Building User Interfaces

React Native 2

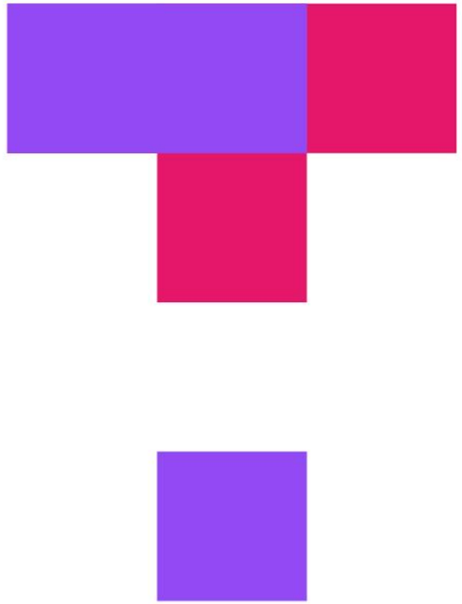
Intermediate Concepts

Professor Bilge Mutlu

What we will learn today?

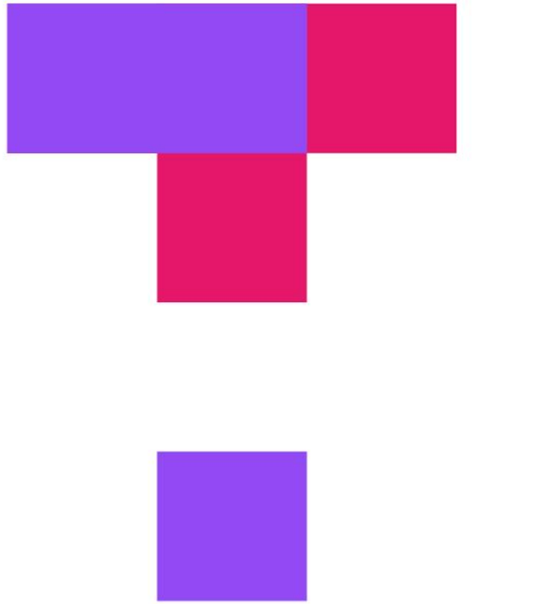
- Mobile Navigation using React Native
- Mobile Input via Gestures using React Native + Animation
- Working with Date object in JS
- Assignment Preview

TopHat Attendance



TOP HAT

TopHat Questions



TOP HAT

Mobile Navigation using React Native

The Options

There are two main ways of implementing navigation in RN:

1. Using ReactJS navigation, i.e., `react-navigation`
2. Using RN navigation, i.e., `react-native-navigation`

more low-level native code

We will be covering `react-navigation` in depth. `react-native-navigation` is for advanced use, as it involves modifying native components, while `react-navigation` is programmed in JS.

Setting up ReactJS

```
npm install react-navigation
npm install react-native-gesture-handler
npm install react-native-reanimated
npm install react-native-screens
```

How does navigation in HTML work?

The History API[1] provides a `Window` object that gives access to a `history` object, which includes a stack of all the pages that the user has previously visited.

When a new link (`<a>`) is pressed, the current URL is pushed to the history stack. The "back" button calls the following function, which pops the previous URL and pushes the current URL.

```
window.history.back()
```

[1] More on the History API

When the "forward" button is pressed, it calls the following function, which pushes the current URL in the stack and pops the previous one.

```
window.history.forward()
```

We can also navigate in the stack and pop a particular URL in the history:

```
window.history.go(3);
```

This will push the current URL to the stack.

How does navigation in RN work?

RN provides a set of *navigators* that accomplish stack-based and other types of navigation:

1. Switch navigator
2. Stack navigator
3. Tab navigator
4. Drawer navigator

Different styles of navigation. Can be combined

Switch Navigator

Definition: Enables showing one screen at a time and does not involve "back" actions. Used primarily in authentication flows.

Good for login/signup

Stack Navigator

Definition: Enables transition between screens where each screen is placed on a stack, as the History API does. The navigator automatically implements the native transition animations.

Very simple!

Primarily used to go back and forth between list and detail views or to walk the user through a process.

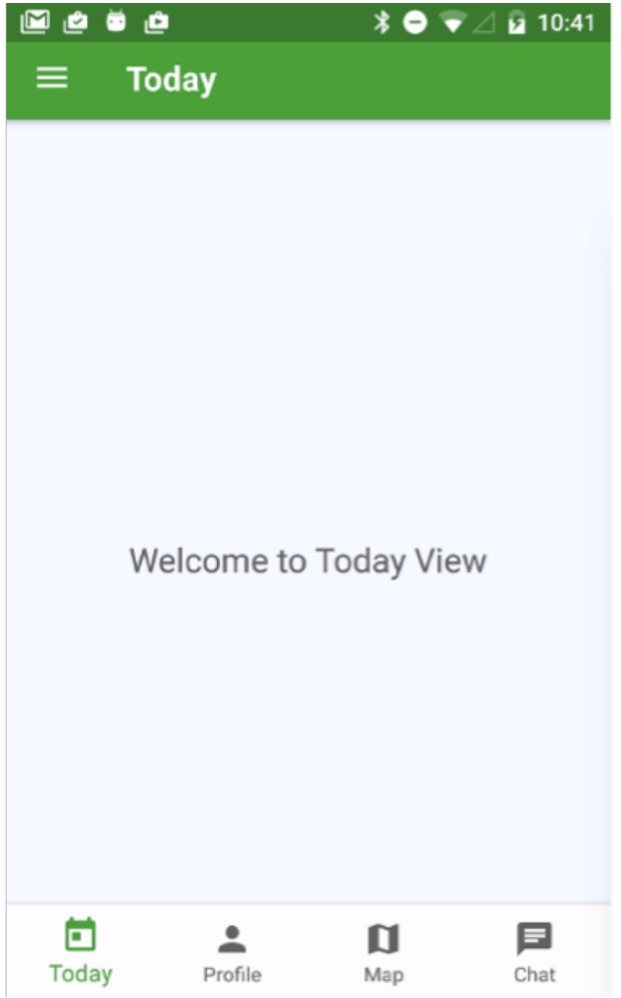
Contacts app

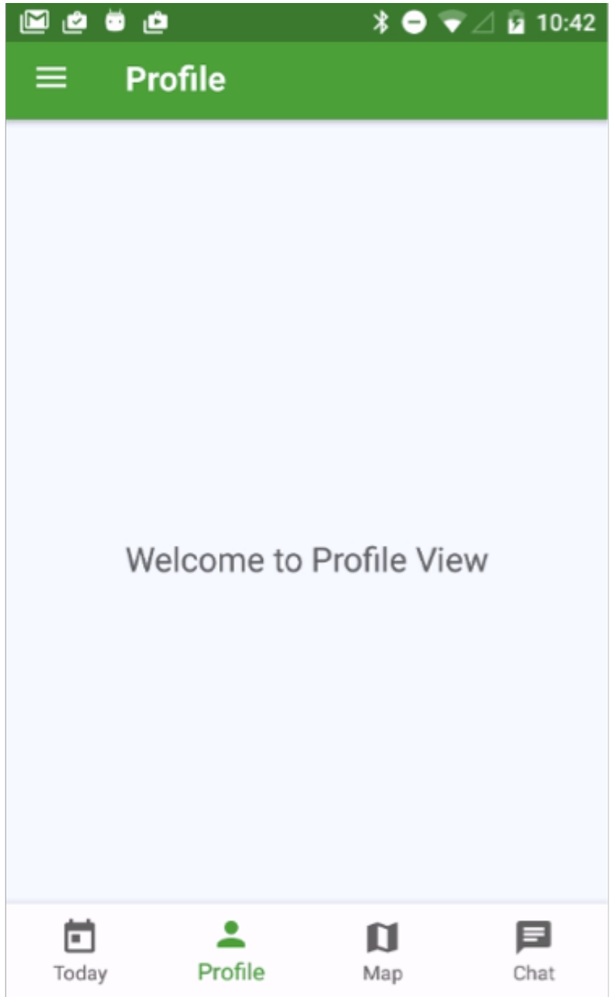
Shopping

Tab Navigator[2]

Definition: Implements tabs at the bottom or the top of the screen to enable transitions among them.

Most commonly used navigation to establish a main menu for the different sections/parts of an application.



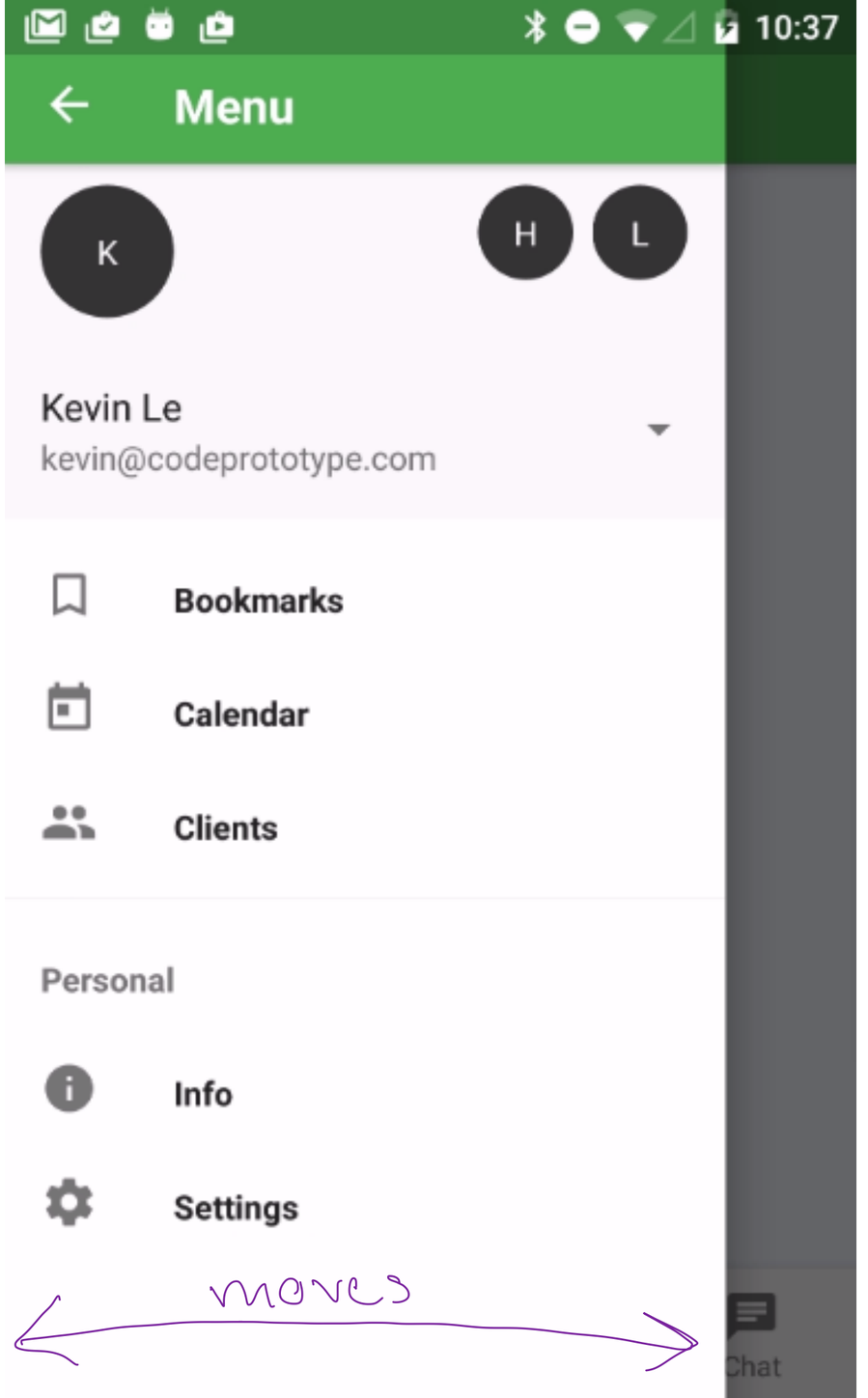


[2] Image source

Drawer Navigator[3]

Definition: Enables tab-like transitions through a hidden drawer that can be exposed and hidden.

Used primarily for options and settings.



[3] Image source

The Big Picture

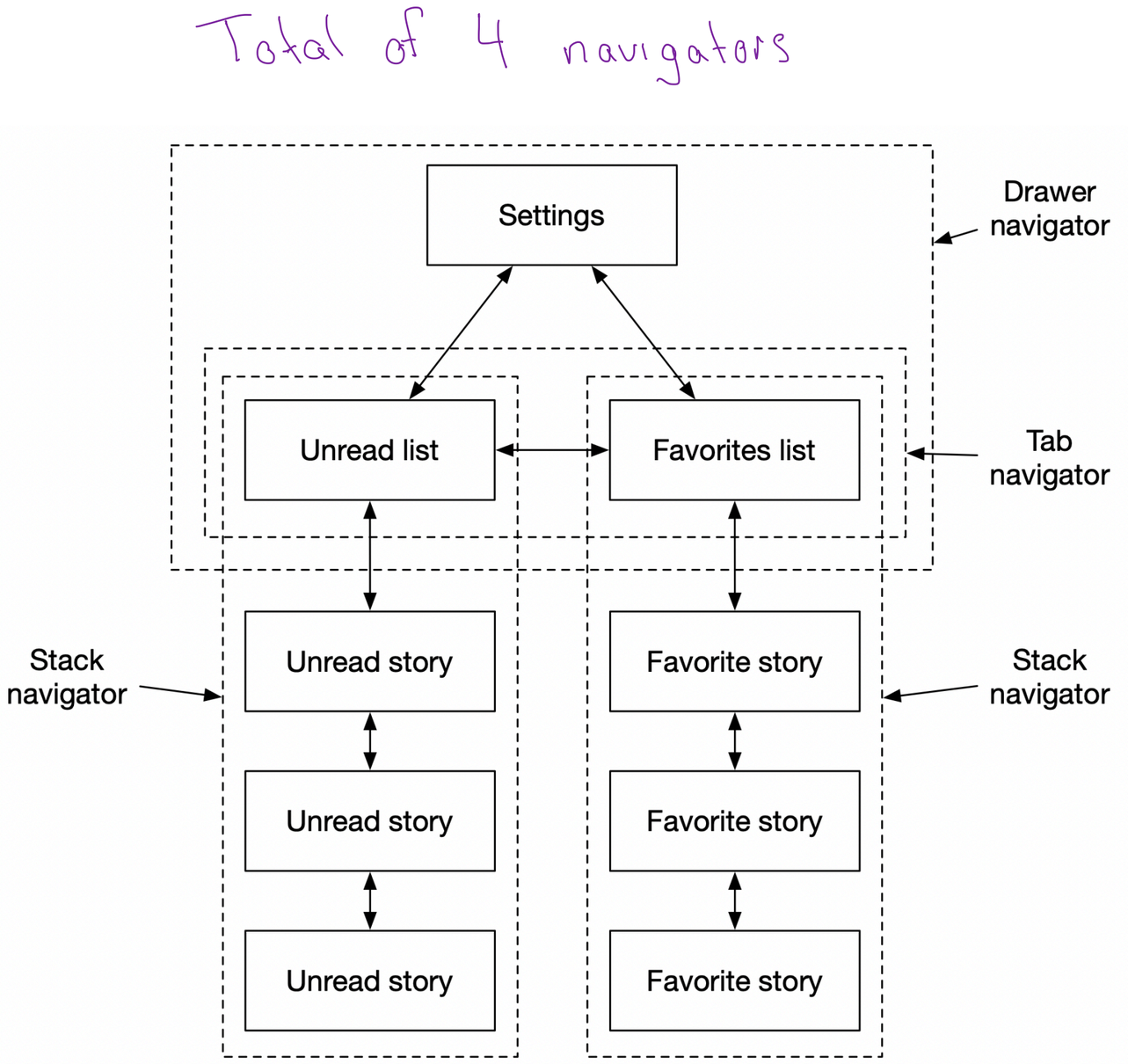
Every RN project will use a combination of these navigators.

As a working example, let's imagine a *news/RSS reader* app with the following specifications:

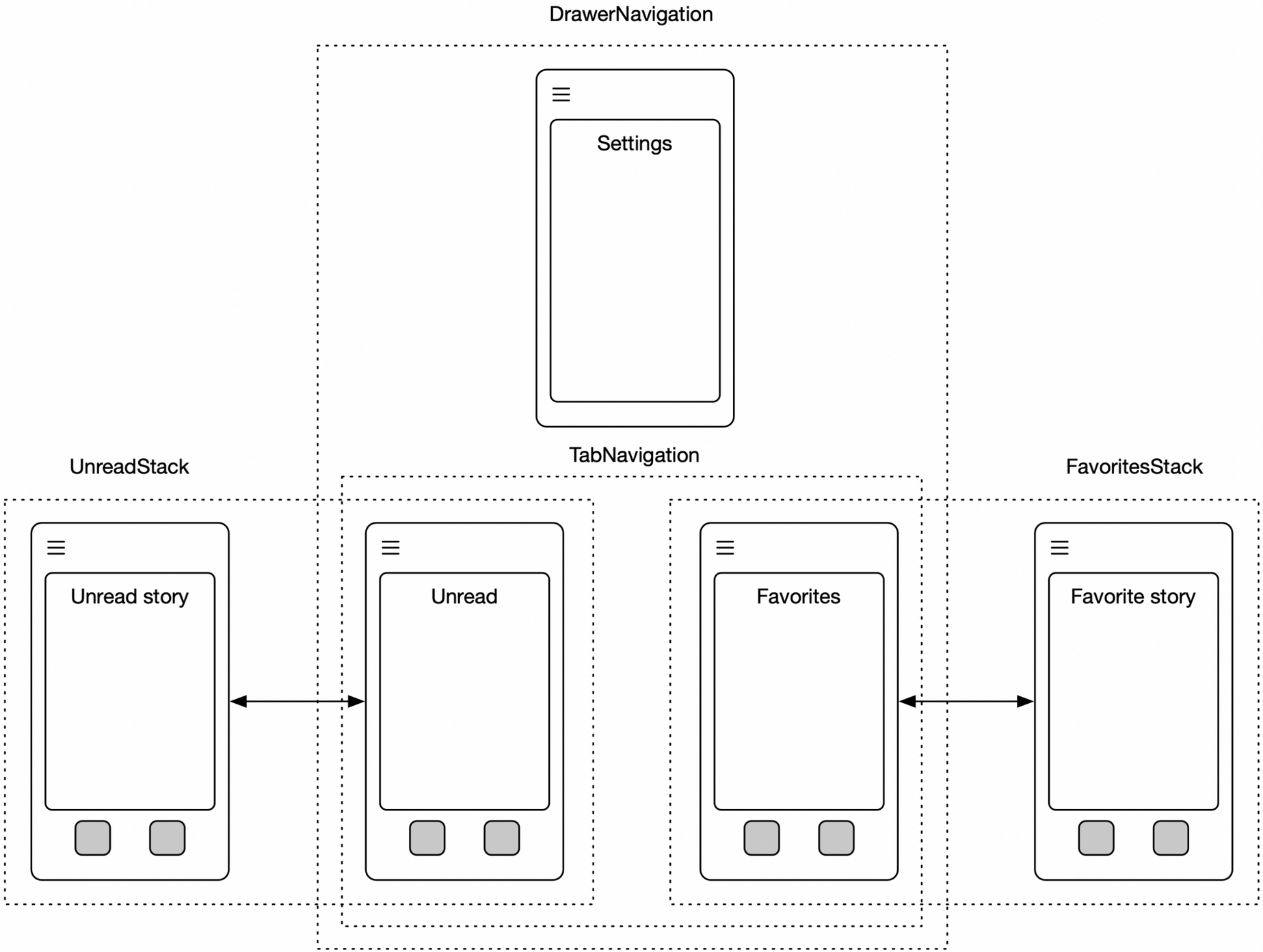
1. Landing page with *unread* and *favorites* tabs
2. Pages to show unread and favorite stories
3. Settings to change reading mode

My implementation should include:[4]

1. Tab navigator for the unread and favorites pages
2. Stack navigators for the unread and favorite stories
3. Drawer navigator for the drawer and the tabbed pages



[4] See example on Snack


DrawerNavigation
Settings
TabNavigation
UnreadStack
Unread story
Unread
FavoritesStack
Favorites
Favorite story

Screens

Each screen is a React class component with the elements we would like on them.

```
class UnreadScreen extends React.Component {
  render() {
    return (
      <View>
        <Text>Unread Stories</Text>
            ...
      </View>
    );
  }
}
```

UnreadStack & FavoritesStack

To create a stack navigator, we can use `createStackNavigator`:

```
import { createStackNavigator } from 'react-navigation-stack';
...
createStackNavigator(RouteConfigs, StackNavigatorConfig);
```

`RouteConfigs` is an object that maps route name to a route config. For each screen, we define a mapping:

```
Unread: UnreadScreen
```

Creates mapping between screen and component

And a set of parameters for that screen:

```
navigationOptions: ({ navigation }) => ({
    title: `${navigation.state.params.count} New Stories`,
})
```

`StackNavigatorConfig` includes parameters and options for the router, for example:

```
// routing options
initialRouteName: 'Home'
// visual options
headerMode: 'none'
```

Many options

There won't be a header

```
import { createStackNavigator } from 'react-navigation';

const UnreadStack = createStackNavigator({
  Unread: UnreadScreen,
  Story: UnreadStory,},
  { headerMode: "none" }
);
```

TabNavigation

To create a tab navigator, we can use `createBottomTabNavigator`:

```
import { createBottomTabNavigator } from 'react-navigation-tabs';
...
createBottomTabNavigator(RouteConfigs, TabNavigatorConfig);
```

```
const TabNavigation = createBottomTabNavigator({
    Unread: UnreadStack,
    Favorites: FavoritesStack},
    { navigationOptions: ({ navigation }) => ({
        tabBarIcon: ({ focused, tintColor }) => {
        // Icon formatting
        },
    }),
    tabBarOptions: { activeTintColor: 'tomato', inactiveTintColor: 'gray' },
    animationEnabled: true,
    }
);
```

Different navigators will have different options

DrawerNavigation

To create the drawer navigation for settings, we can use `createDrawerNavigator`:

```
import { createDrawerNavigator } from 'react-navigation-drawer';
...
createDrawerNavigator(RouteConfigs, DrawerNavigatorConfig);
```

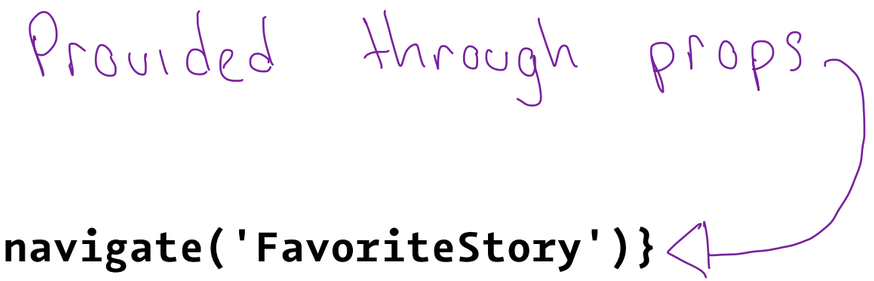
```
const DrawerNavigator = createDrawerNavigator ({
    Home:{ screen:TabNavigationStackNavigator },
    Settings:{ screen:SettingsStackNavigator }},
    { initialRouteName:'Home' }
)
```

Optional in most cases

`navigation` prop[5]

Each screen is automatically provided with a `navigation` prop (no need to use `constructor()` for the `navigation` prop) that provides access to parameters and actions, e.g., `navigate`, `goBack`, `state`.

```
<Button
    style={styles.button}
    color="tomato"
    title="Read"
    onPress={() => this.props.navigation.navigate('FavoriteStory')}
/>
```



[5] Read more on `navigation` prop

Navigator actions

Each navigator has a set of specialized actions associated with them that provide low-level access to the navigation behavior of the navigator:

- `StackActions` include `reset`, `replace`, `push`, `pop`, `popToTop`
- `SwitchActions` include `jumpTo`
- `DrawerActions` include `openDrawer`, `closeDrawer`, `toggleDrawer`
- `NavigationActions` include `navigate`, `back`, `setParams`, `init`

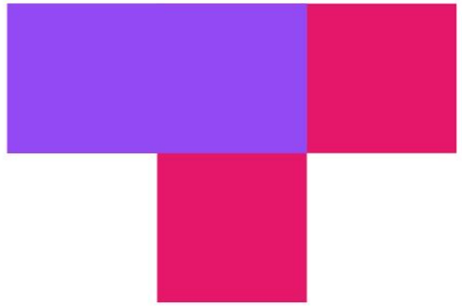
Scrollable `View` Components

For content that does not fit into the device screen, RN provides scrollable `View` components, including `ScrollView`, `FlatList`, and `SectionList`.

Most common and easiest to use

```
<ScrollView style={styles.scrollView}>
    <Text style={styles.text}>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
        et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
        aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
        dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa
        qui officia deserunt mollit anim id est laborum.
    </Text>
</ScrollView>
```

TopHat Quiz



TOP HAT

Mobile Input via Gestures using React Native

Why worry about gesture?

Unlike the web's indirect mapping

Because of the direct/absolute mapping between input space and the screen space and the touch-sensitive input capabilities, gestures are a resource for mobile development. A number of RN packages provide access to gestures:

» Gesture Responder System
» PanResponder
» React Native Gesture Handler
» React Native Swipe Gestures

Most common

Handling Gestures Using `PanResponder`

`PanResponder` uses the core gesture responder system to reconcile several touches into a single gesture that can be used to recognize multi-touch gestures.

To initialize, we create a PanResponder object with event handlers:

```
import { PanResponder } from 'react-native';
...
this._panResponder = PanResponder.create({
    onStartShouldSetPanResponder: (evt, gestureState) => true,
    onStartShouldSetPanResponderCapture: (evt, gestureState) => true,
    onMoveShouldSetPanResponder: (evt, gestureState) => true,
    onMoveShouldSetPanResponderCapture: (evt, gestureState) => true,
    onPanResponderGrant: (evt, gestureState) => { },
    onPanResponderMove: (evt, gestureState) => { },
    onPanResponderTerminationRequest: (evt, gestureState) => true,
    onPanResponderRelease: (evt, gestureState) => { },
    onPanResponderTerminate: (evt, gestureState) => { },
    onShouldBlockNativeResponder: (evt, gestureState) => { return true; },
});
```

`PanResponder` Event Handlers

Event handlers utilize `nativeEvent` and `gestureState` objects:

```
onPanResponderMove: (event, gestureState) => {}
```

`nativeEvent` object provides properties such as `locationX` and `locationY` (position of the touch with respect to the element).

`gestureState` object provides properties about the gesture, such as `vx` and `vy` (velocity of the gesture).

Most important. Begins gesture measuring

```
onPanResponderGrant: (evt, gestureState) => { }
```

Indicates that the gesture has started. The screen should provide the user with visual feedback on what's happening.

```
onPanResponderMove: (evt, gestureState) => { }
```

`gestureState` provides access to the most recent move distance (`gestureState.move{X,Y}`) and the accumulated gesture distance (`gestureState.d{x,y}`).

```
onPanResponderRelease: (evt, gestureState) => { }
```

Indicates that the user has released all touches while this view is the responder.

```
onPanResponderTerminate: (evt, gestureState) => { }
```

Indicates that another component has become the responder, so this gesture should be cancelled.

```
onShouldBlockNativeResponder: (evt, gestureState) => { return true; }
```

Returns whether this component should block native components from becoming the JS responder (only on Android).

Associating Gestures with Screens[6]

We provide `panHandlers` as a `prop` into the component:

```
<View style={styles.container} {...this._panResponder.panHandlers}>
    // ...
</View>
```

[6] See example in Snack

Are we done? *No*.

We need to be able to respond to the gestures with appropriate behaviors on the interface, and that's done using animation packages, particularly:

>> `Animated`

>> `LayoutAnimation`

Animated

The `Animated` library provides the ability to create time-based animation using a number of methods.

```
this.state = { // Create Animated.Value
    fadeValue: new Animated.Value(0) // Connect it to style attributes
};

_start = () => {
  Animated.timing(this.state.fadeValue, { // Animate
    toValue: 1,
    duration: 1000
  }).start();
};
```

`Animated.sequence()` allows sequencing several animations.

`Animated.spring()` animates attributes without a set time in different motion styles, e.g., `velocity`, `bounciness`, `speed`, `tension`, `friction`.

`Animated.interpolate()` maps input ranges to output ranges using linear interpolation.

`Easing` functions help in gradual acceleration or deceleration (e.g., `easing: Easing.back()`).[7]

---

[7] See example in Snack

`LayoutAnimation`

The `LayoutAnimation` library animates the entire screen when there are changes in the layout, e.g., when an element is removed from the screen.

`LayoutAnimation` is used before `setState()` is called.

`Animated` animates specific components without changing the layout of the screen, while `LayoutAnimation` animates all components on the screen when the layout changes.

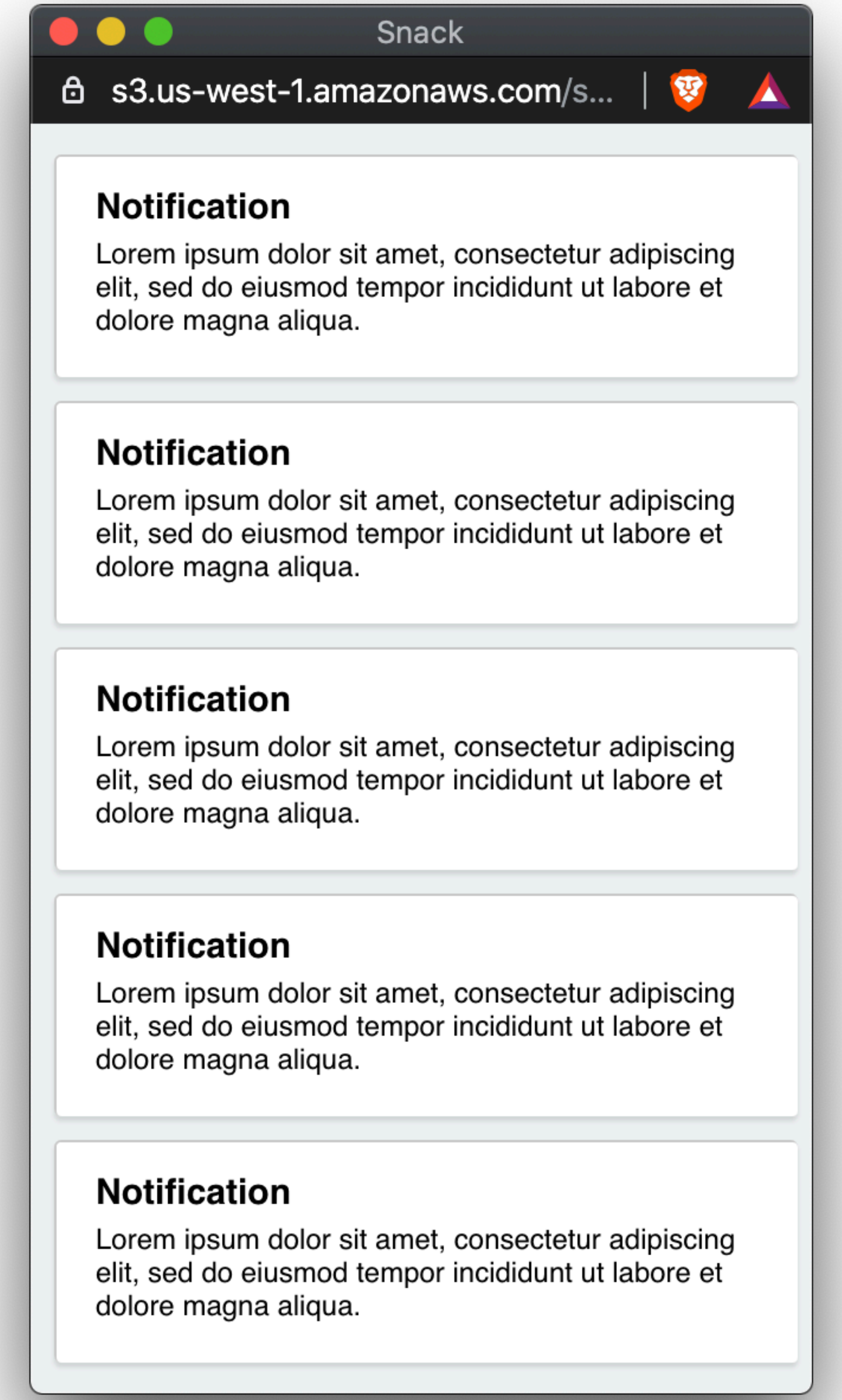
8

```
import { UIManager, LayoutAnimation } from 'react-native';
...
<TouchableOpacity
    onPress={() => {
        LayoutAnimation.configureNext(LayoutAnimation.Presets.spring);
        this.setState({expanded: !this.state.expanded}); }}>
  <Text>{this.state.expanded ? 'Expanded text' : 'Collapsed text'}</Text>
</TouchableOpacity>
```

Takes care of what happens next

[8] See example in Snack

Notifications example[9]



[9] See combined example in Snack

Working with Date Objects in JS

`Date`

The `Date` object represents a single moment in time in a platform-independent format. We need to use the object in ways that are meaningful both for the server API and for the user.

Users would like to see something like:

```
Thu Nov 07 2019 11:53:47 GMT-0600 (Central Standard Time)
```

While the server expects something like:[10]

```
2019-11-07T11:53:47-06:00
```

---

[10] ISO 8601 Standard for Date and Time Formats

Good news: We can serialize `Date` object into the ISO 8601 format.

```
var date = new Date();
console.log(date); // Thu Nov 07 2019 11:58:58 GMT-0600 (Central Standard Time)

var json = JSON.stringify(date);
console.log(json);  // "2019-11-07T17:58:58.487Z"
```

Bad news: There is no good method to deserialize back to a date format.

```
var json = "\"2019-11-07T17:58:58.487Z\"";


var dateStr = JSON.parse(json);
console.log(dateStr); // 2019-11-07T17:58:58.487Z
```

The trick: We can use the `Date` constructor for this translation.

```
var json = "\"2019-11-07T17:58:58.487Z\"";

var dateStr = JSON.parse(json);
console.log(dateStr); // 2019-11-07T17:58:58.487Z

var date = new Date(dateStr);
console.log(date);  // Thu Nov 07 2019 11:58:58 GMT-0600 (Central Standard Time)
```

Assignment Preview

React Native 2

- A day view that shows user meals and exercises and make it the default view.
- A section of the day view that allows the user to compare their goals versus the current day's stats.
- A view that allows the user to add/edit/remove exercises to the current day.
- Capability to log-out/delete a user.

What did we learn today?

- Mobile Navigation using React Native
- Mobile Input via Gestures using React Native
- Working with Date object in JS
- Assignment Preview