ToDo List Application

React Native Mobile Application Report

Prepared by:

Mohamed Sameh Mohamed

Section: 4

March 15, 2025

Contents

1 Introduction

The ToDo List app is a modern, feature-rich task management application built with React Native and Expo. The app provides a clean and intuitive user interface with a focus on usability and aesthetics, designed to help users organize tasks, set priorities, add due dates, and more.

2 Features

- Create, edit, and delete tasks
- Mark tasks as complete/incomplete
- Categorize tasks with color-coded categories
- Set task priorities (high, medium, low)
- Add due dates to tasks
- Add notes to tasks
- Search and filter tasks
- Dark mode support
- Local storage for task persistence

3 Design Overview

The application provides an intuitive and visually appealing user interface with a focus on usability and aesthetics.

3.1 Key Screens and Features

3.1.1 Home Screen

- Task listing with swipeable actions
- Search and filter capabilities
- Category and priority filtering
- Task completion toggling

3.1.2 Task Form

- Add and edit tasks with comprehensive fields
- Set priority levels
- Assign categories
- Add due dates with date picker
- Include detailed notes

3.1.3 Task Details

- Comprehensive view of task information
- Quick actions for editing, deleting, and toggling completion
- Display of metadata such as creation date and due date

3.1.4 Theme Support

- Dark and light theme options
- Consistent color scheme across the application

4 Colors and Fonts

4.1 Color Scheme

The application uses a carefully selected color palette for both light and dark themes.

4.1.1 Light Theme Colors

Color Name	Hex Value	Description
Primary	#007AFF	Bright blue
Secondary	#5856D6	Rich purple
Background	#F9F9FB	Slightly off-white
Card	#FFFFFF	Pure white
Text	#1C1C1E	Near black
Text Light	#8A8A8E	Medium gray

Table 1: Light Theme Colors

4.1.2 Priority Colors

Color Name	Hex Value	Description
High Priority	#FF3B30	Vivid red
Medium Priority	#FF9500	Bright orange
Low Priority	#34C759	Vibrant green

Table 2: Priority Colors

4.1.3 Status Colors

Color Name	Hex Value	Description
Success	#34C759	Vibrant green
Error	#FF3B30	Vivid red
Warning	#FFCC00	Bright yellow
Info	#5AC8FA	Sky blue

Table 3: Status Colors

4.1.4 Dark Theme Colors

Color Name	Hex Value	
Primary	#0A84FF	Brighter blue
Secondary	#BF5AF2	Bright purple
Background	#121212	Deep black
Card	#1E1E1E	Dark gray
Text	#FFFFFF	White
Text Light	#8E8E93	Medium gray

Table 4: Dark Theme Colors

4.1.5 Category Colors

Category	Hex Value	Color
Personal	#4A6FFF	Blue
Work	#FF4D4F	Red
Shopping	#FAAD14	Orange
Health	#52C41A	Green

Table 5: Category Colors

4.2 Fonts and Typography

The application uses the system font with various sizes and weights:

4.2.1 Font Sizes

- Extra Small: 12px
- Small: 14px
- Medium: 16px
- Large: 18px
- Extra Large: 20px
- Double Extra Large: 24px
- Triple Extra Large: 30px

4.2.2 Font Weights

- Regular: 400
- Medium: 500
- Bold: 700

5 Code Implementation

5.1 Components Design and Styling

5.1.1 TaskItem Component

The TaskItem component represents a single task in the list with swipeable actions:

```
// Key parts of TaskItem.js
const TaskItem = ({ task, onToggle, onEdit, onDelete, onPress,
    getCategoryColor }) => {
  const { theme } = useTheme();

  // Swipe action rendering
  const renderRightActions = (progress, dragX) => {
    const trans = dragX.interpolate({
      inputRange: [-100, 0],
      outputRange: [0, 100],
      extrapolate: 'clamp',
    });

    return (
      <View style={styles.rightActions}>
        <Animated.View style={{ transform: [{ translateX: trans }] }}>
          <TouchableOpacity
            style={[styles.actionButton, { backgroundColor: theme.
    colors.info }]}
            onPress={() => onEdit(task)}
          >
            <Ionicons name="pencil" size={24} color="#fff" />
          </TouchableOpacity>
        </Animated.View>
        <Animated.View style={{ transform: [{ translateX: trans }] }}>
          <TouchableOpacity
            style={[styles.actionButton, { backgroundColor: theme.
    colors.error }]}
            onPress={() => onDelete(id)}
          >
            <Ionicons name="trash" size={24} color="#fff" />
          </TouchableOpacity>
        </Animated.View>
      </View>
    );
  };

  return (
    <Swipeable renderRightActions={renderRightActions}>
      <TouchableOpacity onPress={() => onPress(task)}>
        <View style={[styles.container, {...styling}]}>
          {/* Task content with checkbox, text, priority indicator */}
        </View>
      </TouchableOpacity>
    </Swipeable>
  );
};

// Styling for the TaskItem
const styles = StyleSheet.create({
  container: {
    flexDirection: 'row',
    borderLeftWidth: 4,
    borderBottomWidth: 0,
    padding: 16,
    marginBottom: 12,
    borderRadius: 12,
    shadowColor: '#000',
    shadowOffset: { width: 0, height: 2 },
    shadowOpacity: 0.08,
    shadowRadius: 4,
    elevation: 3,
  },
  // Additional styles...
});
```

Listing 1: Key parts of TaskItem.js

5.2 State Management Logic

5.2.1 TaskContext Implementation

The application uses React Context API for state management:

```
// Key parts of TaskContext.js
export const TaskContext = createContext();

export const TaskProvider = ({ children }) => {
  const [tasks, setTasks] = useState([]);
  const [isLoading, setIsLoading] = useState(true);
  const [categories, setCategories] = useState([
    { id: '1', name: 'Personal', color: '#4A6FFF' },
    { id: '2', name: 'Work', color: '#FF4D4F' },
    { id: '3', name: 'Shopping', color: '#FAAD14' },
    { id: '4', name: 'Health', color: '#52C41A' },
  ]);

  // Local storage integration with AsyncStorage
  const loadTasks = useCallback(async () => {
    try {
      setIsLoading(true);
      const storedTasks = await AsyncStorage.getItem(TASKS_STORAGE_KEY)
;

      if (storedTasks !== null) {
        const parsedTasks = JSON.parse(storedTasks);
        setTasks(parsedTasks);
      } else {
        setTasks([]);
      }
    } catch (error) {
      console.error('Error loading tasks:', error);
      setTasks([]);
    } finally {
      setIsLoading(false);
    }
  }, []);

  // CRUD operations for tasks
  const addTask = useCallback((task) => {
    if (!task || !task.text || task.text.trim() === '') {
      return null;
    }

    const newTask = {
      id: Date.now().toString(),
      createdAt: new Date().toISOString(),
      completed: false,
      text: task.text.trim(),
      priority: task.priority || 'medium',
      categoryId: task.categoryId || '1',
      dueDate: task.dueDate || null,
      notes: task.notes || ''
    };

    setTasks(prevTasks => {
      const updatedTasks = [newTask, ...prevTasks];
      saveTasks(updatedTasks);
      return updatedTasks;
    });

    return newTask;
  }, [saveTasks]);

  // Additional task operations...

  return (
    <TaskContext.Provider value={{
      tasks,
      categories,
      isLoading,
      addTask,
      updateTask,
      deleteTask,
      toggleTaskCompletion,
      // Additional methods...
    }}>
      {children}
    </TaskContext.Provider>
  );
};
```

Listing 2: Key parts of TaskContext.js

5.3 Screen Implementation

5.3.1 HomeScreen Structure

The HomeScreen serves as the main hub of the application:

```
// Key parts of HomeScreen.js
const HomeScreen = () => {
  const { theme, themeMode } = useTheme();
  const {
    tasks,
    categories,
    addTask,
    updateTask,
    deleteTask,
    toggleTaskCompletion,
    // Additional methods...
  } = useTask();

  const [isAddModalVisible, setIsAddModalVisible] = useState(false);
  const [isEditModalVisible, setIsEditModalVisible] = useState(false);
  const [isDetailsModalVisible, setIsDetailsModalVisible] = useState(
   false);
  const [currentTask, setCurrentTask] = useState(null);

  // Task operation handlers
  const handleAddTask = (newTask) => {
    // Task validation and processing...
    const taskToAdd = {
      text: newTask.text.trim(),
      priority: newTask.priority || 'medium',
      categoryId: newTask.categoryId || '1',
      notes: newTask.notes || '',
      dueDate: newTask.dueDate || null
    };

    const addedTask = addTask(taskToAdd);

    // Result handling...
    return addedTask;
  };

  // Additional handlers for editing, deleting, and toggling tasks...

  return (
    <SafeAreaView style={[styles.container, { backgroundColor: theme.colors.background }]}>
      <StatusBar
        barStyle={themeMode === 'dark' ? 'light-content' : 'dark-content'}
        backgroundColor={theme.colors.background}
      />

      <Header title={`My Tasks (${tasks.length})`} />

      <View style={styles.content}>
        <TaskList
          onTaskPress={handleTaskPress}
          onTaskToggle={handleToggleTask}
          onTaskEdit={handleEditTask}
          onTaskDelete={handleDeleteTask}
        />
      </View>

      <FloatingActionButton onPress={() => {
        setCurrentTask(null);
        setIsAddModalVisible(true);
      }} />

      {/* Modals for adding, editing, and viewing tasks */}
      <TaskForm visible={isAddModalVisible} /* props... */ />
      <TaskForm visible={isEditModalVisible} /* props... */ />
      <TaskDetails visible={isDetailsModalVisible} /* props... */ />
    </SafeAreaView>
  );
};
```

Listing 3: Key parts of HomeScreen.js

5.4 Form Implementation

5.4.1 TaskForm Component

The TaskForm handles both adding new tasks and editing existing ones:

```
// Key parts of TaskForm.js
const TaskForm = ({ visible, onClose, onSubmit, initialTask = {}, categories }) => {
  const { theme } = useTheme();
  const textInputRef = useRef(null);
  const isEditMode = initialTask && initialTask.id;

  // Form state
  const [text, setText] = useState('');
  const [category, setCategory] = useState('');
  const [priority, setPriority] = useState('medium');
  const [dueDate, setDueDate] = useState(null);
  const [notes, setNotes] = useState('');
  const [showDatePicker, setShowDatePicker] = useState(false);

  // Initialize form when modal becomes visible
  useEffect(() => {
    if (visible) {
      if (isEditMode) {
        // Load values from initialTask for editing
        setText(initialTask.text || ’’);
        setCategory(initialTask.categoryId || ’’);
        setPriority(initialTask.priority || ’medium’);
        setDueDate(initialTask.dueDate ? new Date(initialTask.dueDate) : null);
        setNotes(initialTask.notes || ’’);
      } else {
        // Set defaults for new task
        setText(’’);
        setCategory(categories[0]?.id || ’’);
        setPriority(’medium’);
        setDueDate(null);
        setNotes(’’);
      }

      // Focus the text input
      setTimeout(() => {
        if (textInputRef.current) {
          textInputRef.current.focus();
        }
      }, 300);
    }
  }, [visible, isEditMode]);

  // Form submission
  const handleSubmit = () => {
    if (!text.trim()) {
      Alert.alert(’Error’, ’Please enter a task’);
      return;
    }

    // Create task object
    const newTask = {
      ...(isEditMode ? { id: initialTask.id } : {}),
      text: text.trim(),
      categoryId: category,
      priority: priority,
      dueDate: dueDate ? dueDate.toISOString() : null,
      notes: notes.trim(),
      completed: isEditMode ? initialTask.completed : false,
      createdAt: isEditMode ? initialTask.createdAt : new Date().toISOString(),
    };

    // Submit and handle result
    const result = onSubmit(newTask);

    if (result) {
      setText(’’);
      onClose();
    }
  };

  // Form UI rendering
  return (
    <Modal visible={visible} transparent animationType="slide">
      <KeyboardAvoidingView behavior={Platform.OS === 'ios' ? 'padding'
    : 'height'}>
        <View style={styles.centeredView}>
          <View style={[styles.modalView, {/* styling */}]}>
            {/* Form fields for task text, category, priority, due date
   , notes */}
            {/* Submit and cancel buttons */}
          </View>
        </View>
      </KeyboardAvoidingView>
    </Modal>
  );
};
```

Listing 4: Key parts of TaskForm.js

6 Conclusion

The ToDo List app demonstrates a well-structured React Native application with a focus on user experience and modern design principles. The application leverages:

1. **Component-Based Architecture**: Clean separation of UI components
2. **Context API for State Management**: Centralized task and theme management
3. **AsyncStorage for Persistence**: Reliable local storage of tasks
4. **Responsive Design**: Adaptable UI for different screen sizes
5. **Theming System**: Consistent styling with dark/light mode support
6. **Swipeable Actions**: Intuitive gesture-based interactions
7. **Form Validation**: User-friendly input handling and validation

The careful attention to design details, color selection, and component styling results in a polished and professional user interface that enhances the overall user experience.