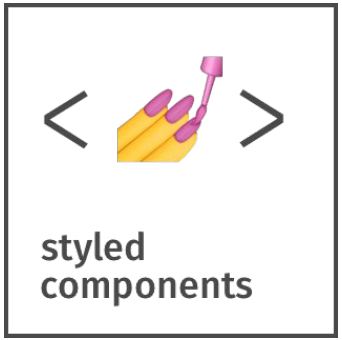
Розробка мобільних додатків

Лекція 5 - Стилізація



Стилізація у **React Native**

У середовищі React Native не існує універсального способу стилізації інтерфейсу. Вибір підходу - це компроміс між **швидкістю розробки**, **продуктивністю додатку** та **підтримкою коду**.

Основні критерії вибору:

1. **Runtime Performance:** наскільки підхід впливає на швидкодію інтерфейсу
2. **Developer Experience:** наскільки швидко розробник може вносити зміни
3. **Scalability:** наскільки підхід залишається керованим із ростом застосунку

StyleSheet — Нативний стандарт

- **Плюси:** Найвища продуктивність (без додаткових обчислень). Не збільшує розмір бандла. Всі помилки підсвічуються відразу.
- **Мінуси:** Погана робота з динамікою (доводиться писати тернарні оператори у масивах стилів).
- **Підходить** для простих компонентів та критичних до швидкості ділянок (великі списки).

Styled-components — Декларативність та DX

- **Плюси:** Чистий JSX (немає `style={styles.container}`). Найкраща робота з пропсами. Підтримка повноцінного CSS-синтаксису.
- **Мінуси:** Невеликий "оверхед" по продуктивності (бібліотека витрачає час на парсинг рядків). Збільшує розмір додатка.
- **Підходить** для команд, де важлива читабельність коду та системність дизайну.

Utility-first CSS - NativeWind

- **Плюси:** Швидкість написання коду. Легко переносити дизайн із веб-версії проекту. Стилі автоматично компілюються в StyleSheet (висока швидкість).
- **Мінуси:** Велика кількість назв класів ускладнює читабельність та сприйняття програмного коду. Потребує налаштування Babel-плагіну, що іноді створює проблеми при оновленні версій React Native.
- **Підходить** для стартапів та швидкого прототипування.

UI-бібліотеки

- **Плюси:** Надаються готові компоненти інтерфейсу (кнопки, модальні вікна, списки тощо), які вже протестовані з точки зору доступності та стабільності роботи. Значно скорочується час розробки завдяки використанню готових UI-рішень.
- **Мінуси:** Обмежені можливості кастомізації дизайну, особливо якщо він відрізняється від закладеної дизайн-системи. Збільшення розміру бандлу застосунку через підключення великої кількості готових компонентів.
- **Підходить** для внутрішніх корпоративних інструментів, де дизайн менш важливий за швидкість реалізації фіч.

StyleSheet.create()

Створення стилів. Жодних додаткових параметрів або опцій метод НЕ має.

```
const styles = StyleSheet.create({
  box: {
    padding: 16,
    backgroundColor: "blue",
  },
});
```

Що робить:

- **перевіряє коректність заданих властивостей стилю** — забезпечує відповідність параметрів допустимим значенням та запобігає використанню некоректних або несумісних властивостей;
- **оптимізує структуру стилів** — перетворює опис стилів у внутрішньо оптимізований формат, що підвищує ефективність їх застосування під час рендерингу інтерфейсу;
- **зменшує кількість створюваних об’єктів у пам’яті** — дозволяє повторно використовувати стилі, що знижує навантаження на механізм керування пам’яттю та покращує продуктивність застосунку.

StyleSheet.compose()

Метод використовується для **об’єднання двох стилів** в один підсумковий стиль без створення зайвих масивів або об’єктів.

Сигнатура методу

StyleSheet.compose(style1, style2)

Параметри:

- `style1` — перший стиль;
- `style2` — другий стиль, який має пріоритет при конфлікті властивостей.

Принцип роботи

Метод:

- поєднує два стилі в один;
- якщо обидва стилі містять однакові властивості — застосовується значення з `style2`;
- оптимізує використання пам’яті, уникаючи створення нового масиву стилів.

```
import { StyleSheet, View } from 'react-native';

const styles = StyleSheet.create({
  base: {
    padding: 10,
    backgroundColor: 'white',
  },
  active: {
    backgroundColor: 'blue',
  },
});

export default function App() {
  const combinedStyle = StyleSheet.compose(
    styles.base,
    styles.active
  );

  return <View style={combinedStyle} />;
}
```

```
LOG [{"backgroundColor": "white", "padding": 10}, {"backgroundColor": "blue"}]
```

StyleSheet.flatten()

Метод використовується для **розгортання (об’єднання) набору стилів у єдиний фінальний об’єкт стилю**, який містить усі застосовані властивості.

Сигнатура методу

StyleSheet.flatten(style)

Параметри

style — стиль або набір стилів:

Принцип роботи

Метод:

- об’єднує декілька стилів у **один об’єкт**;
- рекурсивно розгортає вкладені масиви стилів;
- якщо властивості повторюються — застосовується значення **останнього стилю**;
- повертає фінальний JavaScript-об’єкт стилів.

```
const styles = StyleSheet.create({
  base: {
    padding: 20,
    backgroundColor: 'white',
    borderWidth: 1,
    borderColor: '#000',
  },
  active: {
    backgroundColor: 'blue',
  },
});

const finalStyle = StyleSheet.flatten([
  styles.base,
  styles.active,
  { margin: 20 },
]);

console.log('FINAL STYLE:', finalStyle);

return (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <View style={finalStyle}>
```

```
{"backgroundColor": "blue", "borderColor": "#000", "borderWidth": 1, "margin": 20, "padding": 20}
```

Зі збільшенням розміру проєкту використання стандартного механізму `StyleSheet` може ускладнювати підтримку інтерфейсу. Кількість стилів зростає, виникає дублювання, а стилізація поступово розподіляється між різними частинами коду, що знижує читабельність і ускладнює розвиток застосунку.

Для розв’язання цих проблем застосовується бібліотека **Styled Components**, яка реалізує підхід **CSS-in-JS** у середовищі React Native.

Styled Components

Styled Components – це бібліотека для стилізації компонентів у React та React Native. Вона дозволяє створювати **кастомні стилізовані компоненти** прямо у коді, використовуючи **CSS-подібний синтаксис**.

Основні переваги:

- Створення **повторно використовуваних стилів**
- Відокремлення логіки та стилів
- Підтримка **тем**
- Динамічні стилі (залежні від `props`)

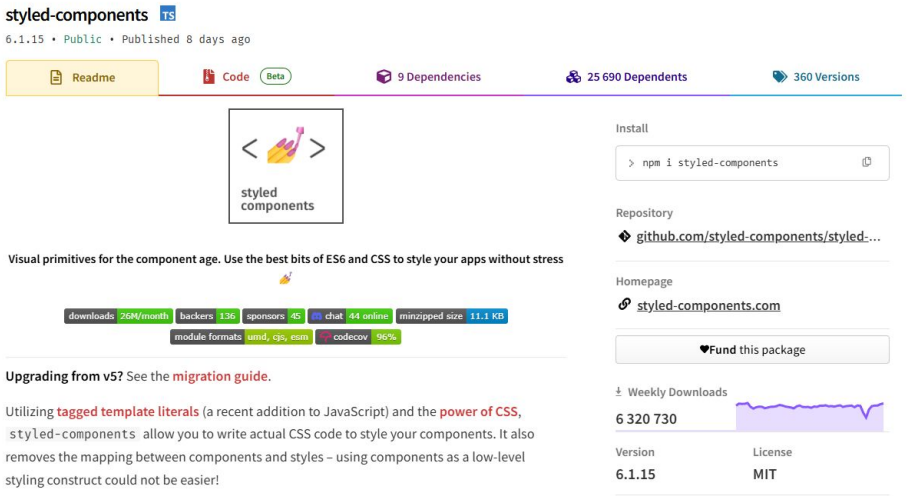
Установка **Styled Components**

Команди:

npm install styled-components
npm install --save-dev
@types/styled-components-react-native

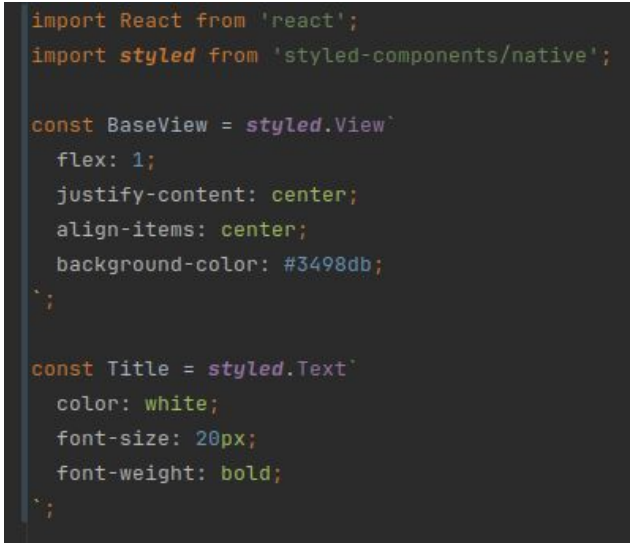
Правило імпорту: Для React Native ми **завжди** імпортуємо з закінченням `/native`:

```
import styled from
'styled-components/native';
```



Як працює **Styled Components?**

Styled Components працює на основі **шаблонних рядків** (template literals) у JavaScript:



```
import React from 'react';
import styled from 'styled-components/native';

const BaseView = styled.View`
  flex: 1;
  justify-content: center;
  align-items: center;
  background-color: #3498db;
`;

const Title = styled.Text`
  color: white;
  font-size: 20px;
  font-weight: bold;
`;
```

Як працюють **Tagged Template Literals?**

Tagged Template Literals — це механізм JavaScript, який дозволяє викликати функцію, використовуючи шаблонні рядки (` `) замість круглих дужок.

Звичайний виклик функції:

```
func(argument);
```

Виклик через tagged template:

```
func`argument`;
```

У цьому випадку функція отримує вміст шаблонного рядка та може його програмно обробляти.

3 `Tagged Template Literals` ми можемо створити власну функцію для обробки рядка:

```
function customTag(strings, ...values) {
  console.log(strings); // Масив частин рядка
  console.log(values);  // Масив змінних із ${}
  return strings.join('Світ');
}

const name = "Світ";
console.log(customTag`Привіт, ${name}!`);
// Виведе: Привіт, Світ!
```

Styled Components дає більш декларативний підхід ніж **StyleSheet:**

```
import styled from 'styled-components/native';

const StyledView = styled.View`
  flex: 1;
  justify-content: center;
  align-items: center;
  background-color: lightblue;
`;

const StyledText = styled.Text`
  font-size: 20px;
  color: white;
`;

Show usages  kipz_nsi *
export default function App() {
    return (
        <StyledView>
            <StyledText>Styled Components y React Native</StyledText>
        </StyledView>
    );
}
```

Динамічні стилі через **props**

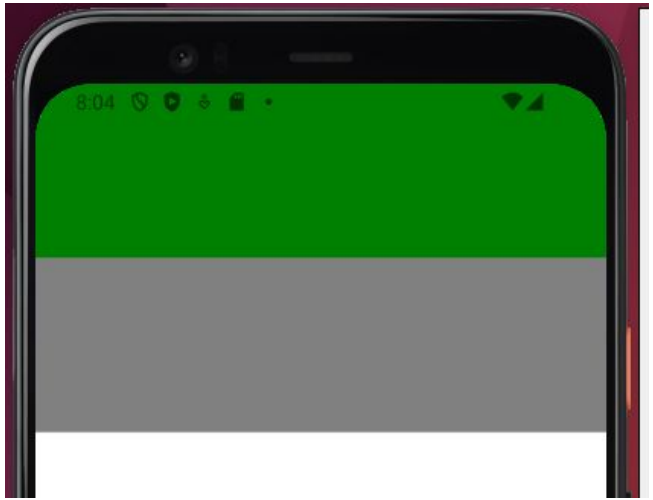
Styled Components дозволяє змінювати стилі компонента залежно від **props**, використовуючи JavaScript-вирази всередині шаблонних рядків. Це дає змогу створювати різні варіанти одного компонента без дублювання стилів.

Синтаксис

property: ${props => умова ? значення1 : значення2};

Динамічні стилі через **props**

```
import React from 'react';
import styled from 'styled-components/native';

const Box = styled.View`
  padding: 60px;
  background-color: ${props =>
  props.active ? 'green' : 'gray'};
`;

export default function App() {
  return (
    <>
      <Box active />
      <Box />
    </>
  );
}
```



Функції для обчислення параметрів

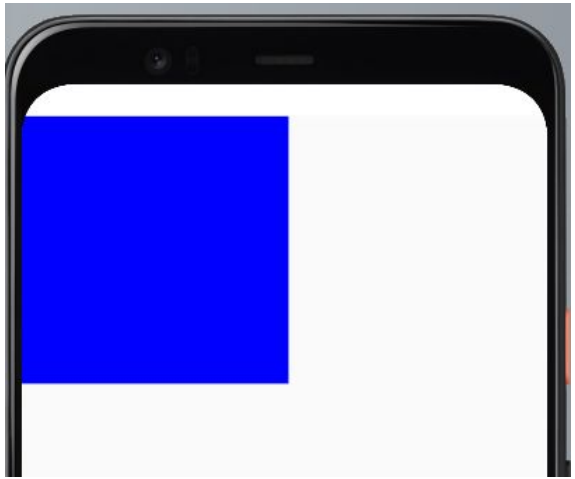
Styled Components дозволяє використовувати **JavaScript-функції** для обчислення значень стилів. Це дає змогу динамічно змінювати параметри оформлення залежно від `props`, стану або логіки застосунку.

Синтаксис

property: ${props => функція(props)};

Функції для обчислення параметрів

```
const getSize = (size) : string => {
    switch (size) {
        case "small":
            return "50px";
        case "large":
            return "200px";
        default:
            return "100px";
    }
};

const Box = styled.View`
  width: ${(props) : string => getSize(props.size)};
  height: ${(props) : string => getSize(props.size)};
  background-color: blue;
`;

Show usages  kipz_nsi *
export default function App() {
    return <Box size="large" />;
}
```



Transient Props ($prop)

Transient props — це спеціальні props, які починаються з символу **$** і використовуються **лише для стилізації**, не передаючись у native компонент.

Це дозволяє уникнути передачі зайвих або невалідних властивостей у React Native елементи.

Проблема без transient props

```
const Button = styled.View`
 background-color: ${props =>
   props.primary ? 'blue' : 'gray'};
`;

<Button primary />
```

`primary` буде переданий у `View`, хоча він там не потрібен.

```
const Container = styled(SafeAreaView)`
  flex: 1;
  padding: 20px;
`;

const Button = styled.TouchableOpacity`
  padding: 15px;
  border-radius: 10px;
  align-items: center;
  margin-bottom: 15px;
  background-color: ${p =>
  p.$danger ? '#e74c3c' : '#3498db'};
`;

const TextStyled = styled.Text`
  color: white;
  font-weight: bold;
  font-size: 16px;
`;
```

```
export default function App() {
  return (
    <Container edges={['top', 'bottom']}>
      <Button>
        <TextStyled>Primary</TextStyled>
      </Button>

      <Button $danger>
        <TextStyled>Danger</TextStyled>
      </Button>
    </Container>
  );
}
```

`.withConfig()` — керування передачею **props**

Метод **`.withConfig()`** дозволяє налаштовувати поведінку styled component, зокрема **контролювати, які props передаються (forward) у базовий React Native компонент**.

Використовується для запобігання передачі службових props у native елементи.

Синтаксис

```
styled(Component).withConfig({
 shouldForwardProp: (prop) => boolean
})
```

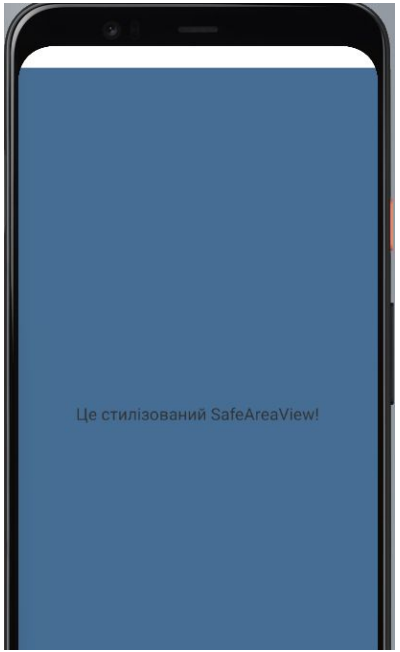
- `true` → prop передається вниз
- `false` → prop блокується


const Button = styled.TouchableOpacity.withConfig({
shouldForwardProp: prop => prop !== 'primary',
})`
padding: 15px;
border-radius: 10px;
align-items: center;
margin-bottom: 16px;
background-color: ${p =>
p.primary ? '#3498db' : '#95a5a6'};
`;

Як обернути компонент, якщо його немає у **styled-components/native**

Якщо потрібно стилізувати компонент, якого **немає у списку `styled-components/native`** (наприклад, `SafeAreaView`), потрібно передати його в функцію `styled()`.

```
// 📌 Обгортаємо SafeAreaView
const StyledSafeArea = styled(SafeAreaView)`
    flex: 1;
    background-color: #476d93;
    justify-content: center;
    align-items: center;
`;
```



attrs()

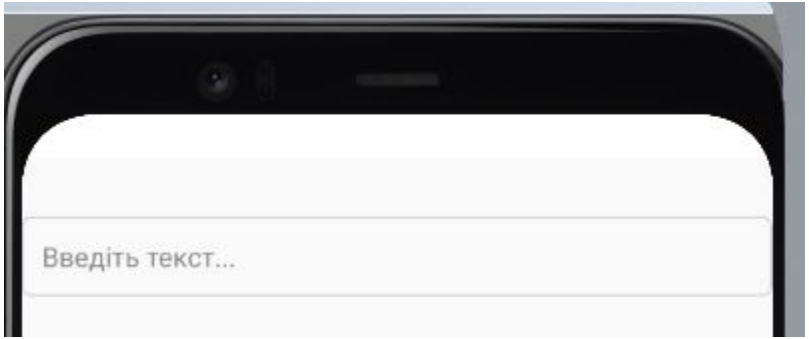
Метод **`attrs()`** дозволяє задавати **статичні або обчислювані props** для styled component під час його створення. Це дає можливість встановлювати значення за замовчуванням і зменшувати дублювання коду.

Синтаксис

```
styled.Component.attrs({
 prop: value
})`
 стилі
`;
```

Використання **attrs()** для задання значень за замовчуванням

```
import styled from "styled-components/native";

const Input : ... = styled.TextInput.attrs( attrs: {
    placeholder: 'Введіть текст...',
    placeholderTextColor: '#888',
})`
    width: 100%;
    padding: 10px;
    border: 1px solid #ccc;
    border-radius: 5px;
`;

Show usages  kipz_nsi *
export default function App() {
    return (
        <Input/>
    );
}
```



ThemeProvider у Styled Components

`ThemeProvider` — це компонент із **styled-components**, який дозволяє передавати **тему** всередині React-застосунку. Він забезпечує глобальний доступ до стилів, таких як кольори, відступи, розміри шрифтів тощо.

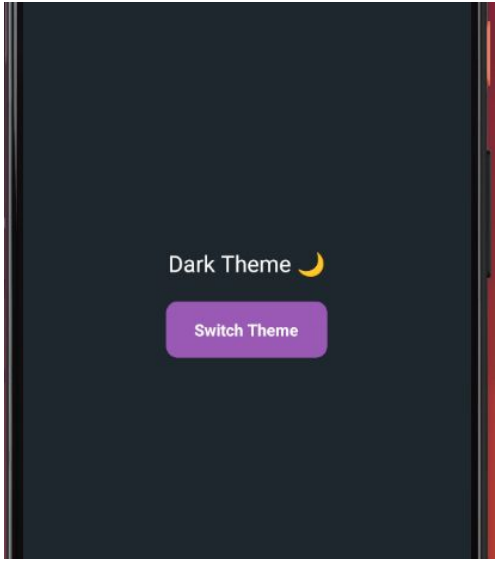
```
import styled, { ThemeProvider } from 'styled-components/native';
```

Глобальні стилі через **ThemeProvider**

```
// Світла тема
const lightTheme = {
  colors: {
    background: '#ecf0f1',
    primary: '#3498db',
    text: '#2c3e50',
  },
};

// Темна тема
const darkTheme = {
  colors: {
    background: '#1e272e',
    primary: '#9b59b6',
    text: '#ffffff',
  },
};
```

```
const Container = styled.View`
  flex: 1;
  justify-content: center;
  align-items: center;
  background-color: ${props => props.theme.colors.background};
`;
```

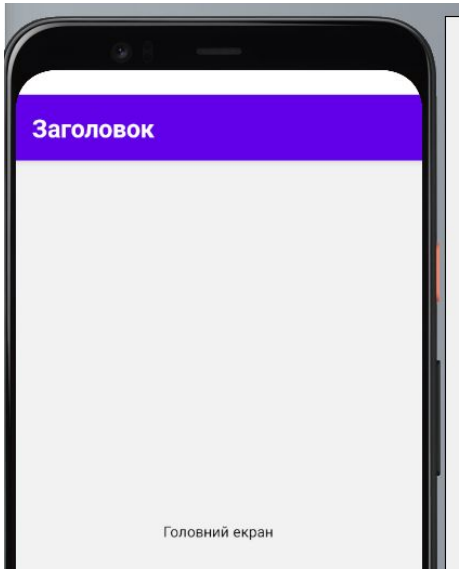
```
const [dark, setDark] = useState(false);

return (
  <ThemeProvider theme={dark ? darkTheme : lightTheme}>
    <Container>
      <Title>
        {dark ? 'Dark Theme 🌙' : 'Light Theme ☀️'}
      </Title>

      <TouchableOpacity onPress={() => setDark(!dark)}>
        <Button>
          <ButtonText>Switch Theme</ButtonText>
        </Button>
      </TouchableOpacity>
    </Container>
  </ThemeProvider>
);
```



Інтеграція **Styled Components** у **React Navigation**

У React Native Navigation не є звичайним `View`, тому її стилізація відбувається styled-компонентів у `screenOptions`.

```
const StyledHeader = styled.Text`
    font-size: 24px;
    color: #ffffff;
    font-weight: bold;
`;

Show usages  kipz_nsi *
export default function App() {
    return (
        <NavigationContainer>
            <Stack.Navigator
                screenOptions={{
                    headerStyle: { backgroundColor: '#6200ea' },
                    headerTitle: (props : HeaderTitleProps ) => <StyledHeader {...props}>Заголовок</StyledHeader>,
                }}>
                <Stack.Screen name="Home" component={HomeScreen} />
            </Stack.Navigator>
        </NavigationContainer>
    );
}
```



extends

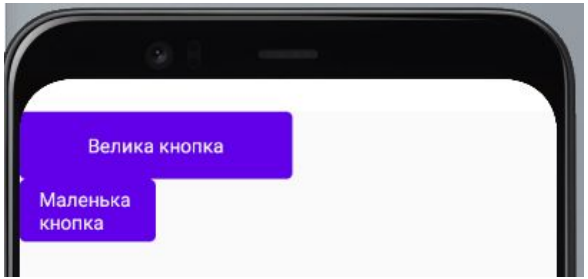
Styled Components дозволяє **успадковувати стилі одного компонента іншим**, розширюючи базовий компонент новими властивостями. Це забезпечує повторне використання стилів і зменшує дублювання коду.

Синтаксис

```
const NewComponent = styled(BaseComponent)`
 додаткові стилі
`;
```

Використання **extends** для створення похідних сти

```
const BaseButton = styled.TouchableOpacity`
  background-color: #6200ea;
  padding: 10px;
  border-radius: 5px;
  align-items: center;
`;

const LargeButton = styled(BaseButton)`
  padding: 15px;
  width: 200px;
`;

const SmallButton = styled(BaseButton)`
  padding: 5px;
  width: 100px;
`;
```



Створення умовних стилів за допомогою **`css`**

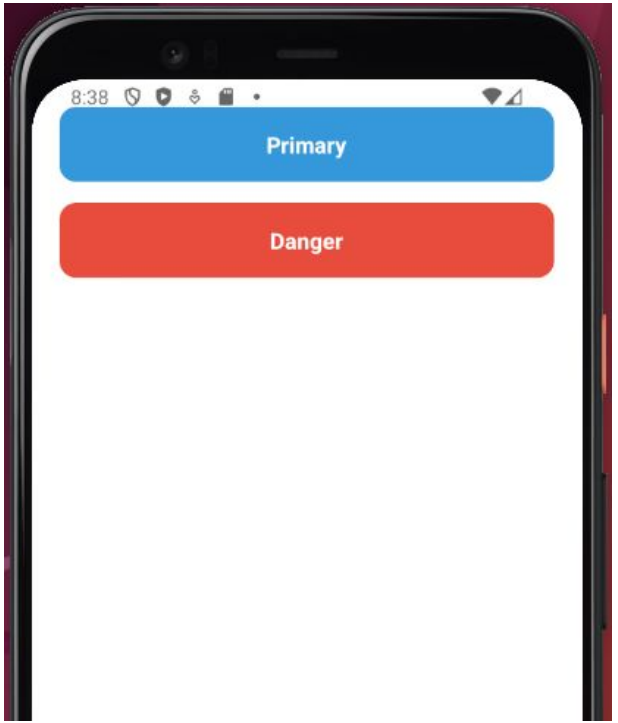
Функція **`css`** у Styled Components дозволяє створювати **умовні блоки стилів**, які можна динамічно застосовувати залежно від `props` або стану компонента. Це спрощує роботу зі складними умовними стилями.

Синтаксис

```
import { css } from 'styled-components/native';

${props => props.condition && css`
 стилі
`}
```

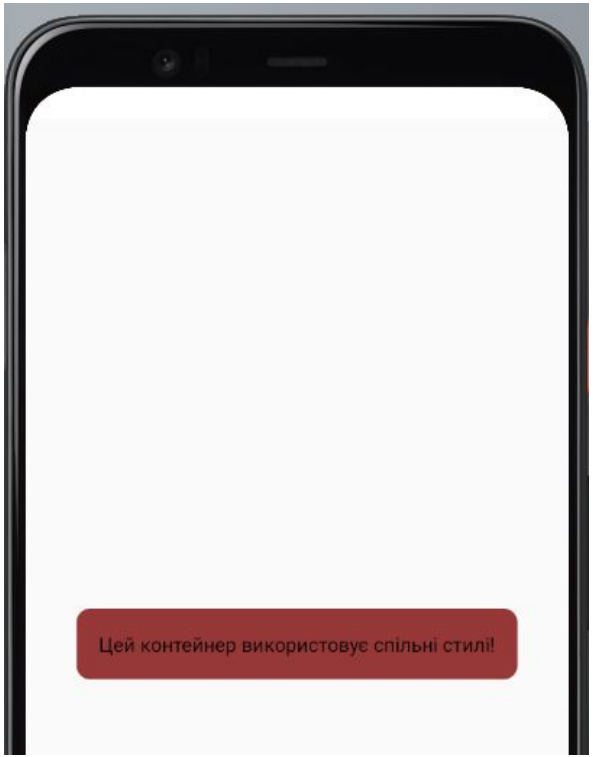
Створення умовних стилів за допомогою **css**

```
// Умовні стилі
const primaryStyle = css`
  background-color: #3498db;
`;

const dangerStyle = css`
  background-color: #e74c3c;
`;

const Button = styled.TouchableOpacity`
  padding: 16px;
  border-radius: 12px;
  margin-bottom: 15px;
  align-items: center;
  background-color: gray;

  ${props => props.primary && primaryStyle}
  ${props => props.danger && dangerStyle}
`;
```



Використання **css** для групування стилів

```
import styled, { css } from 'styled-components/native';

const commonStyles = css`
    padding: 16px;
    border-radius: 10px;
    background-color: #963838;
`;

const Box = styled.View`
    ${commonStyles};
`;

Show usages  kipz_nsi *
export default function App() {
    return (
        <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
            <Box>
                <Text>Цей контейнер використовує спільні стилі!</Text>
            </Box>
        </View>
    );
}
```



as

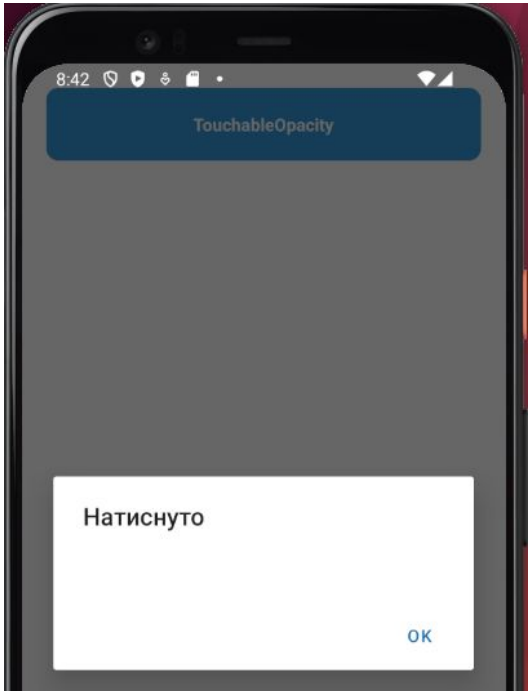
Проп **as** дозволяє змінювати тип компонента, який буде відрендерений, **без зміни стилів**. Один styled component може поводитися як різні React Native компоненти.

Синтаксис

```
<Component as={ІншийКомпонент} />
```

Використання **as** для зміни типу компонента

```
const Box = styled.View`
  padding: 20px;
  background-color: #3498db;
  border-radius: 10px;
  align-items: center;
  margin: 20px;
`;

const TextStyled = styled.Text`
  color: white;
  font-weight: bold;
`;

export default function App() {
  return (
    <Box
      as={TouchableOpacity}
      onPress={() => Alert.alert('Натиснуто')}
    >
      <TextStyled>TouchableOpacity</TextStyled>
    </Box>
  );
}
```



isStyledComponent

`isStyledComponent` — хук Styled Components, що дозволяє перевірити, чи є певний компонент **styled component**, а не звичайним React-компонентом.

```
import React from 'react';
import styled, { isStyledComponent } from 'styled-components/native';
import { View } from 'react-native';

const StyledBox = styled.View`
  padding: 20px;
  background-color: blue;
`;

console.log(isStyledComponent(StyledBox)); // true
console.log(isStyledComponent(View));      // false

export default function App() {
  return <StyledBox />;
}
```

useTheme

`useTheme` – це **хук** від Styled Components, який дозволяє отримати **поточну тему** (`theme`) у будь-якому компоненті. Це зручно, коли потрібно отримати **кольори, розміри шрифтів та інші стилі** без передачі `props`.

```
import styled, { ThemeProvider, useTheme } from 'styled-components/native';
```

```
const theme = useTheme(); // Використовуємо тему всередині компонента

console.log(theme);
```

useColorScheme

`useColorScheme` – це **хук у React Native**, який визначає, чи пристрій використовує **світлу (`light`) або темну (`dark`) тему**.

- Він **динамічно змінюється**, якщо користувач перемикає тему у системних налаштуваннях.
- Працює **на iOS, Android та Web** (у Expo Web).


```tsx
export default function App() {
    const theme : {background: string, text: string}  = useColorScheme() === 'dark' ? darkTheme : lightTheme;

    return (
        <ThemeProvider theme={theme}>
            <Container>
                <ThemedText>Поточна тема: {useColorScheme()}</ThemedText>
            </Container>
        </ThemeProvider>
    );
}
```

React Native Dimensions

`Dimensions` – це API у **React Native**, що дозволяє отримувати розміри екрану пристрою. Це корисно для **адаптивного дизайну**, коли потрібно коригувати UI під різні розміри екранів.

Як отримати розміри екрану

```
import { Dimensions } from 'react-native';

const { width, height } = Dimensions.get('window');

console.log(`Ширина: ${width}, Висота: ${height}`);
```

`Dimensions.get('window')` повертає об'єкт: `{ width: 390, height: 844, scale: 3, fontScale: 1 }`

- **`width`** – ширина екрана
- **`height`** – висота екрана
- **`scale`** – коефіцієнт масштабування (DPI)
- **`fontScale`** – масштаб тексту

Що таке коефіцієнт масштабування **(DPI)** у **React Native?**

DPI (Dots Per Inch, "точок на дюйм") – це **щільність пікселів на дюйм екрана**.

У контексті React Native, DPI часто згадується як **Pixel Ratio (коефіцієнт масштабування пікселів)**, який визначає, **скільки фізичних пікселів міститься в одному логічному пікселі**.

window vs screen:

```
Dimensions.get('window'); // Включає статусбар, не включає навігаційні кнопки

Dimensions.get('screen'); // Включає ВСЮ область екрану
```

- `window` – використовується для верстки інтерфейсу
- `screen` – включає навігаційні кнопки та статусбар

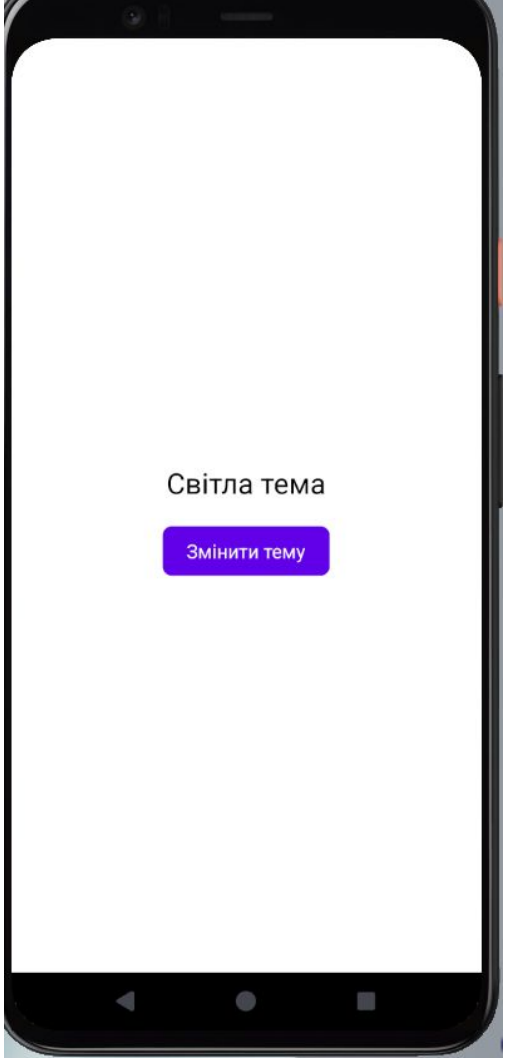
Перевірка відмінностей:

```
const windowSize = Dimensions.get('window');

const screenSize = Dimensions.get('screen');

console.log('Window:', windowSize);

console.log('Screen:', screenSize);
```

```
(NOBRIDGE) LOG Window: {"fontScale": 1, "height": 829.0909090909091
, "scale": 2.75, "width": 392.72727272727275}
(NOBRIDGE) LOG Screen: {"fontScale": 1, "height": 829.0909090909091
, "scale": 2.75, "width": 392.72727272727275}
```

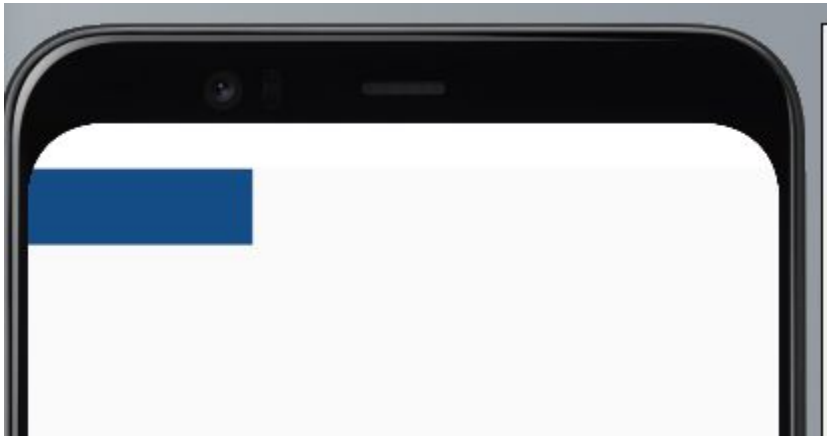
На Android `screen.height` може бути більше через навігаційні кнопки.

```
 (NOBRIDGE) LOG  Window: {"fontScale": 1, "height": 805.0909
090909091, "scale": 2.75, "width": 392.72727272727275}
 (NOBRIDGE) LOG  Screen: {"fontScale": 1, "height": 829.0909
090909091, "scale": 2.75, "width": 392.72727272727275}
```



Використання **Dimensions** для адаптивних стилів

```
import React from 'react';
import { Dimensions } from 'react-native';
import styled from 'styled-components/native';

const { width : number } = Dimensions.get('window');

const Container = styled.View`
    width: ${width > 100 ? '30%' : '100%'};
    padding: 20px;
    background-color: #154d85;
`;

Show usages  kipz_nsi *
export default function App() {
  return <Container/>;
}
```



CSS-хелпер для перевикористання фрагментів

Бібліотека надає функцію css, яка дозволяє виносити цілі блоки стилів у змінні.

```
import { css } from 'styled-components/native';

const commonShadow = css`
  shadow-color: #000;
  shadow-opacity: 0.2;
  elevation: 5;
`;

const Card = styled.View`
  background-color: white;
  ${commonShadow}
`;
```

csstox

https://github.com/styled-components/css-to-react-native?tab=readme-ov-file

Utility-First стилізація

Utility-First — це підхід до стилізації інтерфейсу, за якого зовнішній вигляд компонентів формується за допомогою невеликих утилітарних CSS-класів, кожен з яких відповідає за одну конкретну властивість стилю (відступ, колір, розмір шрифту, позиціонування тощо).

На відміну від традиційного підходу, де створюються окремі CSS-класи з описом стилів, у Utility-First стилі оформлення задається безпосередньо в розмітці.

NativeWind

NativeWind — це бібліотека для React Native, яка реалізує utility-first підхід стилізації, дозволяючи використовувати синтаксис Tailwind CSS через властивість `className`.

```
<View className="flex-1 items-center justify-center bg-blue-500">
  <Text className="text-white text-lg">
    Hello NativeWind
  </Text>
</View>
```

Особливості роботи **NativeWind**

NativeWind не є класичною CSS-бібліотекою та працює інакше, ніж Tailwind CSS у веброзробці:

- **Не додає CSS у React Native** — оскільки середовище React Native не підтримує CSS.
- **Не використовує браузерний rendering engine** — стилі застосовуються без DOM та браузерного рушія.
- **Перетворює utility-класи у нативні стилі** — класи `className` компілюються у JavaScript-об’єкти стилів, сумісні з нативною системою стилізації React Native.

Фактично NativeWind виступає шаром трансляції між синтаксисом Tailwind CSS і нативними стилями платформи.

1. Встановлення NativeWind

Інсталяція бібліотек

npm install nativewind react-native-reanimated react-native-safe-area-context
npm install --dev tailwindcss prettier-plugin-tailwindcss babel-preset-expo

Основні залежності:

- **nativewind** — utility-first стилізація
- **tailwindcss** — система утиліт
- **reanimated** — анімації
- **safe-area-context** — робота з safe zones пристроїв

Installation

nativewind

4.2.2 • Public • Published a day ago

Readme | Code Beta | 3 Dependencies | 190 Dependents | 488 Versions

Nativewind

About

Do you like using Tailwind CSS to style your apps? This helps you do that in React Native. Nativewind is **not** a component library, it's a styling library. If you're looking for component libraries that support Nativewind, see below.

Nativewind makes sure you're using the best style engine for any given platform (e.g. CSS StyleSheet or StyleSheet.create). Its goals are to to provide a consistent styling experience across all platforms, improving developer UX, component performance, and code maintainability.

Nativewind processes your styles during your application's build step and uses a minimal runtime to selectively apply reactive styles (eg changes to device orientation, light dark mode).

Install

```
> npm i nativewind
```

Repository

github.com/marklawlor/nativewind

Homepage

nativewind.dev

Weekly Downloads

385 720

Version	License
4.2.2	MIT

Unpacked Size	Total Files
419 kB	146

1.1. Ініціалізація **Tailwind** конфігурації

Створення конфігурації Tailwind:

npx tailwindcss init

Створюється файл:

tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: ["./App.js", "./src/**/*.{js,jsx,ts,tsx}"],
  presets: [require("nativewind/preset")],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

1.2. Створення global.css

Створіть файл:

global.css

Вміст:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Цей файл використовується NativeWind для генерації стилів.

1.3. Налаштування Babel

Створити файл:

babel.config.js

Роль Babel:

- знаходить `className`
- трансформує utility-класи
- генерує native styles під час збірки

```
module.exports = function (api) {
  api.cache(true);
  return {
    presets: [
      ["babel-preset-expo", { jsxImportSource: "nativewind" }],
      "nativewind/babel",
    ],
  };
};
```

1.4. Конфігурація Metro

Створити файл:

`metro.config.js`

Metro bundler:

- обробляє Tailwind стилі
- інтегрує NativeWind у pipeline збірки

```
const { getDefaultConfig } = require("expo/metro-config");
const { withNativeWind } = require('nativewind/metro');

const config = getDefaultConfig(__dirname)

module.exports = withNativeWind(config, { input: './global.css' })
```

1.5. Підключення CSS

Імпорт стилів:

import "./global.css";

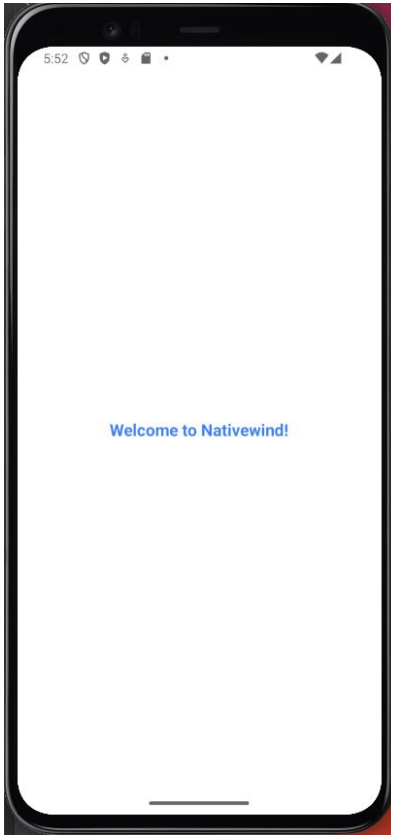
Додається у `App.jsx`.

Приклад використання

Після конфігурації виконати:

npx expo start -c

```
import "./global.css"
import { Text, View } from "react-native";

export default function App() {
  return (
    <View className="flex-1 items-center justify-center bg-white">
      <Text className="text-xl font-bold text-blue-500">
        Welcome to Nativewind!
      </Text>
    </View>
  );
}
```



Button з варіантами **(NativeWind)**

Writing Custom Components

```tsx
import { Pressable, Text } from "react-native";

export default function Button({
  title,
  variant = "primary",
}) {
  const styles = {
    primary: "bg-blue-500 text-white",
    secondary: "bg-gray-200 text-black",
  };

  return (
    <Pressable className={`px-4 py-2 rounded ${styles[variant]}`}>
      <Text className="font-bold">{title}</Text>
    </Pressable>
  );
}
```

```tsx
<Button title="Primary" variant="primary" />
<Button title="Secondary" variant="secondary" />
```

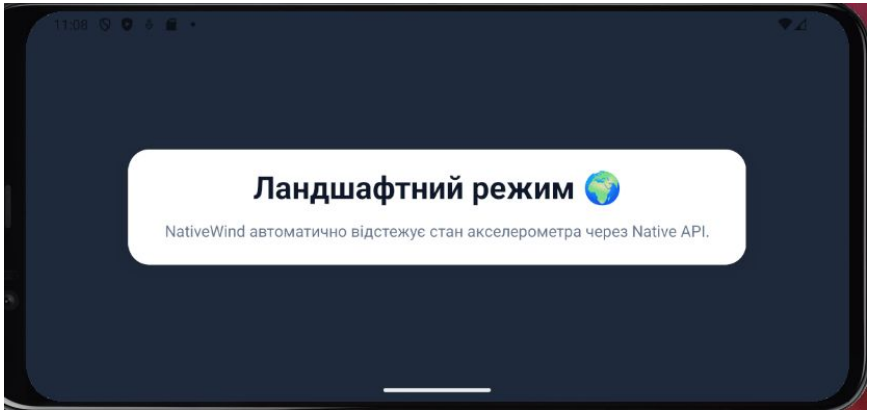
Layout utilities (Flexbox)

React Native використовує **Flexbox** як основну модель розташування елементів. NativeWind надає готові utilities для Flexbox.

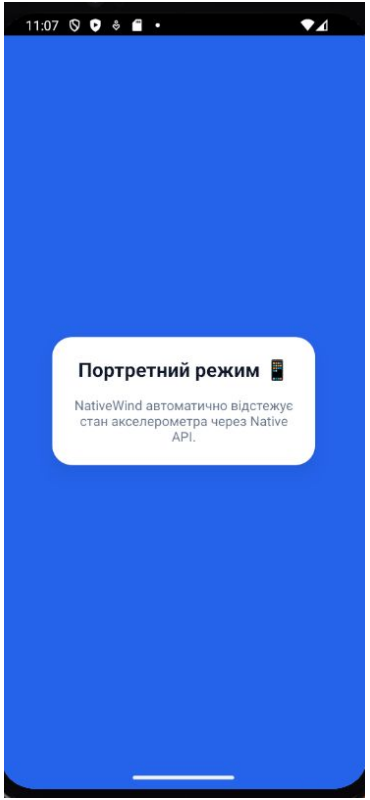
Flex

Orientation

NativeWind автоматично підписується на зміну орієнтації пристрою через React Native API та миттєво перемикає класи без використання JS-хуків у кожному компоненті.




```
"slug": "styled_components",
"version": "1.0.0",
"orientation": "default",
"icon": "./assets/icon.png",
"userInterfaceStyle": "light",
```



Safe Area utilities

NativeWind інтегрується з `react-native-safe-area-context`, щоб автоматично додавати відступи там, де це необхідно.

Основні класи:

- **`pt-safe`** — додає верхній відступ (Padding Top).
- **`pb-safe`** — додає нижній відступ (Padding Bottom).
- **`min-h-safe`** — встановлює мінімальну висоту з урахуванням усіх безпечних зон.

```tsx
export default function App() {
  return (
    <View className="flex-1 bg-slate-900 pt-safe pb-safe">
      <View className="flex-1 bg-white rounded-t-3xl p-6">
        <Text className="text-2xl font-bold text-slate-900">
          Контент
        </Text>
      </View>
    </View>
  );
}
```

Pressable state integration

У стандартному React Native для зміни стилю при натисканні (active state) потрібно використовувати функцію всередині пропса `style`. NativeWind дозволяє робити це за допомогою спеціальних префіксів-станів.

Ключові класи станів:

- **`pressed:`** — стилі, що застосовуються в момент натискання (аналог `:active` у вебі).
- **`hover:`** — актуально для Web та Desktop (наведення курсору).
- **`focus:`** — стан фокусу (наприклад, для текстових полів або кнопок).

Appearance API (system theme)

Appearance API — це вбудований модуль React Native, який дозволяє застосунку визначати **системну тему пристрою**:

- Light mode (світла тема)
- Dark mode (темна тема)

API читає налаштування ОС (Android / iOS) і дозволяє адаптувати UI автоматично.

```
export default function App() {
  return (
    <View className="flex-1 items-center justify-center bg-white dark:bg-slate-900">
      <Text className="text-black dark:text-white text-xl font-bold">
        NativeWind Theme
      </Text>
    </View>
  );
}
```

Native shadow system

NativeWind автоматично генерує:

iOS

- shadowColor
- shadowOpacity
- shadowRadius
- shadowOffset

Android

- elevation

Один клас → кросплатформна тінь.

Conditional class evaluation через **JS runtime**

Conditional Class Evaluation — це механізм NativeWind, який дозволяє **динамічно обчислювати Tailwind-класи під час виконання JavaScript**.

```
export default function App() {
  const [selected, setSelected] = useState(false);

  return (
    <Pressable
      onPress={() => setSelected(!selected)}
      className={`m-9 p-4 rounded-2xl ${
        selected ? "bg-emerald-500" : "bg-slate-400"
      }`}
    >
      <Text className="text-white font-bold text-center">
        Toggle
      </Text>
    </Pressable>
  );
```