CSIT242 Mobile Application Development

LECTURE 4 – STORAGE

Outline

- Android - Storage options
- Android - SQLite
- iOS – Persistent storage
- iOS – Core Data

Storage - Android

Android – Data and file storage

- Android uses a file system that's similar to disk-based file systems on other platforms
- It provides several options for saving the app data
- The chosen solution depends on specific needs like
  - how much space the data requires
  - what kind of data need to be stored
  - whether the data should be private to the app or accessible to other apps and the user

Android – Data and file storage

- There are five storage options available on Android:
  - App-specific files – store files meant for app's use only (in dedicated directories within internal or external storage)
  - Shared storage - store files that your app intends to share with other app
    - Media - shareable media files (images, audio files, videos)
    - Documents and other files - other types of shareable content, including downloaded files
  - App preferences - store private primitive data in key-value pairs
  - Databases - store structured data in a private database

Android – Data and file storage

- There are two types of physical storage locations:
  - Internal file storage - always available on devices (more reliable)
  - External file storage – bigger capacity
- Scoped storage
  - Android 10+ (API level 29+) apps are given scoped access into external storage
  - Apps have access only to the app-specific directory on external storage and specific types of media (MediaStore) that the app has created

Android – Data and file storage

- Permissions for read/write access to external storage
  - earlier versions of Android (API level 28 or lower) apps needed to
    - declare the `READ_EXTERNAL_STORAGE`/ `WRITE_EXTERNAL_STORAGE` permission to access/write to any file outside the app-specific directories on external storage
  - recent versions of Android rely more on a file's purpose (not location)
    - apps are given access only to the areas of the device's file system that they actually use
  - Android 11: `MANAGE_EXTERNAL_STORAGE` permission
    - provides write access to files outside the app-specific directory and MediaStore

- The app-private data is not naturally accessible to other apps
- If the app-data or files need to be shared with other apps, Android provides:
  - FileProvider API - for sharing specific files with other apps
  - ContentProvider - give control of available read/write access to other apps

Android – App-specific files

- Internal storage directories - store sensitive data that only the app itself can access
  - dedicated location for storing persistent files
  - another location for storing cache data
  - on Android 10/API level 29+ these locations are encrypted

- External storage directories - store data meant for use only by the app
  - dedicated location for storing persistent files,
  - another location for storing cache data
  - it's possible for another app to access these directories (with the proper permissions)

Internal Storage	External Storage
It's always available	It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device
Files saved here are accessible only by the app	It's world-readable, so files saved here may be read outside of users control.
When the user uninstalls the app, the system removes all the app's files from internal storage	When the user uninstalls the app, the system removes the app's files from external storage

If the created files need to be accessible for other apps, the app should store these files in the shared storage part of external storage.

Android – App-specific files

- For the internal storage, `File` API can be used for access and store files
  - To access or store the persistent files in the app directory should be used `filesDir` or `cacheDir` property of a context object

```
File file = new File(context.getFilesDir(), filename);
File cacheFile = new File(context.getCacheDir(), filename);
```

- Other useful methods:
  - `openFileOutput()` – to write (as a stream) to a file within the directory
  - `openFileInput()` – to read a file as a stream
  - `fileList()` – get (return) an array containing the names of all files within the directory
  - `File.createTempFile(filename, null, context.getCacheDir())` - create a cached file
  - to delete the file from cache directory can be used:
    - `delete()` method on a `File` object that represents the file
    - `deleteFile(...)` method of the app's context, passing in the name of the file

Android – App-specific files

- The external storage, need to be verified - is physical volume accessible
- Method `Environment.getExternalStorageState()` can return the state:
  - `MEDIA_MOUNTED` - reading and writing is allowed for the files within external storage
  - `MEDIA_MOUNTED_READ_ONLY` - the files are read-only

- To access app-specific files from external storage, methods `getExternalFilesDir()/ getExternalCacheDir()` can be used
  - To access app-specific files from external storage

```
File appSpecificExternalDir = new
                        File(context.getExternalFilesDir(null), filename);
```

  - To add an app-specific file to the cache within external storage

```
File externalCacheFile = new File(context.getExternalCacheDir(),
                                                          filename);
```

  - To delete the file from cache directory can be used `delete()` method

Android – App-specific files

- For the media files (like images that should be used only by the app) that need to be stored in the app-specific external storage, it is important to use directory names provided by API constants (e.g. DIRECTORY_PICTURES) or store it in the root app-specific directory
  - These directory names ensure that the files are treated properly by the system

```
File file = new
    File(context.getExternalFilesDir(Environment.DIRECTORY_PICTURES),
                                                              name);
```

Android – Shared storage

- Shared storage should be used If the app’s user data need to be accessible to other apps
- The data in shared storage will be accessible and will not be deleted when the user uninstalls the app
- Android APIs support storing and accessing:
  - Media content (provides standard public directories/common locations for photos, music or audio files) using the platform's MediaStore API
  - Documents and other files (provides special directory for containing other file types – like PDF, EPUB) using the platform's Storage Access Framework
  - Datasets (on Android 11 - API level 30+ the system caches large datasets that multiple apps might use e.g. machine learning and media playback) using the BlobStoreManager API

Android – Shared storage

- The system automatically scans an external storage volume and adds media files to the `MediaStore` collections:
  - `MediaStore.Images` for images, including photographs and screenshots stored in the DCIM/ and Pictures/ directories
  - `MediaStore.Video` for videos stored in the DCIM/, Movies/, and Pictures/ directories
  - `MediaStore.Audio` for audio files stored in the Alarms/, Audiobooks/, Music/, Notifications/, Podcasts/, and Ringtones/ directories, and audio playlists in the Music/ or Movies/ directories
  - `MediaStore.Downloads` for downloaded files stored in the Download/ directory
    - This isn't available on Android 9 (API level 28) and lower
  - `MediaStore.Files` - the contents depend on
    - If scoped storage is enabled (Android 10+), shows only the photos, videos, and audio files that the app has created
    - If scoped storage is unavailable or not being used, shows all types of media files

Android – Shared storage

- When scoped storage is enabled for an app that targets Android 10 or higher
  - if the app creates a media file that is stored in the photos, videos, or audio files media collection (`MediaStore` collections), the app owns that file and has access to it
  - To access media files created by other apps create,
    - the appropriate storage-related permissions should be declared (`android.permission.READ_MEDIA_IMAGES`, `android.permission.READ_MEDIA_VIDEO`, `android.permission.READ_MEDIA_AUDIO`) and
    - the files must reside in MediaStore.Images, MediaStore.Video, or MediaStore.Audio
  - To retrieve unredacted EXIF metadata from photos, the `ACCESS_MEDIA_LOCATION` permission should be declared (in app's manifest) and request this permission at runtime
- To access a file within the `MediaStore.Downloads` collection created by other apps, the Storage Access Framework should be used

Because the ACCESS_MEDIA_LOCATION permission is requested at runtime, there is no guarantee that the app has access to unredacted Exif metadata from photos (for which explicit user consent is needed).

Android – Shared storage Storage Access Framework

- The app (Android 4.4/API level 19+) can use the Storage Access Framework (SAF) to interact with a documents provider
- The Storage Access Framework allows/ supports:
  - users to browse content from all document providers and select specific documents and other files
  - users to add, edit, save, and delete files on the provider
  - multiple user accounts and transient roots such as USB storage providers (appear only if the drive is plugged in)
- The SAF includes:
  - Document provider - A content provider that allows a storage service to reveal the files it manages
  - Client app - A custom app that invokes the action intent and receives the files returned by document providers
  - Picker - A system UI that lets users access the documents
- Cloud services can participate in this ecosystem by implementing a *DocumentsProvider* that encapsulates their services

Android – Shared storage
Storage Access Framework

- The use of SAF mechanism doesn't require any system permissions
  - The user is involved in selecting the files or directories that the app can access
  - These files remain on the device after your app is uninstalled
- Using the framework involves the following steps:
  - An app invokes an intent that contains a storage-related action (corresponding to specific use case)
  - The user sees a system picker (allowing to browse a documents provider and choose a location or document)
  - The app gains read and write access to a URI that represents the user's chosen location or document
  - Using this URI, the app can perform operations on the chosen location

Android – Shared storage Storage Access Framework

- The Storage Access Framework supports the following use cases for accessing files and other documents
  - Create a new file - ACTION_CREATE_DOCUMENT intent action allows users to save a file in a specific location
  - Open a document or file - ACTION_OPEN_DOCUMENT intent action allows users to select a specific document or file to open
  - Grant access to a directory's contents - ACTION_OPEN_DOCUMENT_TREE intent action, allows users to select a specific directory, granting the app access to all of the files and sub-directories within that directory

```
// creating a PDF document
private static final int CREATE_FILE = 1;
private void createFile(Uri pickerInitialUri) {
    Intent intent = new Intent(Intent.ACTION_CREATE_DOCUMENT);
    intent.addCategory(Intent.CATEGORY_OPENABLE);
    intent.setType("application/pdf");
    intent.putExtra(Intent.EXTRA_TITLE, "my_file.pdf");

    // Optionally, specify a URI for the desired initial location visible to user
    //when file chooser is shown
    intent.putExtra(DocumentsContract.EXTRA_INITIAL_URI, pickerInitialUri);

    startActivityForResult(intent, CREATE_FILE);
```

Android – Shared preferences

- `SharedPreferences` API can be used for storing small amount of data that doesn't require structure
- A `SharedPreferences` object points to a file containing key-value pairs and provides simple methods to read and write them
- Each `SharedPreferences` file is managed by the framework and can be private or shared
- Create a new shared preference file or access an existing one can be done by calling:
  - `getSharedPreferences()` - this method can be used (within any Context in the app) if there are multiple shared preference files (file name is sent as input parameter)
    - The name of shared preference files should be uniquely identifiable to the app
  - `getPreferences()` - this method can be used from an Activity if only one shared preference file for the activity needs to be used
    - Because this retrieves a default shared preference file that belongs to the activity, no need to supply a name

Android – Shared preferences

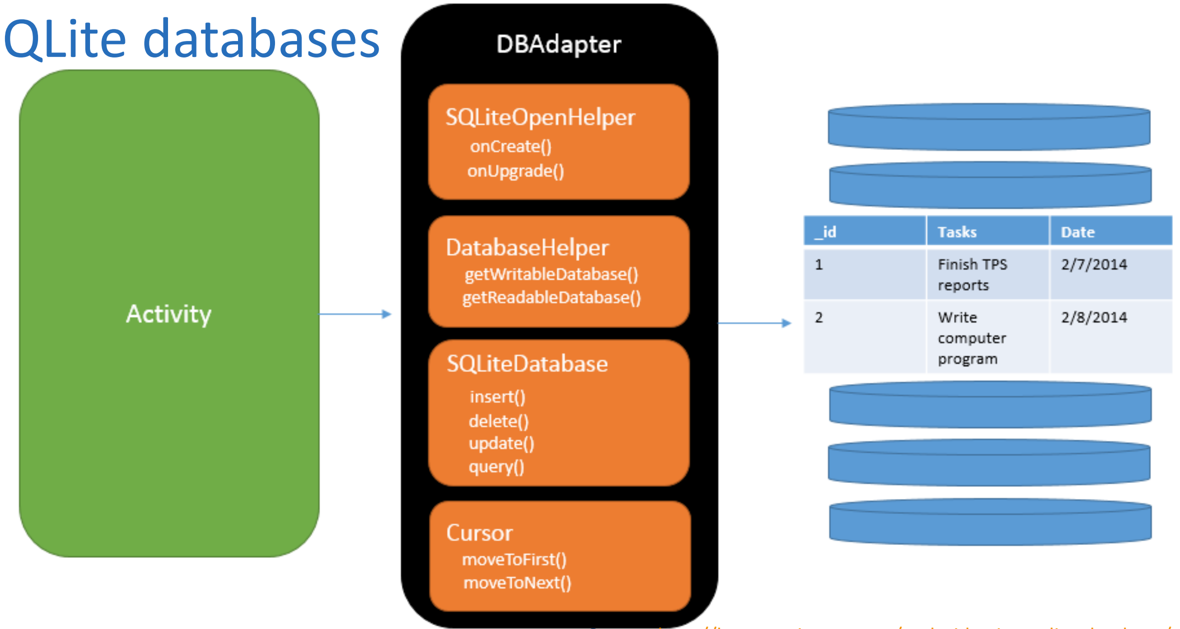
```
//write to shared preferences
SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE);
//SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
//use this inside the fragment
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString(getString(R.string.my_text), "this is my sp text");
editor.commit(); //apply()
```

```
//retrieve value from shared preferences

SharedPreferences sharedPref = getPreferences(Context.MODE_PRIVATE);
//SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
//use this inside the fragment
String myText = sharedPref.getString(getString(R.string.my_text), "default-null");
```

Android – Databases

- The use of databases is an essential aspect of most applications, ranging from applications that are almost entirely data driven, to those that simply need to store small amounts of data such as the prevailing score of a game
  - The importance of persistent data storage becomes even more evident with the ever-present risk that the Android runtime system will terminate an application component to free up resources
- Any database you create is accessible only by your app
- Android provides full support for SQLite databases



_id	Tasks	Date
1	Finish TPS reports	2/7/2014
2	Write computer program	2/8/2014

Source: http://lecturesnippets.com/android-using-sqlite-database/

Android – SQLite

- SQLite is an embedded, relational database management system (RDBMS)
- SQLite is referred to as embedded because it is provided in the form of a library that is linked into applications
  - As such, there is no standalone database server running in the background
- All database operations are handled internally within the application through calls to functions contained in the SQLite library
- The developers of SQLite have placed the technology into the public domain with the result that it is now a widely deployed database solution

Android – SQLite

- The Android SDK provides a Java based “wrapper” around the underlying database interface that consists of a set of classes that may be utilized within the Java code of an application to create and manage SQLite based databases
- The APIs needed to use a database on Android are available in the `android.database.sqlite` package
- `SQLiteOpenHelper` class can be used to manage database creation and version management
  - This class takes care of opening the database if it exists, creating it if it does not, and upgrading it as necessary
  - Transactions are used to make sure the database is always in a sensible state
  - This class makes it easy for *ContentProvider* implementations to defer opening and upgrading the database until first use, to avoid blocking application startup with long-running database upgrades

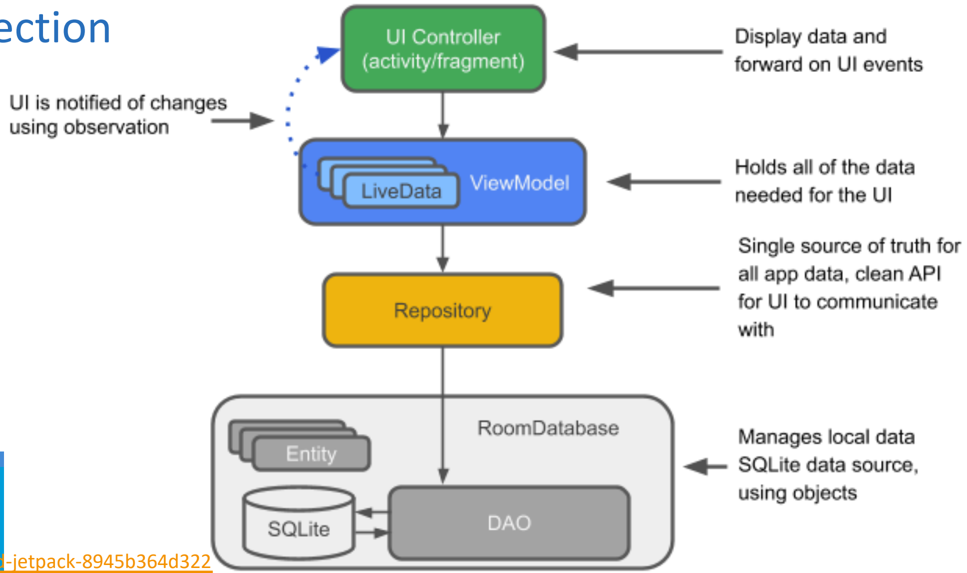
An example is provided in the lab materials.

Android – Room Persistence Library

- The SQLite APIs are fairly low-level and require a great deal of time and effort to use
  - There is no compile-time verification of raw SQL queries
  - As the schema changes, the affected SQL queries need to be updated manually (it can be time consuming and prone to errors)
  - A lots of boilerplate code is needed to convert between SQL queries and Java data objects

Android – Room Persistence Library

- The interaction with the databases can be done with the Room Persistence Library (recommended by Android)
- The Room provides an object-mapping abstraction layer allowing
  - fluent database access
  - Utilisation of the full power of SQLite
- The Room Persistence Library
  - helps in creating a cache of the app's data on a device that is running the app
  - the cache allows users to view a consistent copy of key information within the app, regardless of whether users have an internet connection



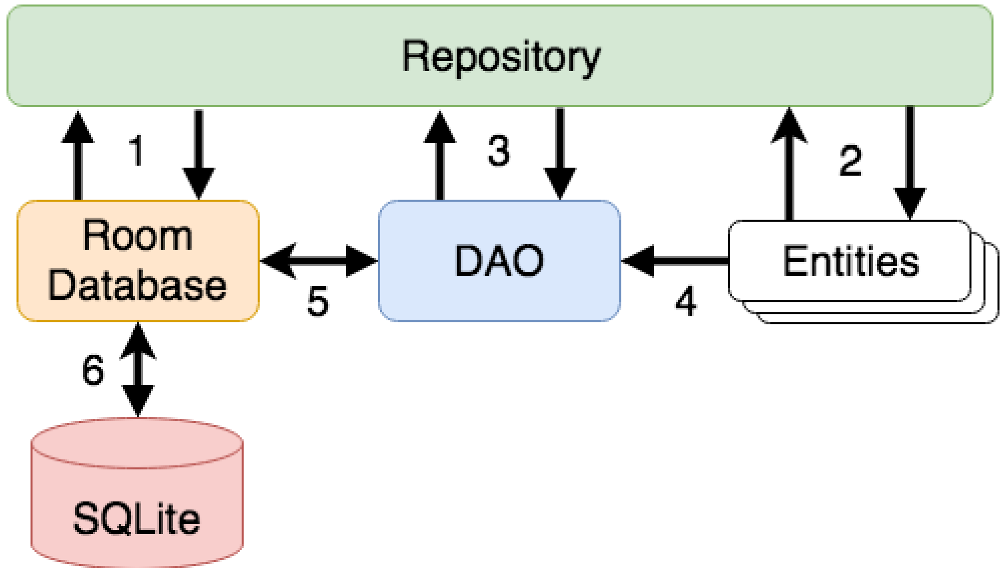
UI Controller (activity/fragment)
Display data and forward on UI events
UI is notified of changes using observation
LiveData
ViewModel
Holds all of the data needed for the UI
Repository
Single source of truth for all app data, clean API for UI to communicate with
RoomDatabase
Entity
SQLite
DAO
Manages local data SQLite data source, using objects

Source: https://medium.com/swlh/basic-implementation-of-room-database-with-repository-and-viewmodel-android-jetpack-8945b364d322

Android – Room Persistence Library

- There are 3 major components in Room:
  - Database - contains the database holder and serves as the main access point for the underlying connection to your app's persisted relational data.
  - Entity - represents a table within the database.
  - DAO (Data Access Objects) - contains the methods used for accessing the database.

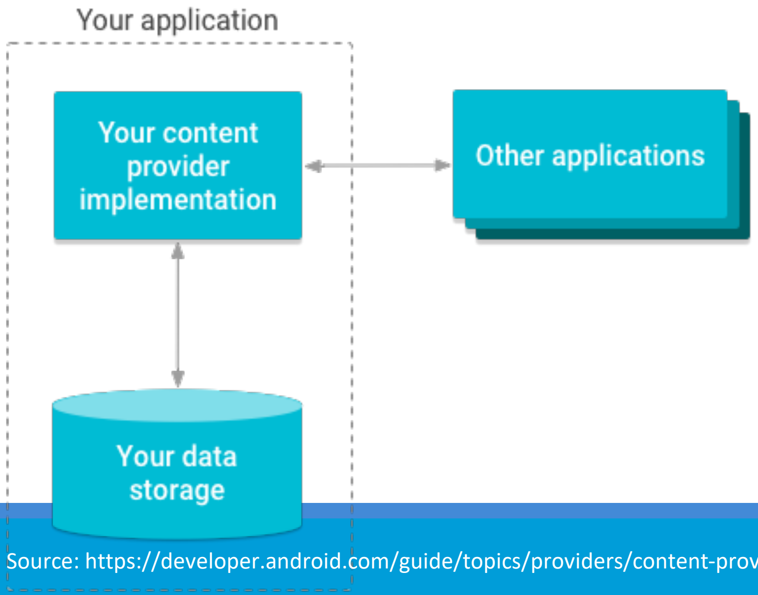
The repository module (shown in the figure) contains all of the code necessary for directly handling all data sources used by the app.
This avoids the need for the UI controller and ViewModel to contain code that directly accesses sources such as databases or web services



Source: Android Studio 3.5 Essentials - Java Edition, Neil Smyth, Payload Media Inc., 2019

Android – Content providers

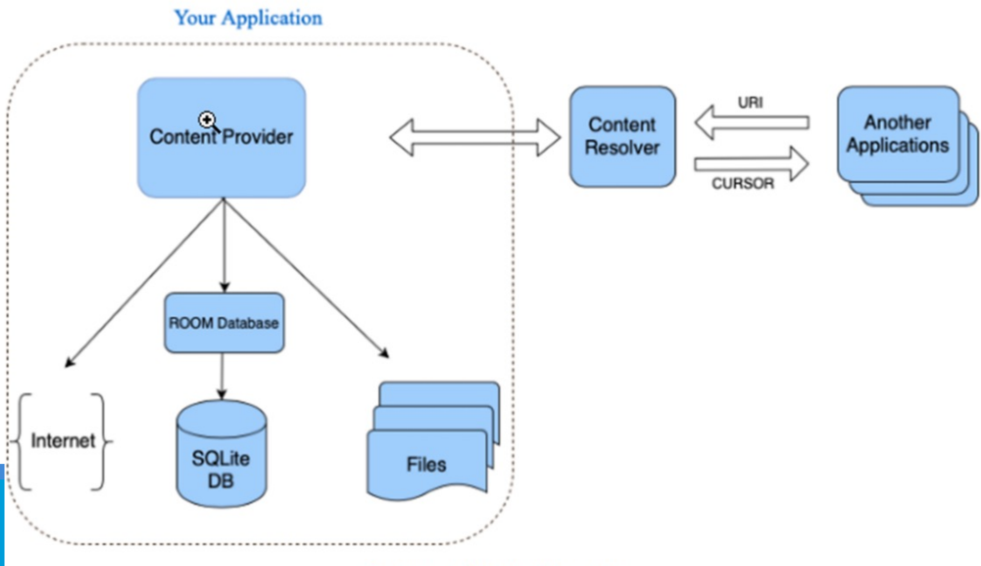
- Content providers can help an application to manage access to
  - data stored by itself (the app)
  - stored by other apps
  - provide a way to share data with other apps
- Content providers encapsulate the data, and provide mechanisms for defining data security
  - They are the standard interface that connects data in one process with code running in another process



Source: https://developer.android.com/guide/topics/providers/content-providers

Android – Content providers

- Implementing a content provider has many advantages
  - Provide a level of abstraction that allows making modifications to the application data storage implementation without affecting other existing applications that rely on the data access
  - Offer granular control over the permissions for data access:
    - restrict access to a content provider only from within the application
    - grant blanket permission to access data from other applications
    - configure different permissions for reading and writing data



Overview of Content Provider

Storage - iOS

iOS – Persistent Storage

- iOS has a file system much like any other operating system allowing applications to store persistent data on behalf of the user
- The iOS file system provides a directory based structure into which files can be created and organized
- Since the introduction of iOS 5 the iOS app developer has two options in terms of storing data:
    - Files and data can be stored on the file system of the local device or
    - Remotely using Apple’s iCloud service
- The iOS platform is responsible for ensuring that the applications cannot interfere with each other, both in terms of memory usage and data storage
- Applications can read and write only to their own directories and usually cannot create or modify files or directories outside of these directories

iOS – App’s Sandbox

- App’s sandbox is a limited portion of the device persistent storage dedicated to the app alone
- Every app is seeing only its own sandbox
  - App’s sandbox is a safe place for storing app’s data
  - Enhance privacy – other apps cannot access files belonging to the app
  - The sandbox (the data) will be deleted if the user uninstalls/deletes the app
- The sandbox contains standard directories as
  - Application Support directory - app's private directory in its own sandbox (user cannot see this directory)
  - Documents directory - if the app supports file sharing (the user can see and modify this directory)
    - if the app is working with common types of file (that the user might obtain elsewhere), it is appropriate for the app to support file sharing
  - Temp directory – for storing temporary files

iOS – App’s Sandbox

- iOS provides an elaborate mechanism for allowing app to share files
  - File sharing lets the user manipulate the contents of the app’s Documents directory
- `UIDocumentInteractionController`
  - allows the user to give permission for document sharing between apps
  - support document preview

Documents and Directories

- The Foundation Framework provides three classes for working with files and directories:
  - `FileManager` class can be used to perform basic operations(e.g. creating, moving, reading and writing) on files, reading and setting file attributes, and directory operations
  - `FileHandle` class is provided for performing lower level operations on files - seeking to a specific position in a file, reading and writing a file's contents, appending data to an existing file
  - `Data` class provides a useful storage buffer into which the contents of a file may be read, or from which dynamically stored data may be written to a file

When an application starts, the current working directory is the file system's root directory represented by a single /.
Path names that begin with a / are said to be absolute path names in that they specify a file system location relative to the system root directory.
Paths that do not begin with a slash are interpreted to be relative to a current working directory.

Documents and Directories

- Working with `FileManager` object and directories
  - To obtain a reference to the application's default file manager instance

```
let filemgr = FileManager.default
```

  - To identifies the (path to) current working directory – app's root directory

```
let currentPath = filemgr.currentDirectoryPath
```

  - To require the path to the Documents directory located in the application's home directory

```
let dirPaths = filemgr.urls(for: .documentDirectory, in: .userDomainMask)
```

  - To obtain the path to the current application's Documents directory

```
let docsDir = dirPaths[0].path
```

  - To create object (referenced by tmpDir) that will contain the path to the temporary directory for the application

```
let tmpDir = NSTemporaryDirectory()
```

Documents and Directories

- Some useful methods for working with directories (`FileManager` class):
  - `changeCurrentDirectoryPath()` - can change the path of current working directory to the specified path
  - `createDirectory(atPath:, withIntermediateDirectories:, attributes:)` - creates a new directory on an iOS device at a specified path (with `at:` can use URL as input parameter)
  - `removeItem(atPath:)` - removes an existing directory (at the specified path or with `at:` can use URL)
  - `contentsOfDirectory(atPath:)` - obtaines a list (array object) of the files and subdirectories contained within a specified directory
  - `attributesOfItem(atPath:)` - obtaines the attributes of a file or directory

Working with files

- Some useful methods (`FileManager` class) for working with files:
  - `fileExists(atPath:)` - checks whether a specified file (or directory) already exists
  - `isReadableFile(atPath:)`, `isWritableFile(atPath:)`, `isExecutableFile(atPath:)`, `isDeletableFile(atPath:)` - file access permission/control
  - `contentsEqual(atPath: , andPath: )` - compares the contents of two files
  - `moveItem(atPath:, toPath:)` - moves the file or directory at the specified to a new location synchronously (can be used for renaming)
  - `copyItem(atPath:, toPath:)` – copies a file
  - `removeItem(atPath:)` - removes a file
  - `contents(atPath: )` - reads and return contents of a file (storing in a Data object)

Working with files using `FileHandle` class

- The `FileHandle` class provides a range of methods designed to provide a more advanced mechanism for working with files
- A `FileHandle` object can be created when the file is opening for reading, writing or updating
  - For example
    - Initializer `init(forReadingAtPath: String)`

      ```
      let file: FileHandle? = FileHandle(forReadingAtPath: filePath1)
      ```

      - returns a file handle initialized for reading the file at the specified path
      - if an attempt to open a file fails, the method returns nil
    - `init(forReadingFrom: URL)`
      - returns a file handle initialized for reading the file at the specified URL ,or nil if no file exists at url

- Subsequently the file need to be closed (using the `close()` method)

There are multiple methods in the File Handle class that can be used for getting/creating the file handle for reading/writing/updating a file … Depending on the method the input parameter can be the file path or url. Also, when error occurs some methods return/throw an error object.

Working with files using `FileHandle` class

- `FileHandle` objects maintain a pointer to the current position in a file - offset
  - When the file is opened - the offset is set to 0 (the beginning of the file)
  - To perform operations at different locations in a file the offset needs to be set at certain value - `seekToEnd()` or `seek(toOffset:)`
  - The current offset may be identified using the `offset()` method
- The `read(upToCount:)` and `readToEnd()` methods read the contents of the file (starting at the location of the offset and read specific number of bytes or up to the end of file) and return it as Data object
- The `write(Data)` method writes the data contained in a `Data object` to the `file` (starting at the current location of the offset)
  - The method will overwrites any existing data in the file at the corresponding location
- The `truncate(atOffset:)` truncates or extends the file represented by the receiver to a specified offset within the file and puts the file pointer at that position
  - To delete the entire contents of a file, specify an offset of 0

Working with files using `FileHandle` class

- Example

```
let fileMng: FileManager = FileManager.default
let dirPaths = fileMng.urls(for: .documentDirectory, in: .userDomainMask)
let dataFile = dirPaths[0].appendingPathComponent("myFile.txt").path
if fileMng.fileExists(atPath: dataFile){
do{
    try fileMng.removeItem(atPath: dataFile)
   } catch let error{ print(error.localizedDescription) }
}
let data = ("The quick brown fox jumped over the lazy dog" as NSString).data(using:
                                                     String.Encoding.utf16.rawValue)!
fileMng.createFile(atPath: dataFile, contents: data, attributes: nil)
---
if fileMng.isWritableFile(atPath: dataFile){
    print("can write into the file")
}else{    print(" can’t write into the file ") }
```

Working with files using `FileHandle` class

```
let file: FileHandle? = FileHandle(forReadingAtPath: dataFile)
// file with text: The quick brown fox jumped over the lazy dog
if file == nil {
    print("File open failed")
} else {
do {
   let databuffer1 = try file?.readToEnd()
   print(String(decoding: databuffer1!, as: UTF8.self))
   print (try file1?.offset() ?? 0)
   try file?.close()
    let file1: FileHandle? = FileHandle(forWritingAtPath: dataFile)
    let data = ("black cat" as NSString).data(using: String.Encoding.utf16.rawValue)
    try file1?.seek(toOffset: 10)
    file1?.write(data) //The quick black cat jumped over the lazy dog
   try file1?.truncate(atOffset: 20)
   try file1?.close()
   } catch let error{print(error.localizedDescription) }
```

iCloud storage

- The iOS SDK provides support for four types of iCloud-based storage
- **iCloud document storage** allows an app to store data files and documents on iCloud in a special area reserved for that app
  - Once stored, these files may be subsequently retrieved from iCloud storage by the same app at any time as long as it is running on a device with the user's iCloud credentials configured
- **iCloud key-value storage** service allows small amounts of data packaged in key/value format to be stored in the cloud
  - This service is intended to provide a way for the same application to synchronize user settings and status when installed on multiple devices
- **iCloud Drive storage -** every Apple user account has cloud area into which files may be stored and accessed from different devices and apps
  - Files stored on iCloud Drive can also be accessed via the built-in Files app on iOS and different apps on other platforms or via the iCloud.com web portal

iCloud storage

- **CloudKit data storage** provides applications with access to the iCloud servers hosted by Apple and provides an easy to use way to store, manage and retrieve data and other asset types in a structured way
  - provides a way for users to store private data and access it from multiple devices, and also for the developer to provide data that is publicly available to all the users of an application

- In order for an application to be able to use iCloud services it must be
  - code signed with an App ID (with iCloud support enabled)
  - configured with specific entitlements to enable one or more of the iCloud storage services
  - Both of these tasks can be performed within the app *Capabilities* screen within Xcode

`UIDocument` class

- The iOS `UIDocument` class is recommended for working with iCloud-based file and document storage
- It is designed to provide an easy to use interface for the creation and management of documents and content
- `UIDocument` is an abstract class
  - The applications must create a subclass of `UIDocument` and override at least two methods:
    - `contents(forType:)` method is responsible for gathering the data to be written and returning it in the form of a `Data` or `FileWrapper` object
    - `load(fromContents:ofType:)` – the method is passed the content that has been read from the file by the `UIDocument` subclass and is responsible for loading that data into the application's internal data model

`UIDocument` class

- `UIDocument` and its underlying architecture provide:
  - Asynchronous reading and writing of data on a background queue
    - The application's responsiveness to users is thus unaffected while reading and writing operations taking place
  - Coordinated reading and writing of document files that is automatically integrated with cloud services
  - Support for discovering conflicts between different versions of a document (if that occurs)
  - Safe-saving of document data by writing data first to a temporary file and then replacing the current document file with it
  - Automatic saving of document data at opportune moments (this mechanism includes support for dealing with suspend behaviours)

Core Data

- iOS application code could directly manipulate the database using SQLite C API calls to construct and execute SQL statements
  - This is a good approach but requires knowledge of SQL and can lead to some complexity in terms of writing code and maintaining the database structure
  - This complexity is further compounded by the non-object-oriented nature of the SQLite C API functions
- In recognition of these shortcomings, Apple introduced the Core Data Framework
- Core Data is essentially a framework that places a wrapper around the SQLite database (and other storage environments)
  - enables the developer to work with data in terms of Swift objects without requiring any knowledge of the underlying database technology

Core Data

- Core Data saves application's permanent data for offline use, cache temporary data, and add undo functionality to the app
- Core Data is a framework that can be used to manage the model layer objects in the application
- It provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence
- It allows data organised by the relational entity–attribute model to be serialized into XML, binary, or SQLite stores
    - The data can be manipulated using higher level objects representing entities and their relationships
    - Core Data interfaces directly with SQLite, insulating the developer from the underlying SQL

Core Data

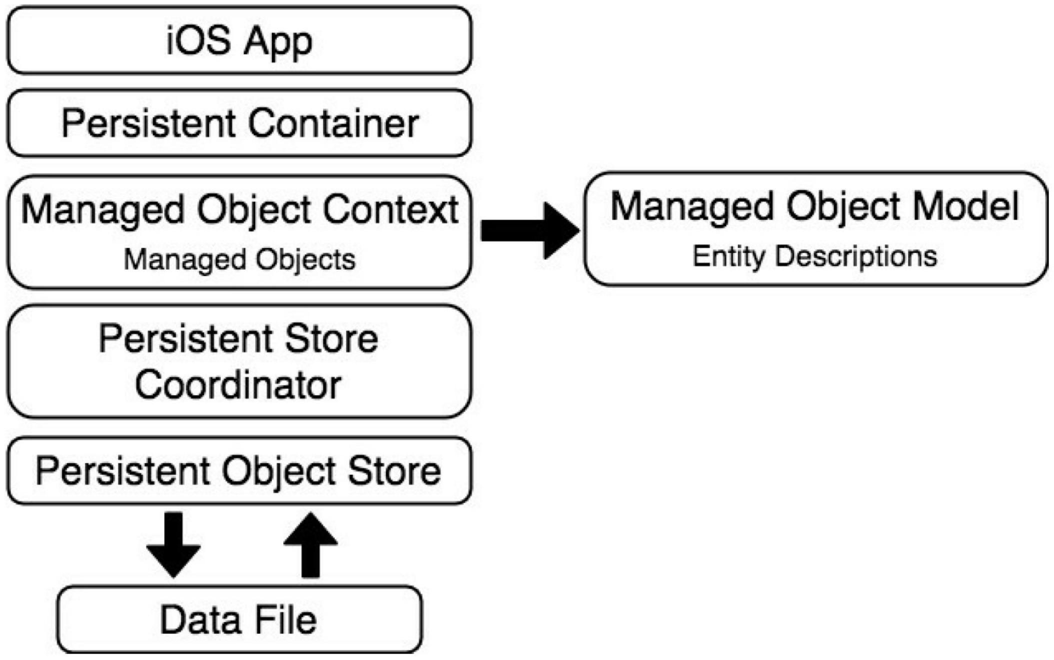
- Core Data typically decreases by 50 to 70 percent the amount of code written to support the model layer
- Core Data describes data with a high level data model expressed in terms of entities and their relationships plus fetch requests that retrieve entities meeting specific criteria
- Code can retrieve and manipulate this data on a purely object level without having to worry about the details of storage and retrieval
- The controller objects available in Interface Builder can retrieve and manipulate these entities directly
- Core Data consists of a number of framework objects, integrated to provide the data storage functionality

Core Data Stack

- The Core Data stack is a collection of framework objects that provide the data storage functionality
- The stack consists of four primary objects:
  - the persistent container (`NSPersistentContainer`)
  - the managed object context (`NSManagedObjectContext`)
  - the managed object model (`NSManagedObjectModel`)
  - the persistent store coordinator (`NSPersistentStoreCoordinator`)

Core Data Stack

- The iOS based application sits on top of the stack and interacts with the managed data objects handled by the managed object context
- Persistent container encapsulates the Core Data stack in the application and provides access to the managed object context
- Persistent container simplifies the creation and management of the Core Data stack by handling the creation of the managed object model, persistent store coordinator, and the managed object context



Source: iOS 12 App Development Essentials, Neil Smyth, Payload Media, Inc., 2018

Core Data Stack

- **Managed objects** are the objects that are created by the application code to store data
- A managed object can be seen as a row or a record in a relational database table
  - For each new record to be added, a new managed object must be created to store the data
  - Retrieved data are returned in the form of managed objects
- Managed objects are instances of the `NSManagedObject` class (or its subclass)
- These objects are contained and maintained by the managed object context

Core Data Stack

- **Managed object context** manipulate and track changes to managed objects
  - It is the object that the application interacts with
- The context maintains the status of the objects in relation to the underlying data store and manages the relationships between managed objects defined by the managed object model
- All interactions with the underlying database are held temporarily within the context until the context is instructed to save the changes
  - the changes are passed down through the Core Data stack and written to the persistent store

Core Data Stack

- **Managed Object Model** defines how the data models are defined – defines a concept referred to as entities
- A managed object model maintains a mapping between each of its entity objects and a corresponding managed object class for use with the persistent storage mechanisms in the Core Data framework
  - Entities define the data model for managed objects as a class description defines a blueprint for an object instance
  - An entity is analogous to the schema that defines a table in a relational database

Core Data Stack

- Entity has a set of attributes that define the data to be stored in managed objects derived from that entity
- Entities can also contain:
  - Relationships - they refer to how one data object relates to another (similar as those in other relational database systems): one-to-one, one-to-many or many-to-many
  - Fetched property - “weak, one way relationships” or “loosely coupled relationships” that allow properties of one data object to be accessed from another data object
  - Fetch request - A predefined query that can be referenced to retrieve data objects based on defined predicates

Core Data Stack

- **Persistent store coordinator** is responsible for coordinating access to multiple persistent object stores
- When multiple stores are required, the coordinator presents these stores to the upper layers of the Core Data stack as a single store

Core Data Stack

- **Persistent object store** refers to the underlying storage environment in which data are stored when using Core Data
- Core Data supports one memory-based and three disk-based persistent store:
  - SQLite
  - XML,
  - binary
- By default, the iOS SDK will use SQLite as the persistent store
- The code will make the same calls to the same Core Data APIs to manage the data objects required by the application (regardless of the choice of persistent store)

Connecting the Model to Views

- In iOS, `NSFetchedResultsController` is used to to manage the results of a Core Data fetch request and to display data to the user
- `NSFetchedResultsController` provides the interface between Core Data and `UITableView` objects
  - `UITableView` is designed to handle high-volume data displays
- When you use Core Data with a `UITableView`–based layout, the `NSFetchedResultsController` (for the data) is typically initialized by the `UITableViewController` instance that will utilize that data
  - This initialization can take place in the `viewDidLoad` or `viewWillAppear` methods, or at another logical point in the life cycle of the view controller

Property Lists

- A property list is a structured data representation used by Core Foundation as a convenient way to store, organize, and access standard types of data
  - Similar context with Shared Preferences in Android
  - Usually referred to as a “plist”
  - Some applications on iOS use property lists in their Settings bundle to define the list of options displayed to users
- Property lists are based on an abstraction for expressing simple hierarchies of data
- The items of data in a property list are of a limited number of types
  - String, Number, Boolean, Date
  - arrays—indexed collections of values
  - dictionaries—collections of values each identified by a key

Property Lists

- Property lists organize data into named values and lists of values using several Core Foundation types (like CFString, CFNumber, CFBoolean, …)
- The property list programming interface allows converting hierarchically structured combinations of these basic types to and from standard XML
  - The XML data can be saved to disk and later used to reconstruct the original Core Foundation objects
- The property lists
  - can be used for data that consists primarily of strings and numbers
  - Cannot/should not be used with large blocks of binary data

Resources

- Android Studio 4.1 Essentials - Java Edition, Neil Smyth, Payload Media Inc., 2020
- Android Studio 3.5 Essentials - Java Edition, Neil Smyth, Payload Media Inc., 2019
- Head First Android Development, Dawn Griffiths and David Griffiths, O'Reilly Media, 2017
- Learn Android Studio 4: Efficient Java-Based Android Apps Development, Ted Hagos, Apress, 2nd ed. edition, 2020.
- https://developer.android.com/guide/topics/data/data-storage
- https://developer.android.com/training/data-storage/sqlite.html
- https://developer.android.com/topic/libraries/architecture/room
- https://developer.android.com/reference/android/os/Environment
- https://developer.android.com/training/data-storage/shared/media
- https://developer.android.com/training/data-storage

Resources

- iOS 17 App Development Essentials, Neil Smyth, Payload Media, Inc., 2023.
- Programming iOS 14 - Dive Deep into Views, View Controllers, and Frameworks, Matt Neuburg, O'Reilly Media Inc., 2020.
- https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/index.html
- https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html

- Additional resources on how to create CRUD operations with Core Data can be found in the below links.

  https://medium.com/@ankurvekariya/core-data-crud-with-swift-4-2-for-beginners-40efe4e7d1cc

  https://manvendrasinghrathore.medium.com/core-data-crud-with-swift-for-beginners-cb9b5b9c02eb