Topic	**SAVE & READ STORIES USING FIREBASE**	
Class Description	**In today's class, the student will use firebase to add stories and get stories, instead of using temporary data.**	
Class	**C88**	
Class time	**45 mins**	
Goal	• Resolve the bug from the last class. • Integrate firebase to save stories to the database. • Read stories from the database.	

Resources Required	• Teacher Resources
	  ○ Visual Studio Code Editor
	  ○ laptop with internet connectivity
	  ○ earphones with mic
	  ○ notebook and pen
	• Student Resources
	  ○ Visual Studio Code Editor
	  ○ laptop with internet connectivity
	  ○ earphones with mic
	  ○ notebook and pen	

Class structure	**Warm-Up** **Teacher-Student Collaborative Activity** **Wrap-Up** **Note: This class requires database configuration; Teacher to ask the student to live share VSC and perform the following activities to avoid writing the same code twice at both ends.*	**5 mins** **35 mins** **5 mins**

- **WARM-UP SESSION - 5 mins**

CONTEXT

- **Discuss the flow of the app to integrate the database with other screens of**

the APP.	
• **Teacher starts slideshow**  **from slides 1 to 10** • Refer to speaker notes and follow the instructions on each slide.	
• **Activity details**	
Hey <student name>. How are you? It's great to see you! Are you excited to learn something new today? ***Run the presentation from slide 1 to slide 3.*** **The following are the warm-up session deliverables:** • Connecting students to the previous class.	**ESR:** Good!
• **QnA Session**	
• **Question**	**Answer**
To which of the following screens have we added themes in our App? A. CreateStory screen B. StoryScreen screen C. FeedScreen screen D. All of the above	**D**
The theme of the app is based on_____ A. The user preference which is stored in the database. B. Selection of the theme based on the app itself. C. User can't select the theme, it is set by default. A. User has to click on the button to pick the theme.	**A**
• **Continue the warm-up session**	
• **Activity details**	**Solution/Guidelines**

Run the presentation from slide 4 to slide 10 to set the problem statement. **The following are the warm-up session deliverables:** • Discuss the flow of the app to integrate the database with other screens of the APP. • Discuss any possible bugs in the APP and ways to fix them.	Narrate the slides by using hand gestures and voice modulation methods to bring in more interest in students.
Teacher ends slideshow 	
• **Teacher-Student Collaborative Activity - 35 mins**	
• **Guide the student to start screen share.** • **Guide Student to live share the code from VSC** • **Teacher gets into fullscreen.**	

Class Steps	Teacher Action	Student Action
Step 2: Teacher-Student Collaborative Activity (35 min)	Today you will be driving the class while I am in the passenger seat. I will surely be guiding you, don't you worry! Are you ready? *NOTE: The previous class code is provided at Teacher Activity 1; In case there is an issue with the student's code you can use this.* Let's start by adding the stories to our Database. We will handle the bug later. We will work in your previous class code; Open the code in VSC and start live share.	**ESR**: Yes!

We will start with creating new stories on our **CreateStory** screen.

Now, the first thing that we need is a button to submit our stories, right?

Let's start by adding the **Submit** button in the **CreateStory.js** file.

import <Button> & Alert at the top from react-native.

import {Button, Alert} from 'react-native'

Now, include the **<Button />** component inside **<View>**.

```
                <View style={styles.fieldContainer}>
                    <TextInput
                        style={[this.state.light_theme ? styles.inputFontLight : styl
                        onChangeText={(moral) => this.setState({ moral })}
                        placeholder={"Moral of the story"}
                        multiline={true}
                        numberOfLines={4}
                        placeholderTextColor={this.state.light_theme ? "black" : "whi
                    />
                </View>
                <View style={styles.submitButton}>
                    <Button
                        onPress={() => this.addStory()}
                        title="Submit"
                        color="#841584"
                    />
                </View>
            </ScrollView>
        </View>
```

Now on this button, we have an **onPress()** event that calls a function **addStory()**.

	As you might have guessed, we need to have this function save the story in the database. Let's do that as shown in the following code snippet:	

```
async addStory() {
      if (this.state.title && this.state.description && this.state.story &&
this.state.moral) {
          let storyData = {
              preview_image: this.state.previewImage,
              title: this.state.title,
              description: this.state.description,
              story: this.state.story,
              moral: this.state.moral,
              author: firebase.auth().currentUser.displayName,
              created_on: new Date(),
              author_uid: firebase.auth().currentUser.uid,
              likes: 0
          }
          await firebase
              .database()
              .ref("/posts/" + (Math.random().toString(36).slice(2)))
              .set(storyData)
              .then(function (snapshot) {

              })
          this.props.navigation.navigate("Feed")
      } else {
          Alert.alert(
              'Error',
              'All fields are required!',
              [
                  { text: 'OK', onPress: () => console.log('OK Pressed') }
              ],
              { cancelable: false }
          );
      }
  }
```

Let's go through this function.

We are first checking if all the fields were filled or not - title, description, story, and moral. We are doing this using the **and** operator **(&&).** If there are not, we are giving out an Alert saying that **All fields are required!** (**Import Alert at the top from "react-native"**)

The teacher can ask the student to run the code to make sure it shows an alert when any of the above fields is not filled.

Next, we create an object **storyData** in which we are saving the data for the story.

This data contains -

1. Image (image_1, image_2, etc.) [Sourced from Assets]
2. Title [To enter by users]
3. Description [To enter by users]
4. Story [To enter by users]
5. Moral [To enter by users]
6. Author [from **firebase.auth()**]
7. Created On [the current date]
8. Author's Unique ID [random user id generated by app uid]
9. Likes [since a new story has 0 likes]

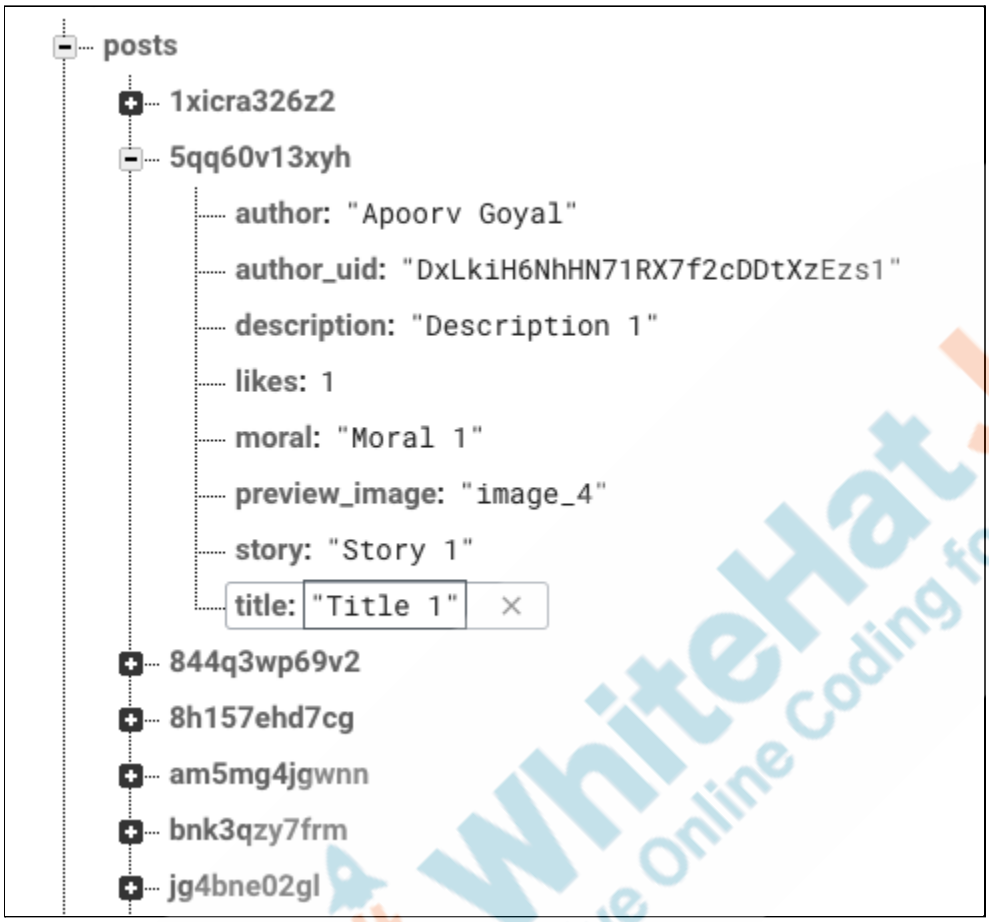
Note: As shown above only 4 fields are to be entered by users; the other 5 fields will be generated /provided by APP.

We are then saving this data into firebase by creating a **random unique id** for records/stories that will be added to our app. We are saving the story inside a reference object called **posts**.

Then we are finally navigating the User to the Feed Screen.

	Awesome! Now our stories are getting saved to the firebase. We can test it out too! *The teacher and student test the code. Make sure to check the database to see how the story is being saved.*	

Output:



Now for the Feed Screen's part, we want to make sure we are fetching the stories from our Firebase Database. For that, let's go back to our Feed Screen and create a new function to **fetchStories()** and call it in our **componentDidMount()**.

We will first add a **new state** for our stories, which will be an **empty array**.

Our constructor would be like -

```
constructor(props) {
        super(props);
        this.state = {
```

```
            fontsLoaded: false,
            light_theme: true,
            stories: []
        };
    }
```

Next, we will create a function **fetchStories()** and call it in the **componentDidMount()** function -

```
componentDidMount() {
        this._loadFontsAsync();
        this.fetchStories();
        this.fetchUser();
    }

    fetchStories = () => {
        firebase
            .database()
            .ref("/posts/")
            .on("value", (snapshot) => {
                let stories = []
                if (snapshot.val()) {
                    Object.keys(snapshot.val()).forEach(function (key) {
                        stories.push({
                            key: key,
                            value: snapshot.val()[key]
                        })
                    });
                }
                this.setState({ stories: stories })
            }, function (errorObject) {
                console.log("The read failed: " + errorObject.code);
            })
    }
```

In this function, we are checking all values on the **posts** object reference and whatever we get, we are iterating over all the key value pairs using the **map()** function and storing them inside another array called **stories**. We are then updating **this.state** property to render the data.

Now, we need to use this new story from the state, but there might be a case that there

are no stories in the database.

Let's now handle them both together in our **render()** function -

```
return (
                <View style={this.state.light_theme ? styles.containerLight :
styles.container}>
                    <SafeAreaView style={styles.droidSafeArea} />
                    <View style={styles.appTitle}>
                        <View style={styles.appIcon}>
                            <Image source={require("../assets/logo.png")} style={{ width:
60, height: 60, resizeMode: 'contain', marginLeft: 10 }}></Image>
                        </View>
                        <View style={styles.appTitleTextContainer}>
                            <Text style={this.state.light_theme ?
styles.appTitleTextLight : styles.appTitleText}>
                                Storytelling App
                            </Text>
                        </View>
                    </View>
                    {
                        !this.state.stories[0] ?
                            <View style={styles.noStories}>
                                <Text style={this.state.light_theme ?
styles.noStoriesTextLight : styles.noStoriesText}>No Stories Available</Text>
                            </View>
                            : <View style={styles.cardContainer}>
                                <FlatList
                                    keyExtractor={this.keyExtractor}
                                    data={this.state.stories}
                                    renderItem={this.renderItem}
                                />
                            </View>
                    }
                </View>
            )
```

Here, near the **FlatList()** function, we have added a condition inside a pair of curly brackets {}.

In React Native, we can add JavaScript code inside the return statement of a

render() function using curly brackets.

We are checking if the first item/data/story from the database is present in the stories or not?

This code would have not worked since we initially had an empty array in our state by default, but since we are trying to fetch stories, if there are no stories available, it will push a **null** value in this array.

Therefore, we are checking if the first value is a valid value or not? If it's an object, it's a valid value else it will be **null** which is not a valid value.

If it's not a valid value, we are displaying a text that says that there are **No Stories Available**. Otherwise, we are using our **<FlatList>** component.

Inside the Flatlist, we have changed our **data** attribute's value to the state.

We need to also add subsequent styling for the newly created text that displays that there are no stories -

```
noStories: {
   flex: 0.85,
   justifyContent: "center",
   alignItems: "center"
 },
 noStoriesTextLight: {
   fontSize: RFValue(40),
   fontFamily: "Bubblegum-Sans"
 },
 noStoriesText: {
   color: "white",
   fontSize: RFValue(40),
   fontFamily: "Bubblegum-Sans"
 }
```

	Great! Now we are done here, however, we still need to make changes to our **StoryCard**. That's because our **story** that we are passing to is not in the same structure as before.	

	This time, our story is an object which has a **key** as the unique ID of the story and **value** as the story's data. Let's make the changes to our **StoryCard**, to include the changes of the **key** and the **value.**	

Let's first change the constructor in our **StoryCard.js** to store the keys and values separately -

```
export default class StoryCard extends Component {
    constructor(props) {
        super(props);
        this.state = {
            fontsLoaded: false,
            light_theme: true,
            story_id: this.props.story.key,
            story_data: this.props.story.value
        };
    }
```

Next, inside our **render()** function, let's create a variable called **story** that is equal to our state **story_data** -

```
render() {
    let story = this.state.story_data
    if (!this.state.fontsLoaded) {
        return <AppLoading />;
```

Remember that when we saved our story, we saved it such that its value is **image_1, image_2 … image_5**.

Therefore, we need to create an object here which maps the value of these keys with the path of their respective image as shown below -

```
render() {
    let story = this.state.story_data
    if (!this.state.fontsLoaded) {
        return <AppLoading />;
    } else {
        let images = {
            "image_1": require("../assets/story_image_1.png"),
            "image_2": require("../assets/story_image_2.png"),
            "image_3": require("../assets/story_image_3.png"),
            "image_4": require("../assets/story_image_4.png"),
            "image_5": require("../assets/story_image_5.png")
        }
        return (
```

Now change the source of the **<Image>** component that displays the image of the story too -

```
<Image source={images[story.preview_image]}
```

And the styles however for it will remain the same.

Now, we were using **this.props.story** to fetch the story until now in this function, but this time, we will only use the **story** variable since we have our story data stored in it.

The final version of the **render()** function would look like -

```
render() {
  let story = this.state.story_data;
  if (!this.state.fontsLoaded) {
    return <AppLoading />;
  } else {
    let images = {
      image_1: require("../assets/story_image_1.png"),
      image_2: require("../assets/story_image_2.png"),
      image_3: require("../assets/story_image_3.png"),
      image_4: require("../assets/story_image_4.png"),
      image_5: require("../assets/story_image_5.png")
```

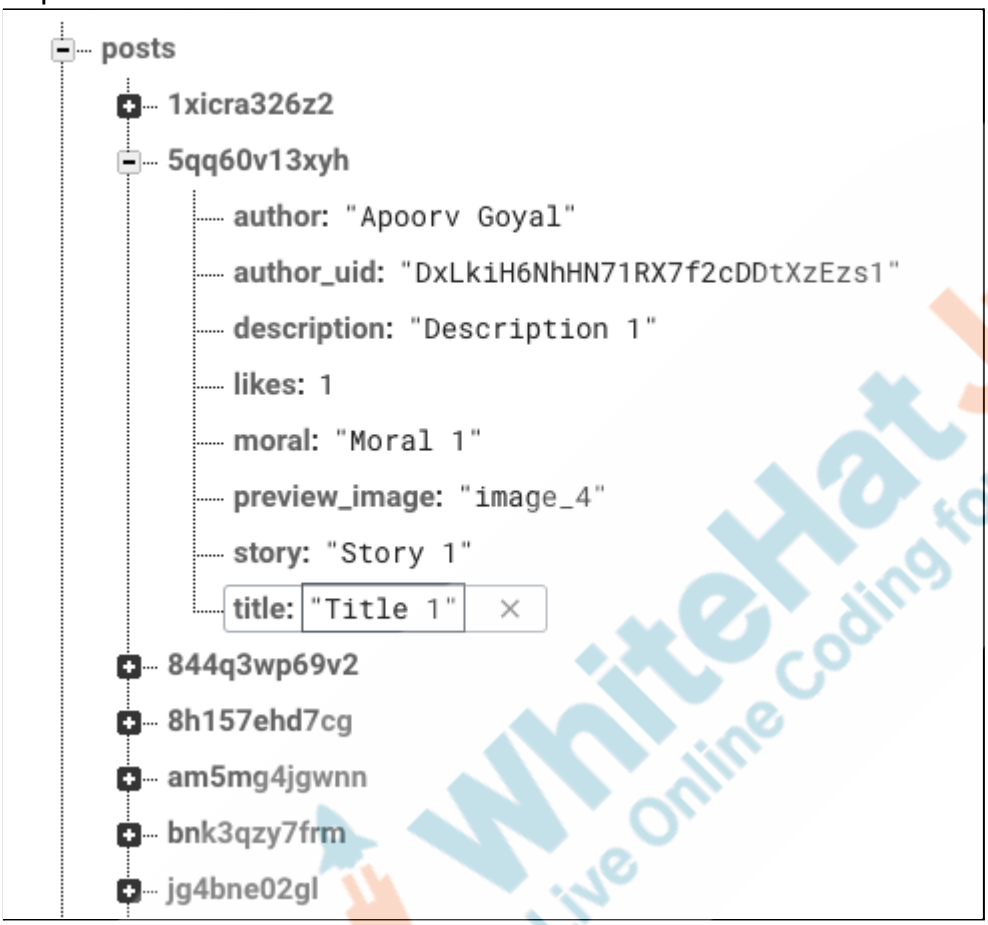
```
  };
  return (
    <TouchableOpacity
      style={styles.container}
      onPress={() =>
        this.props.navigation.navigate("StoryScreen", {
          story: this.props.story
        })
      }
    >
      <SafeAreaView style={styles.droidSafeArea} />
      <View
        style={
          this.state.light_theme
            ? styles.cardContainerLight
            : styles.cardContainer
        }
      >
        <Image
          source={images[story.preview_image]}
          style={styles.storyImage}
        ></Image>
        <View style={styles.titleContainer}>
          <View style={styles.titleTextContainer}>
            <Text
              style={
                this.state.light_theme
                  ? styles.storyTitleTextLight
                  : styles.storyTitleText
              }
            >
              {story.title}
            </Text>
            <Text
              style={
                this.state.light_theme
                  ? styles.storyAuthorTextLight
                  : styles.storyAuthorText
```

```
            }
          >
            {story.author}
          </Text>
          <Text
            style={
              this.state.light_theme
                ? styles.descriptionTextLight
                : styles.descriptionText
            }
          >
            {this.props.story.description}
          </Text>
        </View>
      </View>

      <View style={styles.actionContainer}>
        <View style={styles.likeButton}>
          <Ionicons
            name={"heart"}
            size={RFValue(30)}
            color={this.state.light_theme ? "black" : "white"}
          />
          <Text
            style={
              this.state.light_theme
                ? styles.likeTextLight
                : styles.likeText
            }
          >
            12k
          </Text>
        </View>
      </View>
    </View>
  </TouchableOpacity>
);
}
```

```
}
```

Output:



Our DB is integrated successfully!

Now let's talk about that bug.

The problem is, our **componentDidMount()** on the Create Story Screen and the Feed Screen are not getting called every time we come to those screens because our Navigator is caching those screens.

	Cached *data are files, scripts, images, and other multimedia stored on your device after opening an* ***app*** *or visiting a website for the first time. This data is then used to quickly gather information about the* ***app*** *or website every time you revisit the website or app, reducing the load time. Our app stores previous stories in the RAM of the mobile phone, so here* ***componentDidMount()*** *is retrieving that pre-stored data instead of getting it from the DB.* If we create a new story, we will not be able to see it on the Feed Screen until we close the app entirely and then open it. How do you think we should solve this?	**ESR:** Varied!
	I have a solution! Both the screens are connected through the Tab Navigator, and we converted our Tab Navigator to a class component in the last class. What if we maintain a state in our Tab Navigator and update those states from our screens? That will force the tab navigator to update itself, and it will end up updating the screens as well. Let's see how we can do that!	

First, we will have to create a state in our Tab Navigator's constructor -

```
constructor(props) {
    super(props);
    this.state = {
        light_theme: true,
        isUpdated: false
    };
}
```

This will be for, if our screen **isUpdated** or not. We will set it by default to **False**.

Next, let's create two functions to update this state. One will update it to **true** while the other will update it to **false** -

```
changeUpdated = () => {
    this.setState({ isUpdated: true })
}

removeUpdated = () => {
    this.setState({ isUpdated: false })
}
```

Now we need to pass these functions along with the components in our **<Tab.Screen> components**, but we can't do it directly. We will have to write a wrapper for it.
The reason we can't do it directly is because we are just passing the names of the components there -

```
<Tab.Screen name="Feed" component={Feed} />
<Tab.Screen name="Create Story" component={CreateStory} />
```

Therefore, we can't add any **props** to it. Instead, we can create two functions that return these components with the **props** we need and then use those functions instead of the component in our **<Tab.Screen>**

```
renderFeed = (props) => {
    return <Feed setUpdateToFalse={this.removeUpdated} {...props} />
```

```
}

renderStory = (props) => {
    return <CreateStory setUpdateToTrue={this.changeUpdated} {...props} />
}
```

Here, we have created **renderFeed()** and **renderStory()**.

In the **renderFeed()** function, we are sending a prop **setUpdateToFalse**, and it's our **removeUpdated()** function which sets the **isUpdated** state to **false**.

Whereas, in the **renderStory()**, we are sending the **setUpdatedToTrue** prop, and it's our **changeUpdated()** function which sets the **isUpdated** state to **True**

Now we are going to do this because when we create a story, we want to tell the Tab navigator that our screen **needs to be updated.** The feed screen, once updated, can tell the tab navigator that it has updated itself, and it can change its state back to false in case another story is added.

Now let's use these functions as well -

```
<Tab.Screen name="Feed" component={this.renderFeed} />
<Tab.Screen name="Create Story" component={this.renderStory} />
```

	Great! Now the props that we are passing - **setUpdatedToFalse** & **setUpdatedToTrue**, we need to use them in our screens as well. Let's do that -	

In **CreateStory.js**, we will use the **setUpdatedToTrue** function right before we are navigating to the Feed screen after saving the story in the database -

```
this.props.setUpdateToTrue()
this.props.navigation.navigate("Feed")
```

In the **FeedScreen.js,** we will use the **setUpdatedToFalse** right after we have successfully fetched the stories -

```
fetchStories = () => {
    firebase
        .database()
        .ref("/posts/")
        .on("value", (snapshot) => {
            let stories = []
            if (snapshot.val()) {
                Object.keys(snapshot.val()).forEach(function (key) {
                    stories.push({
                        key: key,
                        value: snapshot.val()[key]
                    })
                });
            }
            this.setState({ stories: stories })
            this.props.setUpdateToFalse()
        }, function (errorObject) {
            console.log("The read failed: " + errorObject.code);
        })
}
```

There's one last thing that needs to be done!

In our Tab Navigator and Drawer Navigator, we can specify that we want to **unmount** a component as soon as a user goes away from a screen.

That will again help with managing this issue with all the screens.

Let's do that -

In the **TabNavigator.js -**

```
<Tab.Screen name="Feed" component={this.renderFeed} options={{ unmountOnBlur: true }} />
<Tab.Screen name="Create Story" component={this.renderStory} options={{ unmountOnBlur: true }} />
```

And in the **DrawerNavigator.js -**

```
<Drawer.Screen name="Home" component={StackNavigator} options={{ unmountOnBlur: true }} />
<Drawer.Screen name="Profile" component={Profile} options={{ unmountOnBlur: true }} />
<Drawer.Screen name="Logout" component={Logout} options={{ unmountOnBlur: true }} />
```

We have added an **options** attribute to the **<Tab.Screen>** and **<Drawer.Screen>** components.

This **options** attribute is available for all navigation type screen components. In it, we have used a specific attribute **unmountOnBlue**, and we have set it to **true** for all the screens. This is the most used type of screen option which unmounts the screen as soon as a user leaves it.

	Now, we are done with fixing bugs, and we also integrated our stories with the database. Our App is almost complete now. *The teacher will ask the student to run and check the App. Enter a story to check if it is getting updated in the DB.* All that is needed is that we make our Drawer look a bit better and also, that our Like functionality works. We'll tackle these in the next class and complete this app.	

Teacher Guides Student to Stops Screen Share

WRAP-UP SESSION - 5 Mins

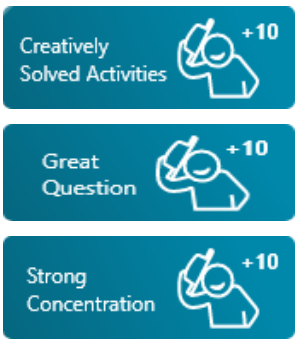
Teacher starts slideshow  from slide 11 to slide 20	
Activity details	**Solution/Guidelines**
Run the presentation from slide 11 to slide 20. **Following are the wrap-up session deliverables:** • **Explain the facts and trivias** • **Next class challenge** • **Project for the day** • **Additional Activity**	Guide the student to develop the project and share it with us.
Quiz time - Click on the in-class quiz	
Question	**Answer**
What does the following piece of code do? `async addStory() { if (this.state.title && this.state.description && this.state.story && this.state.moral) { let storyData = { preview_image: this.state.previewImage, title: this.state.title, description: this.state.description, story: this.state.story, moral: this.state.moral, author: firebase.auth().currentUser.displayName, created_on: new Date(),` A. The function addStory() is adding the story in the database. B. The function addStory() is adding the story only in the application. C. The function addStory() allows the user to write the story. D. The function addStory() add the story in the array called as storyData.	**A**
What does the following snippet of code do? `barStyle={this.state.light_theme ? styles.bottomTabStyleLight : styles.bottomTabStyle}` A.Adds styles to the barStyle attribute based on the theme. B.It toggles between the themes.	**A**

C.Changes the themes based on the user preference. D.None of the above.	
In React Native, if we want to add JavaScript code inside the return statement of a render() function what do we use? A. [] B. {} C. () D. <>	**B**

End the quiz panel

FEEDBACK

- **Appreciate the student for their class**
- **Get them to play around with different ideas**

	Amazing work today! You get a "hats-off". Alright. See you in the next class.	*Make sure you have given at least 2 Hats Off during the class for:* 
Project Overview **Spectagram Stage - 8** **Goal of the Project:** In Class 88, we resolved the bug from the previous class and integrated firebase to save stories and read the		*The students engage with the teacher over the project.*

<table>
<tr><td colspan="2">stories from the database. In this project, you will practice the concepts learned in the class to integrate firebase with the Spectagram app to add posts to the database.

*This is a continuation project of 81 to 87, please make sure to finish that before attempting this one.

Story:
Jenny is a photographer. She wants to share pictures taken by her with others, at the same time she wants to create a space for others to share their talent too. She decided to create a social media app for her and all upcoming talents. She has asked for your help to create an App.

Guide Jenny to resolve the bug and integrate the firebase to add posts to the database.</td><td></td></tr>
<tr><td colspan="3">Teacher ends slideshow  </td></tr>
<tr><td colspan="3">Teacher Clicks  </td></tr>
<tr><td colspan="3">ADDITIONAL ACTIVITY</td></tr>
<tr><td>Additional Activities</td><td>Encourage the student to write reflection notes in their reflection journal using Markdown.

Use these as guiding questions:
• What happened today?
  ○ Describe what happened.
  ○ The code I wrote.
• How did I feel after the class?</td><td>The student uses the Markdown editor to write their reflections in a reflection journal.</td></tr>
</table>

	• What have I learned about programming and developing games? • What aspects of the class helped me? What did I find difficult?	

Activity	Activity Name	Links
Teacher Activity 1	Previous Class Code	https://github.com/pro-whitehatjr/ST-87-Solution
Teacher Activity 2	Reference Code	https://github.com/pro-whitehatjr/ST-88-Solution
Teacher Activity 3	Teacher Aid	https://drive.google.com/file/d/1WA1BQff4dmgv5BInU3f_imk4vlpvAyMa/view?usp=sharing
Teacher Reference visual aid link	Visual aid link	https://curriculum.whitehatjr.com/Visual+Project+Asset/PRO_VD/PRO_V3_C88_LITE_withcues.html
Teacher Reference In-class quiz	In-class quiz	https://s3-whjr-curriculum-uploads.whjr.online/40b6ae39-9de7-47a2-bb08-7307aee00bb5.pdf