{callstack}

Delightful UX with React Native Training

{callstack}

Who we are

Jay

@jayu

@jayu_dev

Maciej

@mcjsmk

@simka

{callstack}

Agenda

13:00

- Start
- Animations
- Gestures

17:00

- Lunch break

18:00

- Gestures
- Theming
- Translations
- Accessibility

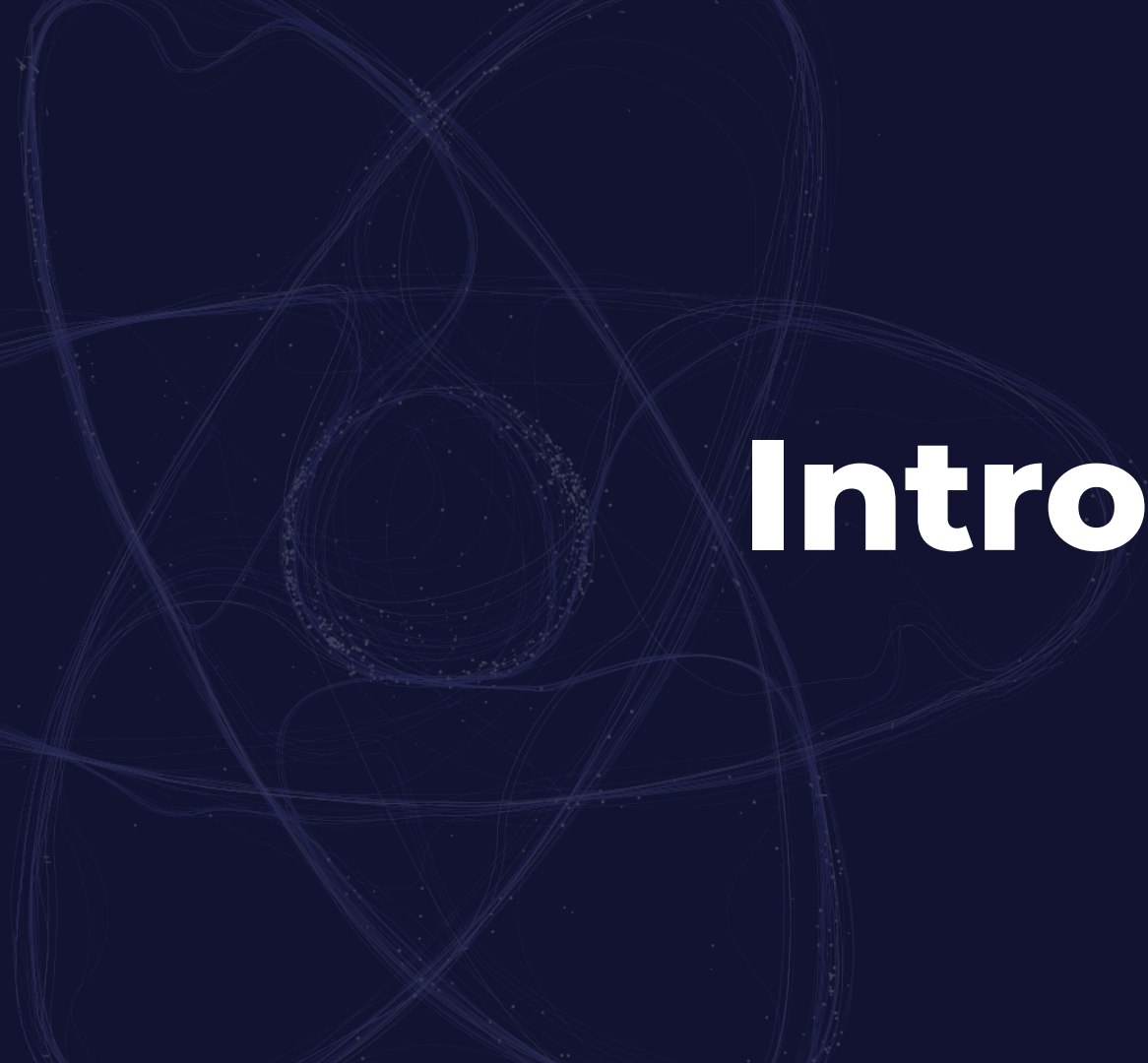
Around 15:00 and 19:30

- Coffee break

{callstack}

Source code

https://github.com/callstack-workshops/pearl-pay-delightful-ux-with-react-native



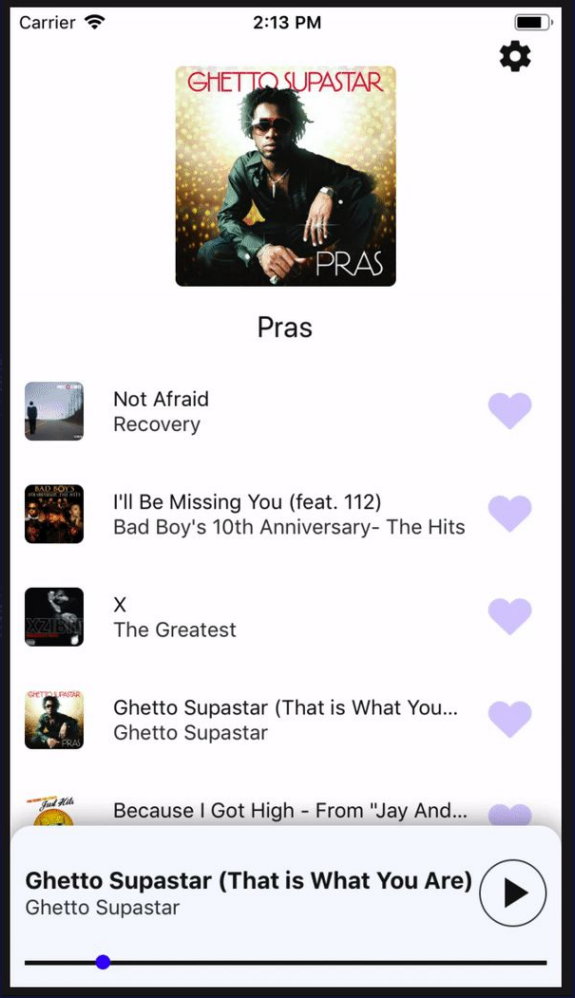
Intro

{callstack}

What are we building?

App

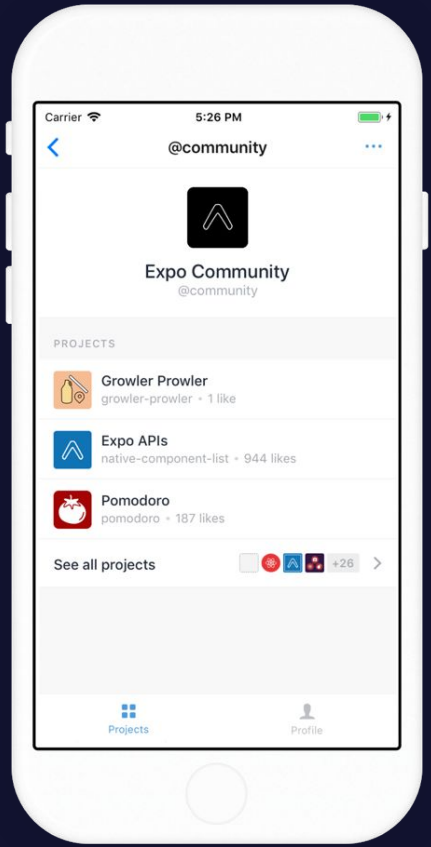
- animated “heart” icon
- animated play/pause button
- collapsible header
- swipeable song item + fancy animation
- dynamic theming for the app
- multi-language support
- accessible login form



{callstack}

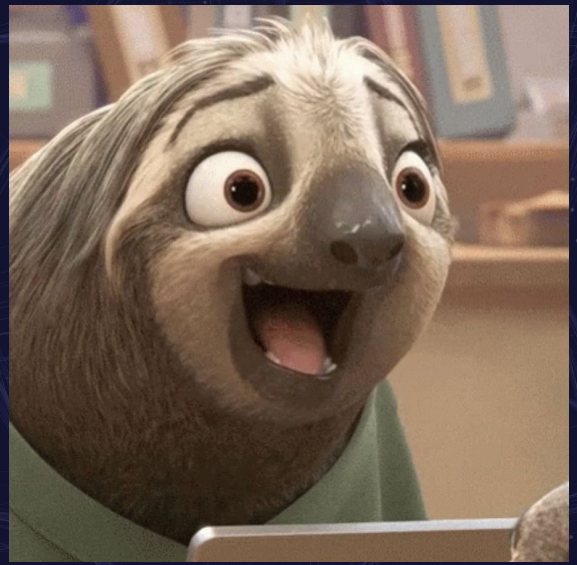
expo

- Install Expo Client app from App Store / Google Play and run many projects
- Already linked native libraries



{callstack}

First - a word about RN

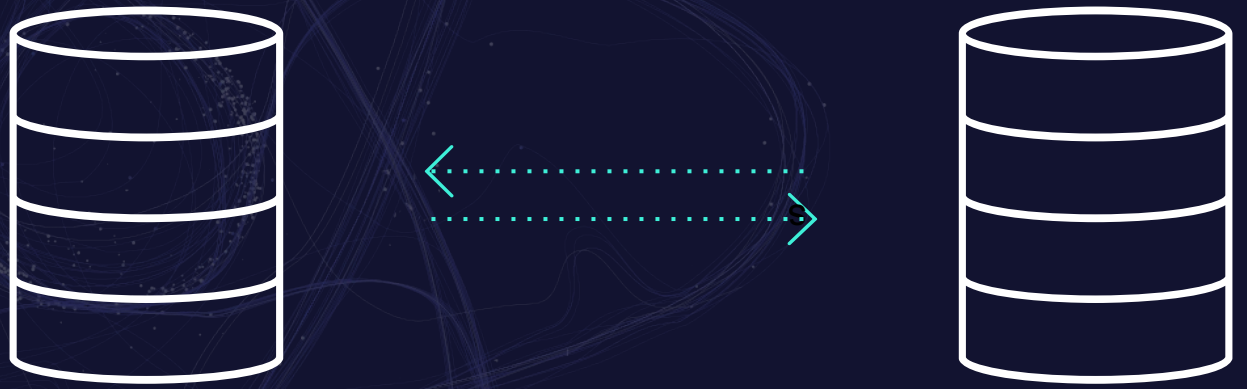


Is easy to learn and multi-platform...

... but lets you use native when needed

{callstack}

Convenience brings a challenge



{callstack}

Advanced animations

{callstack}

Animated vs Reanimated

{callstack}

Reanimated

- Native approach
- More generic functions
- Much more efficient when connected with gestures
- Fully compatible with Animated

Installing

```
expo install react-native-reanimated
```

{callstack}

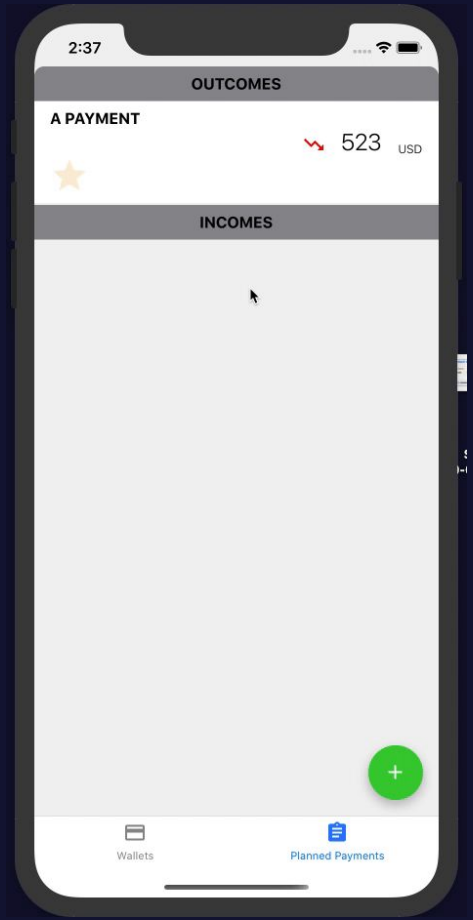
Differences

	Reanimated	Animated
Installation	Needs integration	Works out of the box
Code	Lower level	Higher level
Driver	Native only	Works on JS and native thread
Paradigm	Declarative	Imperative

{callstack}

Simple animation snippet

```
import React, { useState, useEffect, useRef } from 'react'
import { Animated, Easing } from 'react-native'
import Icon from 'react-native-vector-icons/MaterialIcons';

const AnimatedIcon = ({ active }) => {
  const targetValue = active ? 1 : 0.2
  const opacity = useRef(new Animated.Value(targetValue))

  useEffect(() => {

    Animated.timing(opacity.current, {
      toValue: targetValue,
      easing: Easing.inOut(Easing.ease),
      useNativeDriver: true,
      duration: 500
    })
    .start()

  }, [active])

  return (
    <Animated.View style={{ opacity: opacity.current }}>
      <Icon name="grade" size={40} color={"#FAAE3E"} />
    </Animated.View>
  )
}
```



Heart animation

{callstack}

Example

On pressing heart icon we want to:

- Animate its opacity
- 0 if not selected, 1 if selected

Animated values

- a value that can be changed in time
- no rerendering
- calculated in native
- No way to get its value directly

{callstack}

Animated values

Importing

```
import Animated from 'react-native-reanimated';
const { Value } = Animated;
```

Using

```
class FavouriteIcon extends React.Component {
  ...
  progress = new Value(this.props.checked ? 1 : 0.2);
  ...
}
```

Composing reanimated nodes

How to change value over time in Reanimated? Describe it declaratively using **nodes**

- ***cond****(condition, resultIfTrue, resultIfFalse)*
  Classic condition (like if)
- ***set****(animatedValue, value) - sets animated value*
  Sets animated value
- ***clockRunning****(clock)*
  Checks if clock is running
- ***startClock****(clock)*
  Starts clock

Composing reanimated nodes

Importing

```
import Animated from 'react-native-reanimated';
const { cond } = Animated;
```

Using

```
class FavouriteIcon extends React.Component {
  ...
  color = block([cond(/* description of our animation*/)])
  ...
}
```

What is *clock*

- Special type of ***Animated.Value***
- Updates in each frame to the timestamp of the current frame
- Needed for animations - if clock is running means animation should be proceeded

Using proper animated nodes we can

- Start clock - *startClock(clock)*
- Stop clock - *stopClock(clock)*
- Check if its running - *clockRunning(clock)*

First animation - *timing*

- Updates the position of node by running timing-based animation from a given position to a destination (***toValue***).
- Is expected to last ***duration*** milliseconds

Usage

```
timing(
  clock,
  { finished, position, frameTime, time },
  { toValue, duration, easing }
);
```

First animation - *timing*

- ***clock*** - clock
- ***finished*** - indicates if the animation is over
- ***position*** - current animation value
- ***frameTime*** - progress of animation in milliseconds from animation start
- **time** - current time
- **toValue** - value we want to have after animation finishes
- **duration** - duration of the animation
- **easing** - easing function

Usage

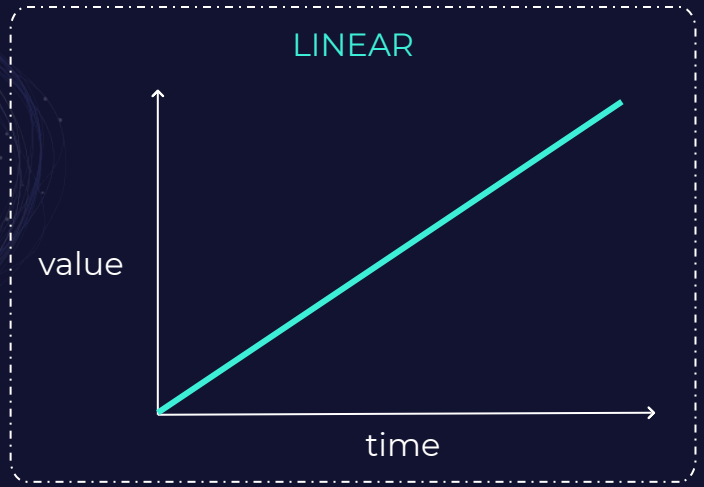
```
timing(
  clock,
  { finished, position, frameTime, time },
  { toValue, duration, easing }
);
```

{callstack}

Easing functions

Specifies how the value should change over time.

- Most basic one is ***Easing.Linear***



Wanna play more?
https://easings.net/

Joining the dots

Let’s add animation to ***FavouriteIcon*** component

- We need ***opacity*** animation value - can be a class property
- We are going to change **opacity** from ***0,2*** (not selected) to ***1*** (selected)

runLinearTiming helper function

- Should return **Animated block**
- Returned block returns value from ***0*** to ***toValue***
- Takes clock value as an argument

```
function runLinearTiming(clock, toValue, duration = 200) {
  // some variables
  // ...
  return block([
    // reanimated block
    // ...
  ]);
}
```

{callstack}

What is an Animated block?

- Array of nodes
- Evaluated in a particular order
- Returns the value of the last node

```
block([
  // reanimated block
  // ...
]);
```

timing state and config

- We only need ***position*** state value, the rest can be an temporary animated values
- toValue, duration and initial position values should be given to function as an arguments
- As easing function let's use ***Easing.linear***

```
const state = {
  finished: new Value(0),
  frameTime: new Value(0),
  position: position,
  time: new Value(0),
};

const config = {
  toValue,
  duration,
  easing: Easing.linear,
};
```

Playing with *clock*

We need to make sure clock value will be properly set.

- ***clock*** equal ***1*** - animation running
- ***clock*** equal ***0*** - animation stopped

To stop animation we need to check if ***state.finished*** is ***1***. If yes we can stop clock

- ***startClock*** - starting clock
- ***stopClock*** - stopping clock
- ***clockRunning*** - checking if clock is running

Final animated block

- (1) Main block
- (2) Check if clock is running
  - (3) Yes - do nothing
  - (4) No - reset all values
    (5) and start clock
- (6) Start animation - *timing*
- (7) Check if animation is finished
  - (8) Yes - stop clock
- (9) Return *state.position*

```
return block([ /* 1 */
  cond(clockRunning(clock) /* 2 */, 0 /* 3 */, [
    set(state.finished, 0), /* 4 */
    // reset all animation values
    // to default one from initalization
    // ...
    startClock(clock), /* 5 */
  ]),
  timing(clock, state, config), /* 6 */
  cond(
    state.finished /* 7 */,
    stopClock(clock) /* 8 */
  ),
  state.position, /* 9 */
]);
```

Changing opacity

- Use ***opacity*** style attribute
- ***Animated.View*** instead of ***View***

```
style={{
  opacity: this.opacity,
}}
```

```
<Animated.View
  style={{
    /* styles with animation */
  }}
>
  <Image
    /* image props*/
  />
</Animated.View>
```

Final touches

- Use ***runLinearTiming*** in component
- Check in ***componentDidUpdate*** if *checked* prop changed
- If yes - trigger animation by setting ***toValue***
- Use evaluated opacity value in style opacity property

```
class FavouriteIcon extends React.Component {
  clock = new Clock();
  toValue = new Value(0.2);
  opacity = runLinearTiming({
    clock: this.clock,
    toValue: this.toValue,
    position: new Value(0.2),
  });

  componentDidUpdate(prevProps) {
    if (prevProps.checked !== this.props.checked) {
      this.toValue.setValue(this.props.checked ? 1 : 0.2);
    }
  }

  /* ... */
}
```

Exercise #1

Play/Pause button animation

Example

On pressing Play/Pause icon we want to combine:

- Opacity
- Rotation

What we need

- Wrapper with two animated views
- Animated views with icons inside

On icon press we want to

- Rotate wrapper
- Change opacity of the first icon to ***0***
- Change opacity of the second icon to ***1***

```
<TouchableWithoutFeedback onPress={this.props.onPress}>
  <View style={computedStyles.container}> {/* rotation */}
    <Animated.View>
      <Animated.View> {/* opacity */}
        <Ionicons /> {/* pause icon */}
      </Animated.View>

      <Animated.View> {/* opacity */}
        <Ionicons /> {/* play icon */}
      </Animated.View>
    </Animated.View>
  </View>
</TouchableWithoutFeedback>
```

Needed animated values

- Play icon ***opacity***
  1 ---> 0
- Pause icon ***opacity***
  0 ---> 1
- Wrapper ***rotation***
  0deg ---> 180deg

All animations could be linear so we can use ***runLinearTiming***

{callstack}

Interpolation

- Run timing only once
- Map created value into another one - interpolation
- One animation multiple animated values

interpolate node

- ***inputRange*** - values we want to map **from**
- ***outputRange*** - values we want to map **to**

Importing

```
import Animated from 'react-native-reanimated';
const { interpolate } = Animated;
```

Using

```
interpolatedValue = interpolate(this.value, {
  inputRange: [0, 100],
  outputRange: [1, 0],
});
```

Joining the dots

Let’s add an animation to ***PlayPauseButton*** component

- ***playOpacity*** value - for which we use ***runLinearTiming***
- Interpolate two other values - ***pauseOpacity*** and ***rotation***

Triggering the animation

We need to keep information about playing state in ***Player*** component.

- Create ***playingState*** animated value
- Create ***handlePlayToggle*** function
- Call it on ***PlayPauseButton*** press
- Pass the value set in ***handlePlayToggle*** to ***PlayPauseButton*** - use it as ***toValue*** there

```
handlePlayToggle = () => {
  this.playingState.setValue(cond(eq(this.playingState, 0), 1, 0));
};
```

Rotation

- ***rotateY*** attribute

- We have numeric value but it needs ***deg*** suffix
- We can use ***concat*** for that - it's a part of Reanimated

```
style={{
  transform: [
    {
      rotateY: '180deg',
    },
  ],
}}
```

```
concat(this.rotation, 'deg')
```

Exercise #2

{callstack}

Flawless gestures

{callstack}

Collapsible header

{callstack}

What am I looking at?

List headers often contains the information about the list.

It’s presence is important but they often

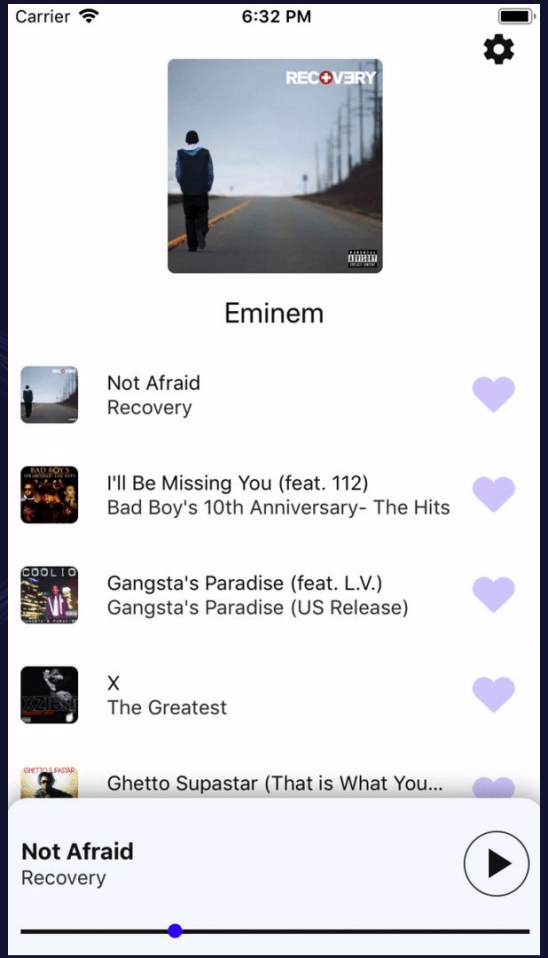
- Covers list content
- Are not visible after list is scrolled down

Let’s make them collapsible!

Example

During scrolling we want to

- Animate big header by
  - Moving it
  - Scaling
  - Fading out
- Animate smaller header to show it
  - Changing the opacity from 0 to 1



{callstack}

{callstack}

Animation details

We need to animate in parallel

- *(CollapsibleHeader)* Fading out song cover and a title below
- *(CollapsibleHeader)* Scaling down the component
- *(CollapsibleHeader)* Moving component down a bit
- *(HeaderTitle)* Showing smaller header using it's opacity

onScroll event

We need scroll-controlled animation

- ***onScroll*** - gives us a possibility to listen to onScroll events
- It's built-in into ***FlatList***
- Reanimated has function that helps us with mapping those events to animated values - ***event***
- ***scrollY*** animated value will become the current value of the Y offset

```
onScroll={event([
  {
    nativeEvent: {
      contentOffset: {
        y: scrollY,
      },
    },
  },
])}
```

Animated Flat List

I lied... a little

- To use *onScroll* we need Animated *FlatList*
- We can create it using ***Animated.createAnimatedComponent()***

```
const AnimatedFlatList = Animated.createAnimatedComponent(FlatList);
```

Animations

- ***scrollY*** from onScroll event - our base animation value
- Use ***interpolate*** to create values we need from it

BUT

For most of the animations we need values within some specific and finite range but our value from event can be between ***0*** and ***something***.

{callstack}

For example with opacity

For ***opacity*** we want to interpolate ***scrollY*** value within range 0 - something into range 0 -1

- **start** animating header after ***scrollY*** = ***50***
- **stop** animating header after ***scrollY*** = ***100***

So we need:

Scroll Y	Opacity
<= 50	0
60	0.2
80	0.8
>= 100	1

Extrapolate for the rescue!

- Interpolate takes another configuration parameter - ***extrapolate***
- We can specify how to estimate values beyond the original observation range
- For our case perfect option is ***Extrapolate.CLAMP***

Code example

Importing

```
import Animated from 'react-native-reanimated';
const { interpolate, Extrapolate } = Animated;
```

Using

```
opacity = interpolate(this.props.scrollY, {
  inputRange: [50, 100],
  outputRange: [0, 1],
  extrapolate: Extrapolate.CLAMP,
});
```

Joining the dots

- Use ***onScroll*** to get ***scrollY*** animated value
- Pass it to ***CollapsibleHeader*** and ***HeaderTitle*** components
- In ***CollapsibleHeader***:
  - Interpolate it to have ***translateY*** value
  - Interpolate it to have ***opacity*** value
  - Interpolate it to have ***scale*** value
- In ***HeaderTitle***:
  - Interpolate it to have ***titleOpacity*** value - we talked about it few seconds earlier
- Use interpolated values as style attributes
- Remember to use ***Animated.Views***

Exercise #3

{callstack}

Gesture Handler

{callstack}

RN Gesture Handler

- Exposes platform native touch and gesture system
- No callback after gesture
- More generic
- Smooth even on busy JS thread
- Declarative

Installing

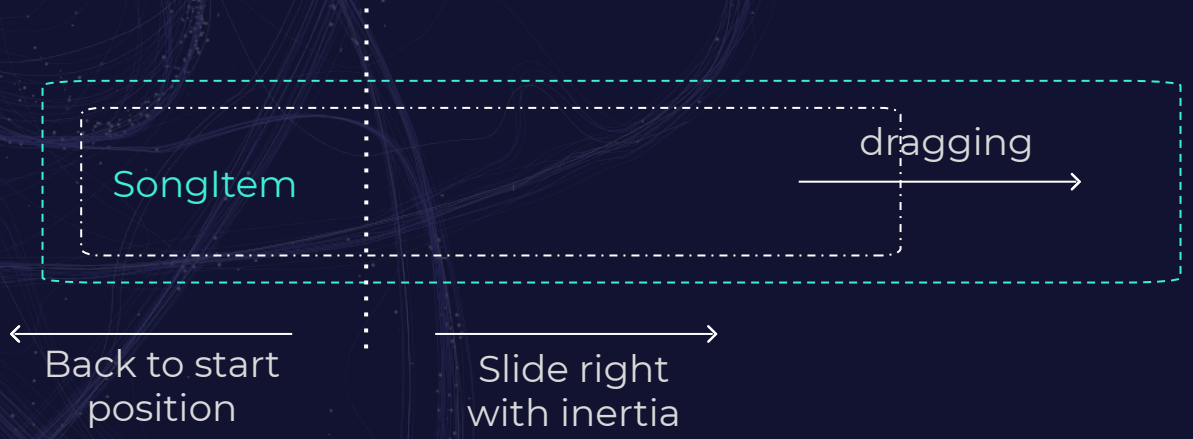
```
expo install react-native-gesture-handler
```

{callstack}

Swipeable Row

Swipe gesture

- Standard these days
- SongItem should follow our finger
- Finger is raised - start particular animation



Swipe gesture - example

PanGestureHandler

- Recognizes panning - dragging
- Activates when finger is on screen and moves by some distance
- Has the same API as ***onScroll*** from ***AnimatedFlatList***
- Contains more useful attributes like for example ***translationX***, ***velocityX***, ***state***

```
event([
  {
    nativeEvent: {
      translationX: dragX,
      velocityX: dragVX,
      state: this.gestureState
    }
  }
]);
```

PanGestureHandler

- ***dragX*** - current gesture position
- ***dragVX*** - gesture velocity
- ***this.gestureState*** - current gesture state

```
event([
  {
    nativeEvent: {
      translationX: dragX,
      velocityX: dragVX,
      state: this.gestureState
    }
  }
]);
```

PanGestureHandler - props

- ***onGestureEvent*** - event we just talked about
- ***onHandlerStateChange*** - event triggers on gesture state change
- ***maxPointers*** - number of pointers can be engaged into gesture
- ***activeOffsetX*** - range along the X axis (in points) where fingers travel without activation of the handler

```
<PanGestureHandler
  onGestureEvent={this.onGestureEvent}
  onHandlerStateChange={this.onGestureEvent}
  maxPointers={1}
  activeOffsetX={10}
>
  {/* ... */}
</PanGestureHandler>
```

activeOffsetX

- Stops accidental gesture triggering

  [0, 10] - only dragging more than 10 points to the right will trigger event

- Blocks dragging in some direction

  10 - blocks dragging right.
  It's like ***[-infinty, 10]***

```
event([
  {
    nativeEvent: {
      translationX: dragX,
      velocityX: dragVX,
      state: this.gestureState
    }
  }
]);
```

{callstack}

Exercise time!

Try to implement dragging only

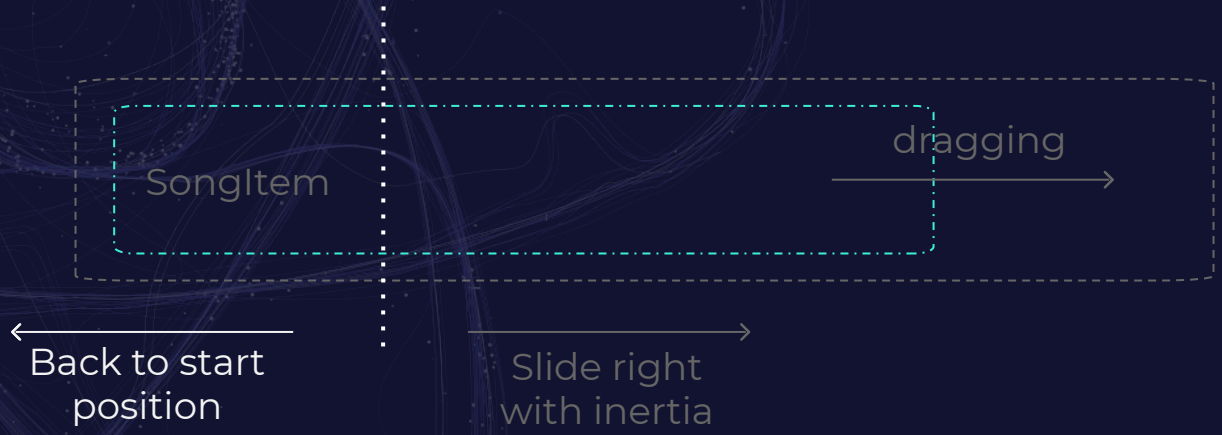
- Wrap content of ***TouchableOpacity*** into ***PanGestureHandler***
- Create an **event** node with just **translationX** mapping
- Use the same event in ***onGestureEvent*** and ***onHandlerStateChange*** props
- Use the value from it in ***translateX*** property of wrapped ***View***
- Allow only swipe right

Exercise #4

{callstack}

Coming back to start

Gesture is finished - animate ***SongItem*** to its start position



Coming back to start

- ***translateX*** - animated value with drag position during dragging and animation value when dragging is finished
- ***gestureState*** will contain value ***State.ACTIVE*** as long as user keeps swiping
- Use ***eq*** node to compare values

```
this.translateX = cond(
  eq(this.gestureState, State.ACTIVE),
  [
    /* if gesture is in active state */
  ],
  [
    /* if gesture is NOT in active state */
  ]
);
```

Coming back to start

- **ACTIVE** - in the middle of the drag gesture
  - Stop all clocks
  - Return ***dragX*** value

- **not ACTIVE**
  - Trigger spring animation

```
this.translateX = cond(
  eq(this.gestureState, State.ACTIVE),
  [
    /* if gesture is in active state */
  ],
  [
    /* if gesture is NOT in active state */
  ]
);
```

runSpring - helper function

- Similar to ***runLinearTiming***
- Should look natural
- With proper velocity
- With bounce at the end
- Uses ***spring*** node

```
export function runSpring(clock, position) {
  const state = {  /* state */ };

  const config = { /* config */ };

  return [ /* animated block */ ];
}
```

spring node - helper function

Tracks the velocity state to create fluid motions as the ***toValue*** updates

- ***velocity*** - velocity of the object attached to animation. It will be the velocity from ***PanGestureHandler*** event
- ***stiffness*** - spring stiffness coefficient
- ***damping*** - defines how the spring's motion should be damped

```
spring(
  clock,
  { finished, position, velocity, time },
  {
    damping,
    mass,
    stiffness,
    overshootClamping,
    restSpeedThreshold,
    restDisplacementThreshold,
    toValue
  }
);
```

spring node - default config

Let's use default config - it will create ideal animation for our purposes

SpringUtils.makeDefaultConfig()

Importing

```
import Animated, { Easing } from "react-native-reanimated";
const { SpringUtils } = Animated;
```

Using

```
{
  stiffness: new Value(100),
  mass: new Value(1),
  damping: new Value(10),
  overshootClamping: false,
  restSpeedThreshold: 0.001,
  restDisplacementThreshold: 0.001,
  toValue: new Value(0),
}
```

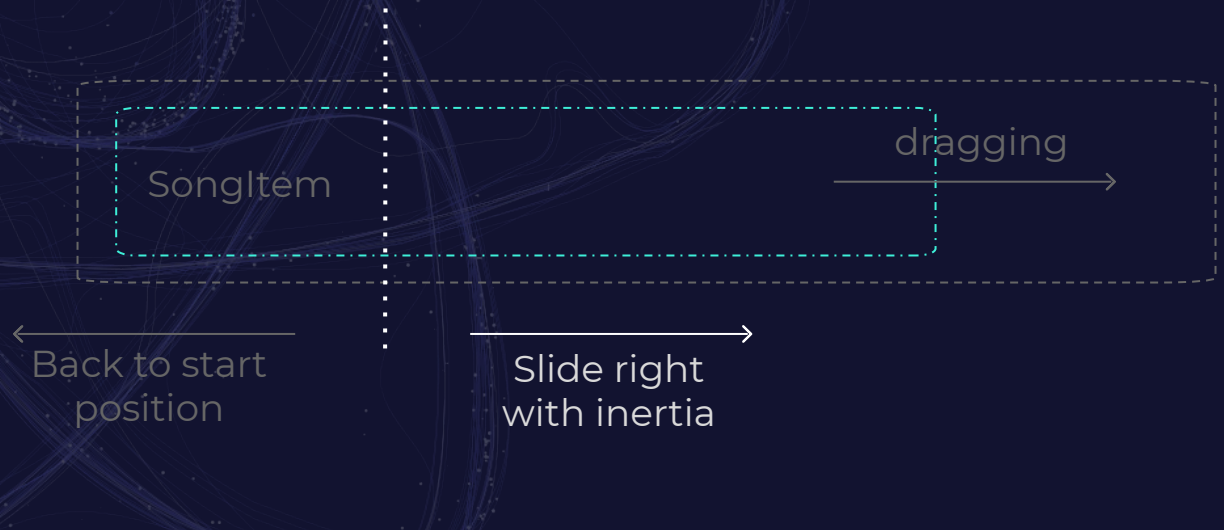
Wanna play more?
https://facebook.github.io/react-native/docs/animated#spring

Exercise #5

{callstack}

Swipe with inertia

- User drags ***SongItem***
- Releases gesture after distance higher than 80
- ***SongItem*** should continue moving with proper inertia and deceleration



decay node

- ***clock***, ***finished***, ***velocity***, ***position***, ***time***
- ***deceleration*** - controls how fast the animation decelerate
- The best results are with ***deceleration*** around ***0.995***

```
decay(
  clock,
  {
    finished,
    velocity,
    position,
    time
  },
  { deceleration }
);
```

runSwipeDecay - helper function

```
export function runSwipeDecay(clock, position, velocity) {
  const state = {
    /* state - lets use position and velocity here */
  };

  const config = { /* config */ };

  return [
    /* animated block */;
  ];
}
```

{callstack}

Animating height and opacity

Let's hide the song!

- Animate ***SongItem*** height
- Animate ***SongItem*** opacity

- ***runLinearTiming*** - will be perfect for our case
- Animate height from ***ROW_HEIGHT*** to ***0***
- Interpolate opacity based on height

Joining the dots

- ***runSwipeDecay*** - function for animating swipe with inertia
- ***runSpring*** - function for animating coming back to start position
- Animation of height and opacity using ***runLinearTiming***
- 3 clocks - one for each animation
- ***dragX < 80*** - run spring animation
- ***dragX >=80*** - run decay animation
- Put everything into ***translateX*** property

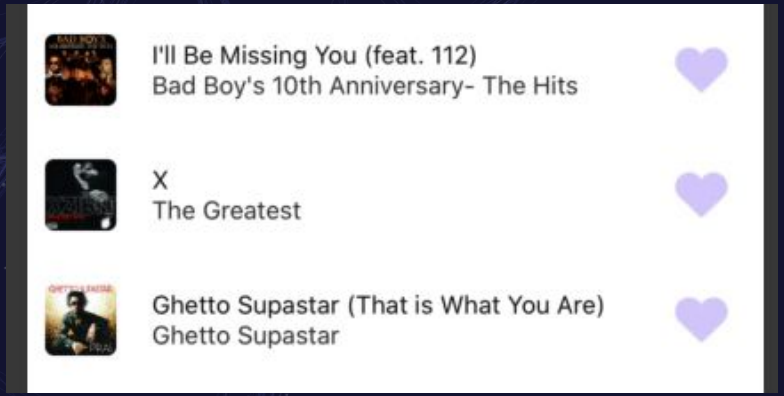
{callstack}

translateX

```
this.translateX = cond(
  eq(this.gestureState, State.ACTIVE),
  [
    /* stopping clocks and returning dragX value */
  ],
  [
    cond(
      greaterThan(dragX, 80),
      [
        /* running linear timing for animation opacity and height */
        /* running runSwipeDecay for animation item position and returning the value */
      ]
      /* running runSpring for animation item position and returning the value */
    )
  ]
);
```

{callstack}

Exercise #6



Timing callback

Let's remove the song, **for real!**

- When? After height animation is over
- Where? Finish condition in ***runLinerTiming***
- How? ***call*** reanimated node
  - Check if it's the right time using ***state.finished***
    - Use ***onSongRemove*** from props

Where?

```
return block([
  cond(clockRunning(clock), 0, [
    /* reset values, start clock */
  ]),
  timing(clock, state, config),
  cond(state.finished, [stopClock(clock) /* HERE */]),
  state.position
]);
```

How?

```
call([state.finished], callback);
```

Exercise #7

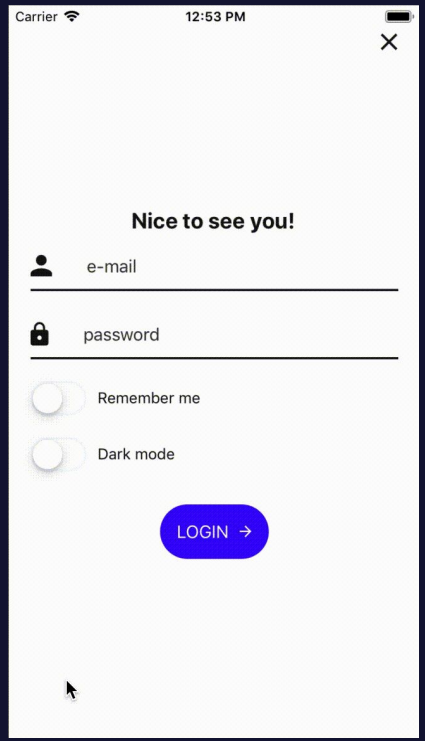

{callstack}
Theming

Color palette - brand VS needs

{callstack}

Example - theme provider

Dynamically toggle between themes:
- easy to add new styles for you
- good for users' needs



What we need

1. Theme provider library
2. Theme objects
   (use `src/utils/theming.js`)

Installing

```
yarn add @callstack/react-theme-provider
```

```
const darkTheme = {
  name: "dark",
  backgroundColor: "#131313",
  secondaryBackgroundColor: "#0c0c0c",
  primaryTextColor: "#FFF",
  secondaryTextColor: "#757575",
  accentColor: "#F8F32B"
};
```

{callstack}

Make it alive!

react-theme-provider exposes method ***createTheming***

themeObject =>
{ ThemeProvider, withTheme }

We can use ***ThemeProvider*** to wrap our parent component:

```
import { createTheming } from "@callstack/react-theme-provider";

const darkTheme = { /* */ }

const { ThemeProvider, withTheme } = createTheming(darkTheme);
```

```
/* */

import { ThemeProvider } from "./src/utils/theming";

const App = () => (
  <ThemeProvider>
    <Home />
  </ThemeProvider>
);

export default App;
```

Final usage

Since then all Home children can use ***withTheme*** HOC!

- Use it to wrap exported component
- Access new ***theme*** prop
- Use theme keys as variables in styles

```
import { withTheme } from "../utils/theming";

const FancyText = (props) => (
    <Text
        style={{ color: props.theme.primaryTextColor }}
    >
        I have changing color!
    </Text>
)

export default withTheme(FancyText)
```

{callstack}

Exercise

Let's make our app dynamically colored!

1. Install ***@callstack/react-theme-provider***
2. Import the library into the file with already created color palettes
3. Create ***ThemeProvider*** and ***withTheme*** HOC
4. Add ***ThemeProvider*** to the entry point of the app
5. Change colors to dynamic in the chosen file

Exercise #8

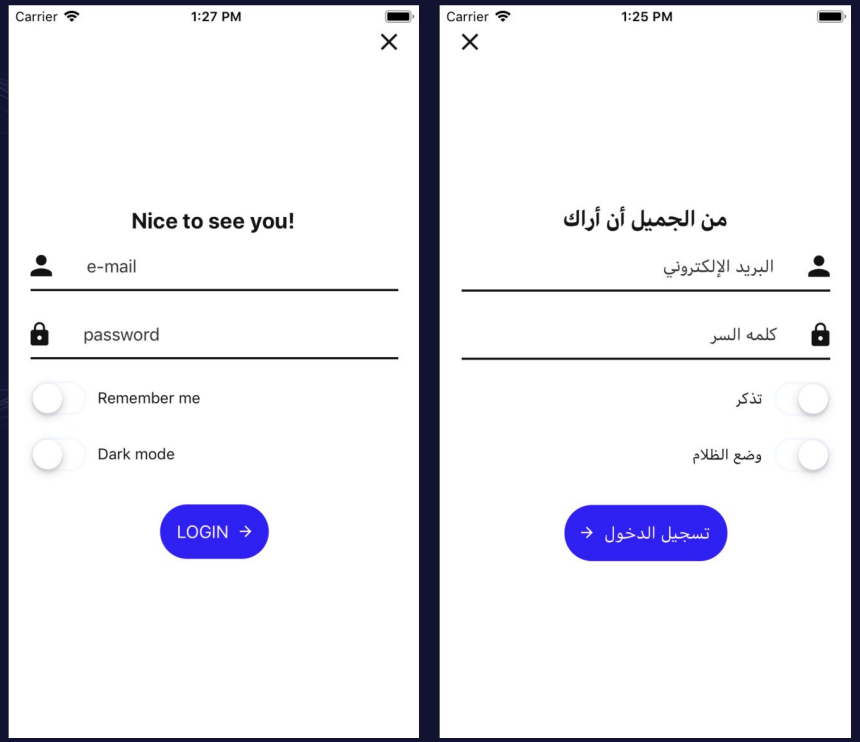
{callstack}

Internationalization

{callstack}

Example - internationalization

I18n fills another users' need - delivering content they understand. Also layout comfortable for them!



{callstack}

What we need

Way i18n works is very similar to providing a theme! We'll use:

1. I18n library
2. Localization library
3. Translation objects (use *'src/utils/translations.js'*)

```
yarn add i18n-js
expo install expo-localization
```

```
export const en = {
  header: "Nice to see you!",
  email: "e-mail",
  /* */
  login_hint: "Press to login"
};
```

Imports and initial setup

1. Import needed files / libs

```
import * as Localization from "expo-localization";
import i18n from "i18n-js";

import { en, pl } from "../utils/translations";
```

2. Tell ***i18n*** what translation it should use (fetch current locale)

```
const currentLocale = Localization.locale;
i18n.locale = currentLocale;
```

3. Assign translations to locales

```
i18n.translations = {
  en: en, // en is the object we imported before
  pl: pl
};
```

Setup - fallbacks

Sometimes it can happen user will have locale not listed by you. That's why we should setup default language.

1. Allow fallbacks
2. Specify default locale

```
i18n.fallbacks = true;

i18n.defaultLocale = "en-GB";
```

Finally - usage!

i18n contains ***t*** method:

translation_key => string

That's all!

```
<Text>{i18n.t("email_hint")}</Text>
```

{callstack}

Let’s do it

- import ***i18n-js*** and ***expo-localization*** libraries
- import translations we need
- assign translations to locales
- get user's locale
- provide fallback
- use translation strings

Work on ***Login.js*** screen

Bonus: RTL support

RN supports RTL out of the box! We just need to pass props to ***I18nManager.***

Flexbox will do the rest.

But when we need to specify some style:

```
import { I18nManager } from 'react-native';

const currentLocale = Localization.locale;
const isRTL = currentLocale.indexOf('ar') === 0;

I18nManager.allowRTL = isRTL;
I18nManager.forceRTL(isRTL); // to test on LTR device
```

```
<Text
    style={{textAlign: isRTL ? 'right' : 'left'}}
/>
```

Exercise #9

{callstack}

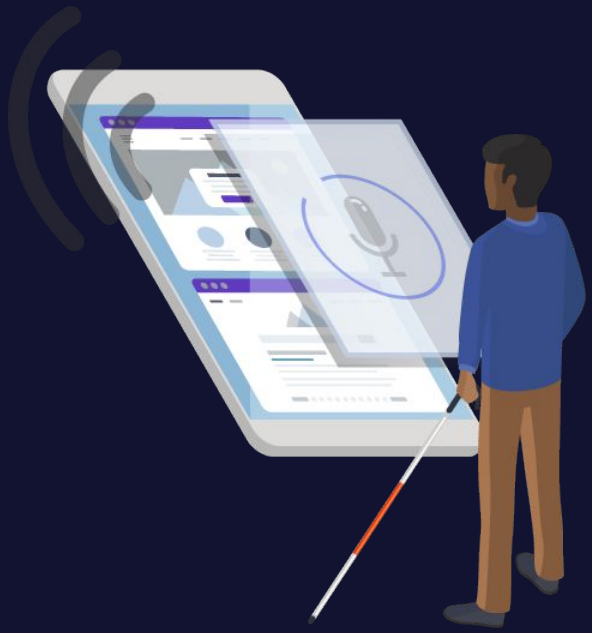
Accessibility

{callstack}

RN vs Accessibility

RN components support accessibility props out of the box!

But in order to test how screen reader will handle our app we need to take few additional steps.



Screen reader config

For the emulator:

- Download apk from ***http://tiny.cc/androidreader***
- Drop the apk to the emulator

On emulator / device go to:

- Settings
- Accessibility
- TalkBack
- Use service

Now no way to test it on iOS emulator.

On device go to:

- Settings
- General
- Accessibility
- Vision
- VoiceOver

Most basic props

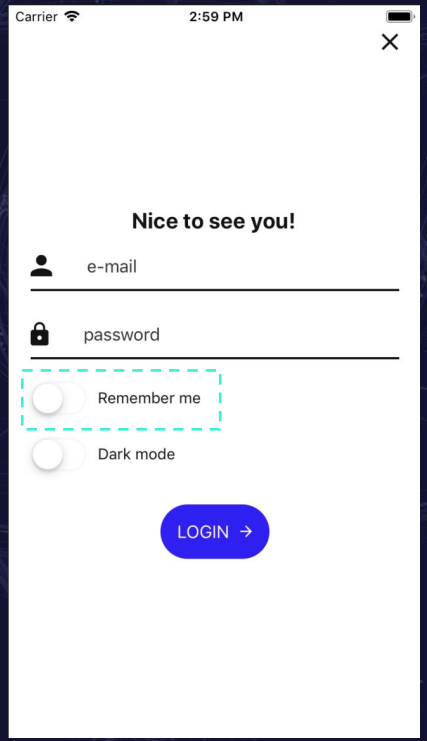
Must-have props are:
- ***accessibilityLabel***
(for short name)
- ***accessibilityHint***
(for more explanation)

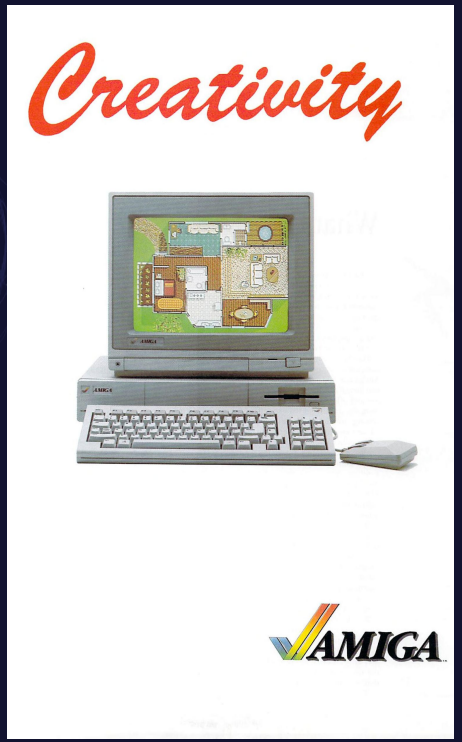
We can connect them with
multi-language setup we already have!

```
<Switch
  accessibilityLabel={i18n.t("remember_me")}
  accessibilityHint={i18n.t("remember_me_hint")}
/>
```

{callstack}

Visual layout vs screen reader





Visual layout vs screen reader

Sometimes you want to hide puzzling elements from screen reader. You can use props:

- ***importantForAccessibility*** for Android
- ***accessibilityElementsHidden*** for iOS

```
<Switch
  accessibilityLabel={i18n.t("remember_me")}
  accessibilityHint={i18n.t("remember_me_hint")}
/>
<Text
    accessibilityElementsHidden={true}
    importantForAccessibility="no"
>
  Remember me
</Text>
```

Visual layout vs screen reader

And sometimes visual representation can be puzzling - e.g. sentences mixed with images.

Using ***accessible*** prop will wrap all elements to one for the screen reader.

Watch out! All ***Touchables*** are ***accessible*** by default.

```
<View
    accessible={true}
    accessibilityLabel="Advert: Amiga boosts creativity"
>
  <Text> Creativity </Text>
  <Image ... />
  <Text> Amiga </Text>
</View>
```

{callstack}

Try it out!

Add proper accessibility props to the ***Login*** screen!

https://facebook.github.io/react-native/docs/accessibility

{callstack}

Exercise #10


{callstack}
Thank you!