OCR PDF Viewer Development Documentation

This document outlines the complete development process for the OCR PDF Viewer application, a React Native application that combines PDF viewing with OCR (Optical Character Recognition) capabilities and features persistent page storage.

Setup and Environment Configuration

Step 1: Install Node.js and npm

React Native requires Node.js and npm (or yarn). 1. Download and install Node.js from the official website: https://nodejs.org/ 2. Verify installation by running: `bash node -v npm -v` If both return version numbers, you have installed them successfully.

Step 2: Install Java Development Kit (JDK)

React Native requires Java for Android development. 1. Download and install JDK (Java 17 or later) from: https://adoptium.net/ 2. Verify installation by running: `bash java -version`

Step 3: Install Android Studio (for Android Development)

If you want to run your React Native app on an Android emulator or a physical Android device: 1. Download Android Studio from: https://developer.android.com/studio 2. Install it and open SDK Manager inside Android Studio. 3. Under SDK Platforms, select Android 12.0 (API Level 31) or higher. 4. Under SDK Tools, install: - Android SDK Build-Tools - Android Emulator - Android SDK Command-line Tools 5. Set up Android environment variables: - Find your SDK path inside Android Studio at File → Settings → Appearance & Behavior → System Settings → Android SDK. - Open your terminal and add the following to your ~/.bashrc, ~/.bash_profile, or ~/.zshrc: `bash export ANDROID_HOME=$HOME/Library/Android/sdk export PATH=$ANDROID_HOME/emulator:$ANDROID_HOME/platform-tools:$PATH` 6. Restart the terminal and verify: `bash adb --version`

Step 4: Install React Native CLI

For a clean installation, install the React Native CLI globally:

```
npm install -g react-native-cli
```

Check if it's installed:

```
react-native -v
```

Step 5: Create a New React Native Project with TypeScript

Run the following command in your terminal:

```
npx react-native init OCRPdfViewer --template react-native-template-typescript
```

This will create a TypeScript-based React Native project in a folder called OCRPdfViewer. Navigate into your project:

```
cd OCRPdfViewer
```

Project Setup

Step 6: Install Required Dependencies

Install the core dependencies for PDF viewing, OCR functionality, and navigation:

```
npm install react-native-pdf
npm install @react-native-ml-kit/text-recognition
npm install react-native-svg
npm install react-native-view-shot
npm install @react-native-async-storage/async-storage
npm install @react-navigation/native @react-navigation/native-stack
npm install @react-native-documents/picker
npm install react-native-fs
npm install react-native-blob-util
npm install react-native-safe-area-context react-native-screens
```

Step 7: Configure Android Permissions

Edit the `android/app/src/main/AndroidManifest.xml` file to add required permissions:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

Step 8: Create Navigation Structure

1. Create the navigation type definitions:

```
// src/navigation/types.ts
export type RootStackParamList = {
  PdfUpload: undefined;
  PdfViewer: {
    pdfPath: string;
  };
};
```

2. Create the app navigator:

```tsx
// src/navigation/AppNavigator.tsx
import React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';
import { RootStackParamList } from './types';
import PdfUploadScreen from '../screens/PdfUploadScreen';
import PdfViewerScreen from '../screens/PdfViewerScreen';

const Stack = createNativeStackNavigator<RootStackParamList>();

const AppNavigator: React.FC = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator
        initialRouteName="PdfUpload"
        screenOptions={{
          headerShown: true,
          headerStyle: {
            backgroundColor: '#4a90e2',
          },
          headerTintColor: '#fff',
          headerTitleStyle: {
            fontWeight: 'bold',
          },
        }}
      >
        <Stack.Screen
          name="PdfUpload"
          component={PdfUploadScreen}
          options={{ title: 'PDF Upload' }}
        />
        <Stack.Screen
          name="PdfViewer"
          component={PdfViewerScreen}
          options={{ title: 'PDF Viewer' }}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
};

export default AppNavigator;
```

3. Update App.tsx to use the navigator:

```tsx
// App.tsx
import React from 'react';
```

```tsx
import { SafeAreaProvider } from 'react-native-safe-area-context';
import AppNavigator from './src/navigation/AppNavigator';

function App(): React.JSX.Element {
  return (
    <SafeAreaProvider>
      <AppNavigator />
    </SafeAreaProvider>
  );
}

export default App;
```

PDF Upload Screen Implementation

Step 9: Create the PDF Upload Screen

Create a screen for selecting and uploading PDF files:

```tsx
// src/screens/PdfUploadScreen.tsx
import React, { useState, useEffect } from 'react';
import {
  SafeAreaView,
  StyleSheet,
  Text,
  View,
  TouchableOpacity,
  Alert,
} from 'react-native';
import { pick, DocumentPickerResponse } from '@react-native-documents/picker';
import RNFS from 'react-native-fs';
import { useNavigation } from '@react-navigation/native';
import { NativeStackNavigationProp } from '@react-navigation/native-stack';
import { RootStackParamList } from '../navigation/types';

type PdfUploadScreenNavigationProp = NativeStackNavigationProp<
  RootStackParamList,
  'PdfUpload'
>;

const PdfUploadScreen: React.FC = () => {
  const navigation = useNavigation<PdfUploadScreenNavigationProp>();
  const [previouslyUploadedPdf, setPreviouslyUploadedPdf] = useState<string | null>(null);

  // Check for previously saved PDF
  useEffect(() => {
    const checkSavedPdf = async () => {
```

```
    try {
      const appFolderPath = RNFS.DocumentDirectoryPath + '/pdfs';
      const exists = await RNFS.exists(appFolderPath);

      if (exists) {
        const files = await RNFS.readDir(appFolderPath);
        const pdfFiles = files.filter(file => file.name.endsWith('.pdf'));

        if (pdfFiles.length > 0) {
          setPreviouslyUploadedPdf(pdfFiles[0].path);
        }
      }
    } catch (error) {
      console.error('Error checking for saved PDFs:', error);
    }
  };

  checkSavedPdf();
}, []);

const pickPdfDocument = async () => {
  try {
    const result = await pick({
      type: ['application/pdf'],
    });

    if (result.length > 0) {
      handleSelectedPdf(result[0]);
    }
  } catch (err) {
    console.error('Error picking document:', err);
    Alert.alert('Error', 'Failed to pick PDF document');
  }
};

const handleSelectedPdf = async (document: DocumentPickerResponse) => {
  try {
    // Create app folder if it doesn't exist
    const appFolderPath = RNFS.DocumentDirectoryPath + '/pdfs';
    const exists = await RNFS.exists(appFolderPath);

    if (!exists) {
      await RNFS.mkdir(appFolderPath);
    }

    // Save the file to app's documents directory
```

```
      const destPath = `${appFolderPath}/${document.name}`;

      // Check if document has a valid URI
      if (!document.uri) {
        throw new Error('Document URI is undefined');
      }

      // Copy file from picked location to app storage
      await RNFS.copyFile(document.uri, destPath);

      // Navigate to the PDF viewer
      navigation.navigate('PdfViewer', { pdfPath: destPath });
    } catch (error) {
      console.error('Error saving PDF:', error);
      Alert.alert('Error', 'Failed to save PDF document');
    }
  };

  const openPreviouslyUploadedPdf = () => {
    if (previouslyUploadedPdf) {
      navigation.navigate('PdfViewer', { pdfPath: previouslyUploadedPdf });
    }
  };

  return (
    <SafeAreaView style={styles.container}>
      <View style={styles.content}>
        <Text style={styles.title}>OCR PDF Viewer</Text>
        <Text style={styles.subtitle}>Select a PDF document to begin</Text>

        <TouchableOpacity
          style={styles.button}
          onPress={pickPdfDocument}
        >
          <Text style={styles.buttonText}>Select PDF Document</Text>
        </TouchableOpacity>

        {previouslyUploadedPdf && (
          <TouchableOpacity
            style={[styles.button, styles.secondaryButton]}
            onPress={openPreviouslyUploadedPdf}
          >
            <Text style={styles.buttonText}>Open Previously Uploaded PDF</Text>
          </TouchableOpacity>
        )}
      </View>
```

```
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#f5f5f5',
  },
  content: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    padding: 20,
  },
  title: {
    fontSize: 24,
    fontWeight: 'bold',
    color: '#333',
    marginBottom: 10,
  },
  subtitle: {
    fontSize: 16,
    color: '#666',
    marginBottom: 30,
    textAlign: 'center',
  },
  button: {
    backgroundColor: '#4a90e2',
    paddingVertical: 12,
    paddingHorizontal: 24,
    borderRadius: 6,
    marginBottom: 16,
    width: '100%',
    alignItems: 'center',
  },
  secondaryButton: {
    backgroundColor: '#5cb85c',
  },
  buttonText: {
    color: 'white',
    fontSize: 16,
    fontWeight: '600',
  },
});
```

```
export default PdfUploadScreen;
```

PDF Viewer and OCR Implementation

Step 10: Create Types for OCR Data

Define the types needed for OCR data:

```
// Types for OCR functionality
type DrawPath = {
  path: string;
  page: number;
};

type Point = {
  x: number;
  y: number;
};

type IntersectingElement = {
  blockIndex: number;
  lineIndex: number;
  elementIndex: number;
};
```

Step 11: Create the PDF Viewer Screen

Create the main PDF viewer screen with OCR capabilities:

```
// src/screens/PdfViewerScreen.tsx
import React, { useState, useRef, useCallback, useEffect } from 'react';
import {
  StyleSheet,
  View,
  Text,
  Dimensions,
  Alert,
  TouchableOpacity,
  SafeAreaView,
  ScrollView,
  ActivityIndicator,
  Image,
  Modal,
  LayoutChangeEvent,
  GestureResponderEvent,
  TextInput,
} from 'react-native';
```

```tsx
import { RouteProp } from '@react-navigation/native';
import Pdf from 'react-native-pdf';
import { RootStackParamList } from '../navigation/types';
import Svg, { Rect, Path } from 'react-native-svg';
import { captureRef } from 'react-native-view-shot';
import TextRecognition, { TextRecognitionResult } from '@react-native-ml-kit/text-recognitio
import AsyncStorage from '@react-native-async-storage/async-storage';

// Constants for storage keys
const LAST_VIEWED_PAGE_KEY = 'LAST_VIEWED_PAGE';
const PDF_FILE_PATH_KEY = 'PDF_FILE_PATH';

// Types definition
// ... add types as defined in Step 10 ...

const PdfViewerScreen: React.FC<PdfViewerScreenProps> = ({ route }) => {
  // State declarations, refs, and functionality implementation
  // ... implement the full PDF viewer screen ...

  return (
    <SafeAreaView style={styles.container}>
      {/* Header, PDF Viewer, Controls, etc. */}
      {/* ... implement the JSX for the PDF viewer UI ... */}
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  // ... implement all the styles for the PDF viewer ...
});

export default PdfViewerScreen;
```

Step 12: Implement OCR Functionality

Add the OCR text recognition capabilities:

```tsx
// Inside the PdfViewerScreen component

// Function to perform OCR on current page
const performOCR = useCallback(async () => {
  // If this page has already been analyzed and we're not forcing a re-analysis, don't do it
  if (analyzedPages.includes(currentPage) && !isDrawingEnabled) {
    return;
  }
```

```
// Prevent multiple OCR processes running simultaneously
if (isProcessing) {
  console.log('OCR already in progress, skipping');
  return;
}

try {
  setIsProcessing(true);
  setOcrResult(null);
  setSelectedText(null);
  setOcrComplete(false);
  setIntersectingElements([]);

  // Capture the PDF view as an image
  if (!pdfViewRef.current) {
    throw new Error("PDF view reference is not available");
  }

  // Add a small delay to ensure the view is fully rendered
  await new Promise(resolve => setTimeout(resolve, 1000));

  // Capture the full page image
  const fullImageUri = await captureRef(pdfViewRef, {
    format: 'jpg',
    quality: 1.0,
    result: 'tmpfile',
  });

  console.log('Full page image captured at:', fullImageUri);
  setPageImageUri(fullImageUri);

  // Get image dimensions for proper scaling
  await new Promise((resolve, reject) => {
    Image.getSize(
      fullImageUri,
      (width, height) => {
        console.log(`Original image dimensions: ${width}x${height}`);
        setImageSize({ width, height });
        resolve(null);
      },
      (error) => {
        console.error('Failed to get image size:', error);
        reject(error);
      }
    );
  });
```

```
    // Process the image with ML Kit text recognition
    console.log('Starting text recognition on full page...');

    // Send the full image to ML Kit
    const result = await TextRecognition.recognize(fullImageUri);

    console.log('Recognition result:', JSON.stringify(result, null, 2));

    if (result) {
      setOcrResult(result);
      // Mark this page as analyzed
      setAnalyzedPages(prev => [...prev, currentPage]);
      console.log(`OCR complete: Found ${result.blocks.length} text blocks`);

      // Check if there are any existing paths to process
      const pathsOnCurrentPage = paths.filter(path => path.page === currentPage);
      if (pathsOnCurrentPage.length > 0) {
        // Process the latest path to find intersecting blocks
        findIntersectingBlocks(pathsOnCurrentPage[pathsOnCurrentPage.length - 1].path);
      }

      if (result.blocks.length === 0) {
        Alert.alert("OCR Complete", "No text was detected on this page.");
      }
    } else {
      Alert.alert("OCR Failed", "The text recognition process failed. Please try again.");
    }
  } catch (error) {
    console.error('OCR Error:', error);
    Alert.alert("OCR Error", `Error performing OCR: ${error instanceof Error ? error.message
  } finally {
    setIsProcessing(false);
    setOcrComplete(true);
  }
}, [currentPage, analyzedPages, paths, pdfViewRef, findIntersectingBlocks, isDrawingEnabled,
```

Step 13: Implement Drawing Functionality for Text Selection

Add capabilities to draw on the PDF and select text:

```
// Inside the PdfViewerScreen component

// Function to handle touch start
const handleTouchStart = (event: GestureResponderEvent) => {
  if (!isDrawingEnabled) return;
```

```
  try {
    const { locationX, locationY } = event.nativeEvent;
    setIsDrawing(true);
    // Start a new path at the touch location
    setCurrentPath(`M ${locationX} ${locationY}`);
  } catch (error) {
    console.error('Touch start error:', error);
  }
};

// Function to handle touch move
const handleTouchMove = (event: GestureResponderEvent) => {
  if (!isDrawingEnabled || !isDrawing) return;

  try {
    const { locationX, locationY } = event.nativeEvent;
    // Add line to path
    setCurrentPath(prevPath => `${prevPath} L ${locationX} ${locationY}`);
  } catch (error) {
    console.error('Touch move error:', error);
  }
};

// Function to handle touch end
const handleTouchEnd = () => {
  if (!isDrawingEnabled || !isDrawing) return;

  try {
    setIsDrawing(false);

    // Only save the path if it's not empty
    if (currentPath) {
      const newPath: DrawPath = {
        path: currentPath,
        page: currentPage,
      };

      // Replace any existing paths on the current page instead of adding
      setPaths(prevPaths => {
        // Filter out paths on the current page
        const pathsOnOtherPages = prevPaths.filter(path => path.page !== currentPage);
        // Add the new path
        return [...pathsOnOtherPages, newPath];
      });
```

```
      // If OCR is complete, determine which blocks intersect with the path
      if (ocrComplete && ocrResult) {
        findIntersectingBlocks(currentPath);
      }
    }

    // Reset current path
    setCurrentPath('');
  } catch (error) {
    console.error('Touch end error:', error);
    setIsDrawing(false);
    setCurrentPath('');
  }
};
```

Persistent Page Storage Implementation

Step 14: Implement Page Storage with AsyncStorage

Add functionality to remember the last viewed page:

```
// Inside the PdfViewerScreen component

// Function to save the current page state
const saveCurrentPageState = async (page: number, filePath: string) => {
  try {
    await AsyncStorage.setItem(LAST_VIEWED_PAGE_KEY, page.toString());
    await AsyncStorage.setItem(PDF_FILE_PATH_KEY, filePath);
    console.log(`Saved page ${page} for PDF: ${filePath}`);
  } catch (error) {
    console.error('Error saving page state:', error);
  }
};

// Add a function to handle page changes
const handlePageChange = (page: number, noOfPages: number) => {
  console.log('handlePageChange', page);
  setCurrentPage(page);

  if (totalPages !== noOfPages) {
    setTotalPages(noOfPages);
  }

  // Save the current page and PDF path to AsyncStorage for persistence
  saveCurrentPageState(page, pdfPath);
};
```

```
// Load the last viewed page when the component mounts
useEffect(() => {
  const loadLastViewedPage = async () => {
    try {
      const savedPdfPath = await AsyncStorage.getItem(PDF_FILE_PATH_KEY);
      const lastViewedPage = await AsyncStorage.getItem(LAST_VIEWED_PAGE_KEY);

      // Only restore page if the PDF path matches the current one
      if (savedPdfPath === pdfPath && lastViewedPage) {
        const page = parseInt(lastViewedPage, 10);
        console.log(`Restored to page ${page} for PDF: ${pdfPath}`);

        // Update the current page state - the Pdf component will handle navigation
        setCurrentPage(page);
        setInitialPage(page);
      }
    } catch (error) {
      console.error('Error loading last viewed page:', error);
    }
  };

  loadLastViewedPage();
}, [pdfPath]);
```

Step 15: Add Page Navigation Controls

Implement the navigation bar for PDF page control:

```
// Inside the PdfViewerScreen component's return statement

{/* Page Navigation Bar */}
{totalPages > 0 && (
  <View style={styles.pageNavBar}>
    <TouchableOpacity
      style={[styles.pageNavBarButton, currentPage <= 1 && styles.disabledButton]}
      onPress={() => {
        if (currentPage > 1) {
          const newPage = 1;
          setCurrentPage(newPage);
          saveCurrentPageState(newPage, pdfPath);
        }
      }}
      disabled={currentPage <= 1}
    >
      <Text style={styles.pageNavBarButtonText}>{'<<'}</Text>
    </TouchableOpacity>
```

```
<TouchableOpacity
  style={[styles.pageNavBarButton, currentPage <= 1 && styles.disabledButton]}
  onPress={() => {
    if (currentPage > 1) {
      const newPage = currentPage - 1;
      setCurrentPage(newPage);
      saveCurrentPageState(newPage, pdfPath);
    }
  }}
  disabled={currentPage <= 1}
>
  <Text style={styles.pageNavBarButtonText}>{'<'}</Text>
</TouchableOpacity>

<View style={styles.pageInputContainer}>
  <TextInput
    style={styles.pageInput}
    keyboardType="number-pad"
    returnKeyType="go"
    value={String(currentPage)}
    onChangeText={(text) => {
      const page = parseInt(text, 10);
      if (!isNaN(page) && page > 0 && page <= totalPages) {
        setCurrentPage(page);
        saveCurrentPageState(page, pdfPath);
      }
    }}
    maxLength={5}
  />
  <Text style={styles.pageInputLabel}>/ {totalPages}</Text>
</View>

<TouchableOpacity
  style={[styles.pageNavBarButton, currentPage >= totalPages && styles.disabledButton]}
  onPress={() => {
    if (currentPage < totalPages) {
      const newPage = currentPage + 1;
      setCurrentPage(newPage);
      saveCurrentPageState(newPage, pdfPath);
    }
  }}
  disabled={currentPage >= totalPages}
>
  <Text style={styles.pageNavBarButtonText}>{'>'}</Text>
</TouchableOpacity>
```

```
    <TouchableOpacity
      style={[styles.pageNavBarButton, currentPage >= totalPages && styles.disabledButton]}
      onPress={() => {
        if (currentPage < totalPages) {
          const newPage = totalPages;
          setCurrentPage(newPage);
          saveCurrentPageState(newPage, pdfPath);
        }
      }}
      disabled={currentPage >= totalPages}
    >
      <Text style={styles.pageNavBarButtonText}>{'>>'}</Text>
    </TouchableOpacity>
  </View>
)}
```

Running and Testing

Step 16: Start the Metro Bundler

Run the following command in your terminal:

```
npx react-native start
```

Step 17: Run on Android or iOS

In a new terminal window, run one of these commands:

For Android:

```
npx react-native run-android
```

For iOS:

```
npx react-native run-ios
```

Step 18: Testing the Application

1. **PDF Upload Test**:
   - Launch the app
   - Select a PDF document using the "Select PDF Document" button
   - Verify the PDF is copied to the app's storage
2. **PDF Viewing Test**:
   - Verify the PDF opens correctly
   - Swipe through pages and check that navigation works
3. **OCR Functionality Test**:
   - Enable drawing mode with the "Drawing: ON" button
   - Draw around text on the current page

- Verify OCR processing occurs (loading indicator appears)
- Check that text is identified correctly and highlighted on the page
- Verify the selected text appears in the text panel

4. **Persistent Storage Test**:
   - Navigate to a specific page (e.g., page 5)
   - Close the app completely
   - Relaunch the app and open the same PDF
   - Verify that the app returns to the last viewed page (page 5)

Troubleshooting Common Issues

Step 19: Resolve Common Problems

1. **Build Errors**:
   - Clean the project: `cd android && ./gradlew clean`
   - Run `npm install` to ensure all dependencies are installed
   - Check that Android SDK is properly configured
2. **OCR Performance Issues**:
   - Ensure the PDF is of good quality
   - Check that the ML Kit dependency is properly installed
   - Try analyzing pages with clearer text
3. **Storage Issues**:
   - Verify that AsyncStorage is properly set up
   - Check console logs for any errors related to storage operations
   - Clear the app's data and cache if persistent storage stops working

Step 20: Performance Optimization

1. **Reduce Unnecessary Re-Renders**:
   - Use `useCallback` and `useMemo` hooks for functions and values
   - Implement proper dependency arrays in useEffect hooks
2. **Optimize OCR Processing**:
   - Cache OCR results for previously analyzed pages
   - Use a debounce mechanism for drawing operations
3. **Improve PDF Loading**:
   - Implement progressive loading for large PDFs
   - Add feedback indicators for loading progress

Conclusion

Following these steps will result in a fully functional OCR PDF Viewer application with persistent page storage. The app allows users to view PDF documents, extract text using OCR technology, and automatically return to their last viewed page when reopening the app.

This development process combines several key React Native technologies: - PDF handling with react-native-pdf - Machine learning text recognition with

ML Kit - Persistent storage with AsyncStorage - Vector graphics with react-native-svg - Navigation with React Navigation

The resulting application provides a seamless user experience for viewing and interacting with PDF documents on mobile devices.