );
```

Приклад з масивом

```
setState( state => ( { array: [ ...state.array, newItem ] } ) );
```

Більш детально тут

Масив https://react.dev/learn/updating-arrays-in-state

Об’єкт https://react.dev/learn/updating-objects-in-state

React events

Для прив'язки DOM-події React-елемент потрібно передати проп, ім'я якого будується як:

- on + EventName;
- onClick onKeyPress onFocus.

У проп потрібно передати функцію-обробник події. При її виклик першим агрументом буде об'єкт синтетичної події.

React робить однаковий інтерфей подій у всех браузерах, а також додає свої власні параметри та методи:

- nativeEvent: подія DOM. Оригінальний об’єкт події браузера;
- isDefaultPrevented(): повертає логічне значення, яке вказує, чи було викликано preventDefault;
- isPropagationStopped(): повертає логічне значення, яке вказує, чи був виклик stopPropagation;
- persist(): не використовується з React DOM. За допомогою React Native викличте це, щоб прочитати властивості події після події;
- isPersistent(): не використовується з React DOM. З React Native повертає, чи було викликано persist.

Детальніше тут https://react.dev/reference/react-dom/components/common#react-event-object

React events and state

Всередині обробників подій ми можемо запитувати на оновлення стану, тим самим замикаючи цикл інтерактивності.

У класових компонентах обробники подій зазвичай виносять методи класу. Це зручно та оптимально для продуктивності. Але є одна проблема – за умовчанням методи класу не прив'язані до контексту.

Тобто: якщо прямо в JSX передати обробник як this.handlerName, тоді в момент виконання JS this всередині функції обробника дорівнюватиме undefined.

Рішення проблеми - .bind(this) до метода, або зробити стрілочну функцію у методі.

Також є експериментальний синтаксис:

onClick={::this.handleToggle} це аналог this.handleToggle.bind(this)

React events and state

Насправді React прив'язує події до кореневого елемента програми (починаючи з версії 17 до події прив'язувалися до document). У момент спрацювання нативної події React робить таке:

- Через event.target дізнається, на якому DOM-елементі спрацювала нативна подія
- Створює екземпляр SynteticEvent, копіює в нього дані з нативної події, стандартизує їх
- SynteticEvent надає властивості target значення event.target
- Знаючи структуру DOM та дерева компонентів, React знаходить DOM-елемент, до якого ми прив'язували подію у JSX і цей DOM-елемент присвоювати у SynteticEvent.currentTarget
- Викликає функцію-обробник події, яку ми додали до JSX та передає до неї SynteticEvent

React with forms

Форми - це насамперед поля (input, поле введення). У React з полями можна працювати у двох режимах:

- кероване поле
- некероване поле

Поля HTML вже самі собою інтерактивні - браузер перебирає реалізацію інтерактивності. React в цей цикл інтерактивності не втручається - все робить сам браузер. Такі поля називаються некерованими.

Якщо нам потрібно контролювати цикл інтерактивності, то його реалізацію нам потрібно робити самим – за допомогою обробки подій, встановлення стану та надання значення полю. Такі поля називаються керованими.

React with forms

Щоб створити керований елемент, потрібно зав’язати його значення на стані компонету, та додати обробник події.

Наприклад:

```
export class SearchForm extends React.Component {
  state : {query: string}  = { query: "" };
  1 usage  new *
  handleChange = (e) : void  => {
    this.setState( state: { query: e.target.value });
  };
  no usages  new *
  render() {
    const { query : string  } = this.state;
    return <input value={query} onChange={this.handleChange} type="text" />;
  }
}
```

Також тут ви можете валідувати вхідні дані, як у звичайному js, але зберігати стан валідації у середині стану компонента.

React with forms

Для роботи з некерованими компонентами є два варіанти:

- додаємо обробник події onChange і при зміні значення копіюємо його в state але не зав’язуємо його на value
- у момент надсилання форми звертаємося до DOM-елемента кожного поля та беремо з нього значення поля. Для цього потрібно використовувати рефи (ref) – їх ми розберемо на наступних заняттях.

React with forms

Для перехоплення відправки форми є подія onSubmit. В обробнику події ми скасовуємо відправлення форми браузером і робимо щось.

<form onSubmit={this.submitHandler.bind(this)}>

- Якщо всі поля керовані - значення полів беремо з state.
- Якщо поля не керовані – але значення полів беремо безпосередньо з DOM.

Дякую за увагу