



Basics of Mobile Application Development

Storage



iOS method

Core Data

- Core Data
 - A framework that you use to manage the model layer
 - Provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence
- Essentially an object graph is created, which can be backed with a DBS
 - Often in SQL, but it can be in XML, or memory based DBS as well

Core Data

- Properties
 - Maintenance of change propagation, including maintaining the consistency of relationships among objects
 - Lazy loading of objects, partially materialized futures (faulting), and copy-on-write data sharing to reduce overhead
 - Automatic validation of property values
 - Schema migration tools that simplify schema changes and allow you to perform efficient in-place schema migration
 - Grouping, filtering, and organizing data in memory and in the user interface
 - Automatic support for storing objects in external data repositories
 - Sophisticated query compilation.
 - Instead of writing SQL, you can create complex queries by associating an NSPredicate object with a fetch request

Creating a model

- Data Model
 - The database entities and the objects are bounded together
- File | New | File
 - New file for the data model
- Section
 - Core Data
- Template
 - Data Model

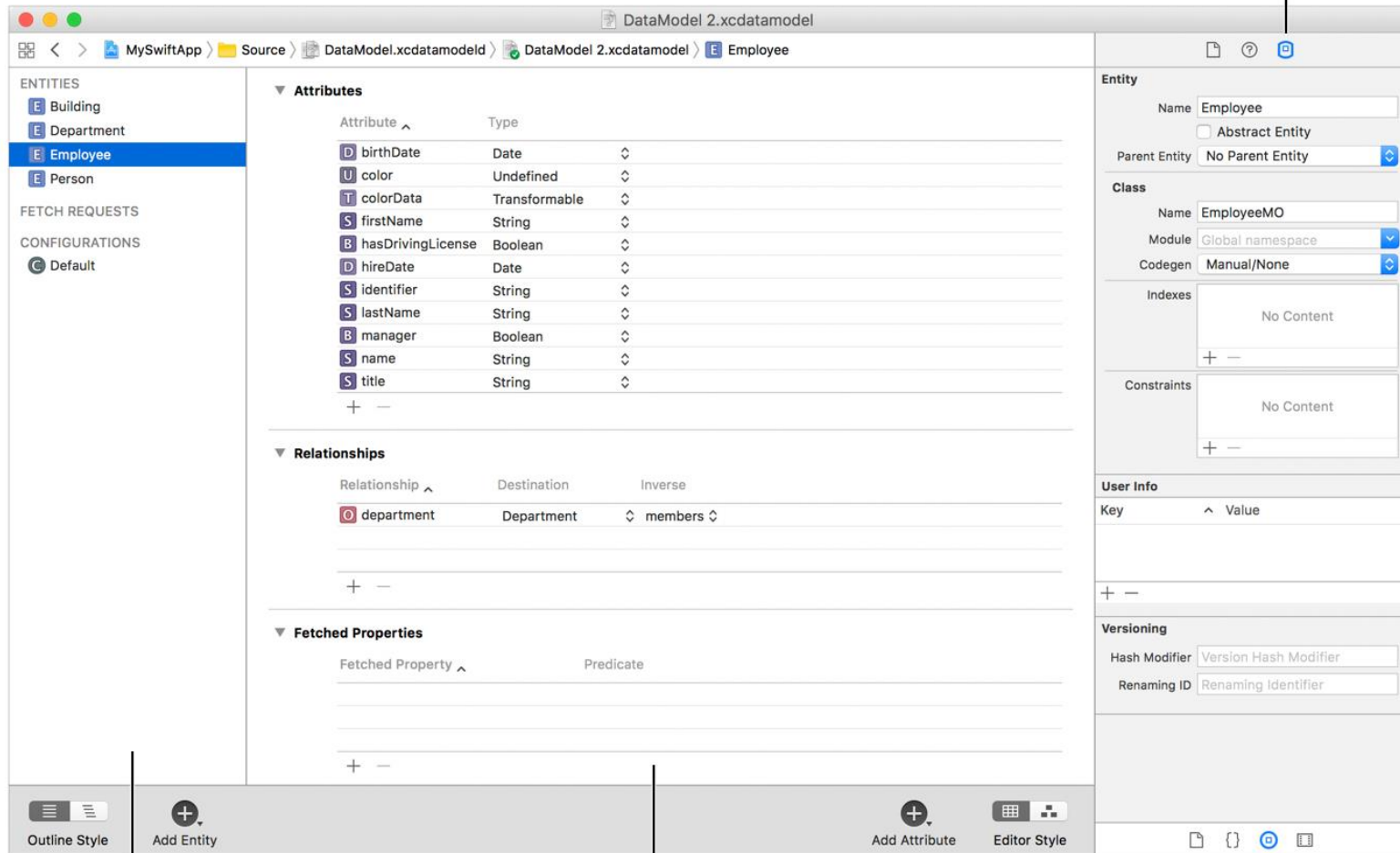
Parts of data model

- Data model uses the following notation
 - Entity
 - As it has been used in the notation of Database Management
 - Attributes
 - Name and type
 - Relationships
 - Between entities

Parts of data model

- For each entity a class is defined which is a subclass of `NSManagedObject`
- Attributes corresponds to the fields of classes
 - The attributes can be accessed through the methods of the `NSManagedObject`
- You can specify the data model by using a graphical editor

Data Model inspector



The screenshot shows the Data Model inspector application interface. The main window is titled "DataModel 2.xcdatamodel" and displays the "Employee" entity. The interface is divided into three main sections: the Navigator Area on the left, the Editor Area in the center, and the Data Model inspector on the right.

Navigator Area: Contains a list of entities (Building, Department, Employee, Person) and a list of fetch requests (Default). The "Employee" entity is selected.

Editor Area: Displays the "Attributes" and "Relationships" for the "Employee" entity. The "Attributes" section shows a table with columns "Attribute" and "Type". The "Relationships" section shows a table with columns "Relationship", "Destination", and "Inverse".

Attribute	Type
birthDate	Date
color	Undefined
colorData	Transformable
firstName	String
hasDrivingLicense	Boolean
hireDate	Date
identifier	String
lastName	String
manager	Boolean
name	String
title	String

Relationship	Destination	Inverse
department	Department	members

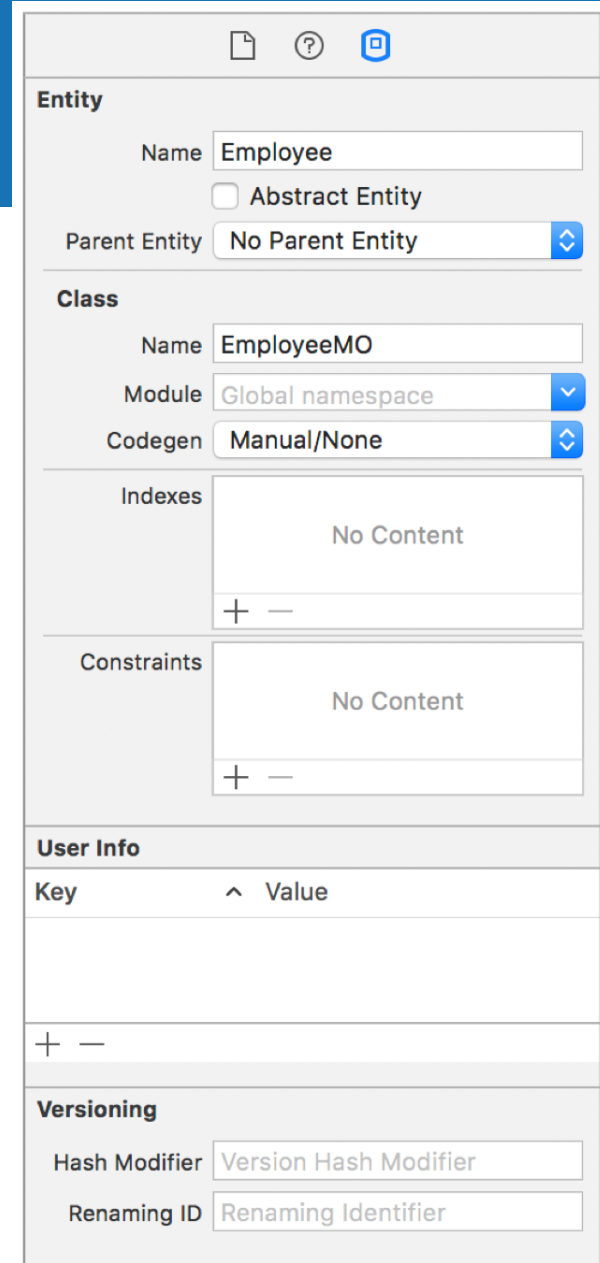
Data Model inspector: Contains settings for the "Employee" entity, including Name, Parent Entity, Class, Module, Codegen, Indexes, Constraints, User Info, and Versioning.

Navigator Area

Editor Area

Entity – example

- Note that the entity name and the class name are not the same
- The entity structure in the data model does not need to match the class hierarchy
- When an entity is abstract then we cannot create instances of that entity
- Entity inheritance works in a similar way to class inheritance



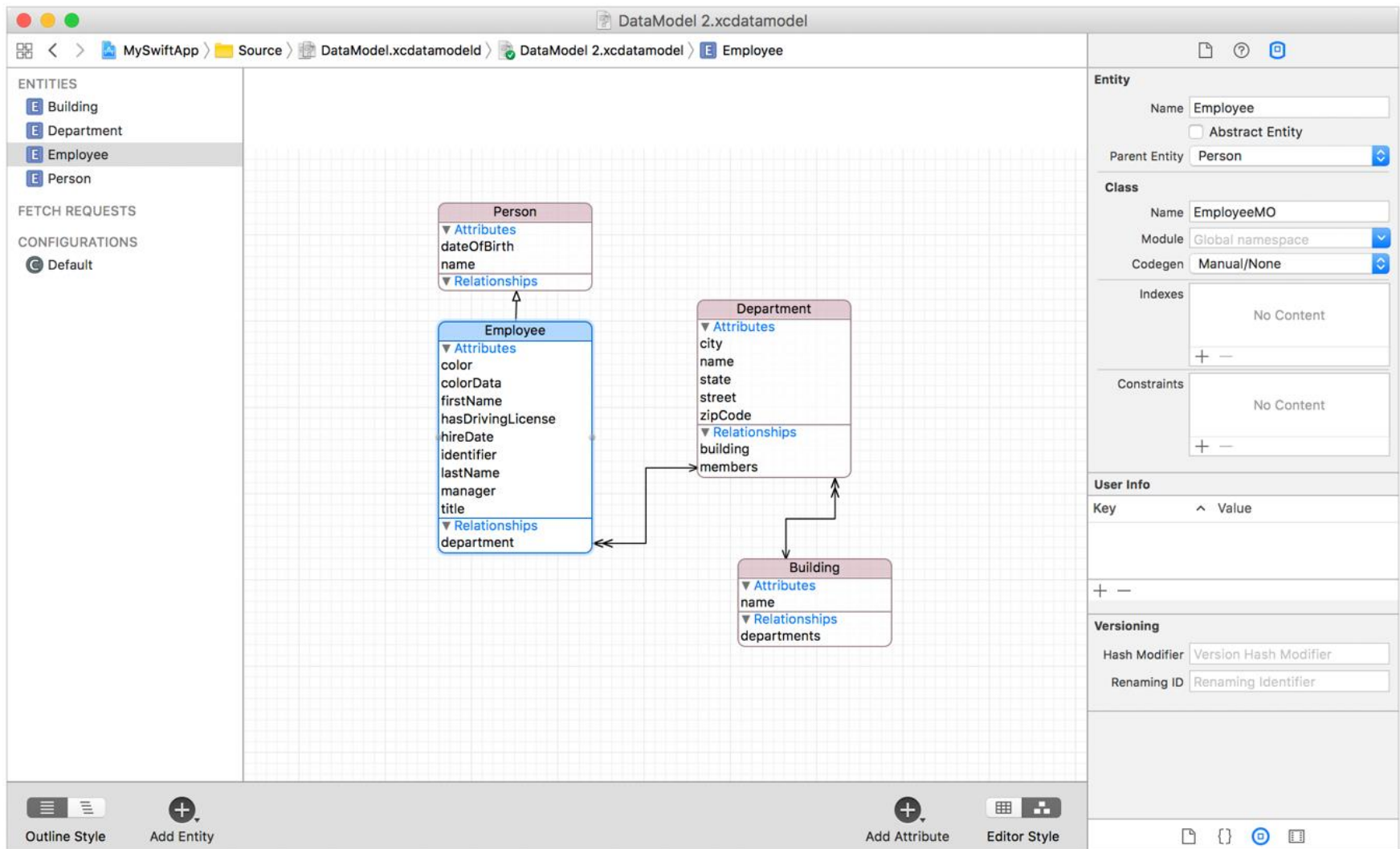
The screenshot shows a configuration window for an entity and class. The 'Entity' section has a 'Name' field set to 'Employee', an 'Abstract Entity' checkbox, and a 'Parent Entity' dropdown set to 'No Parent Entity'. The 'Class' section has a 'Name' field set to 'EmployeeMO', a 'Module' dropdown set to 'Global namespace', and a 'Codegen' dropdown set to 'Manual/None'. Below these are expandable sections for 'Indexes' and 'Constraints', both currently showing 'No Content'. At the bottom, there is a 'User Info' table with columns 'Key' and 'Value', and a 'Versioning' section with 'Hash Modifier' set to 'Version Hash Modifier' and 'Renaming ID' set to 'Renaming Identifier'.

Key	Value
-----	-------

Versioning

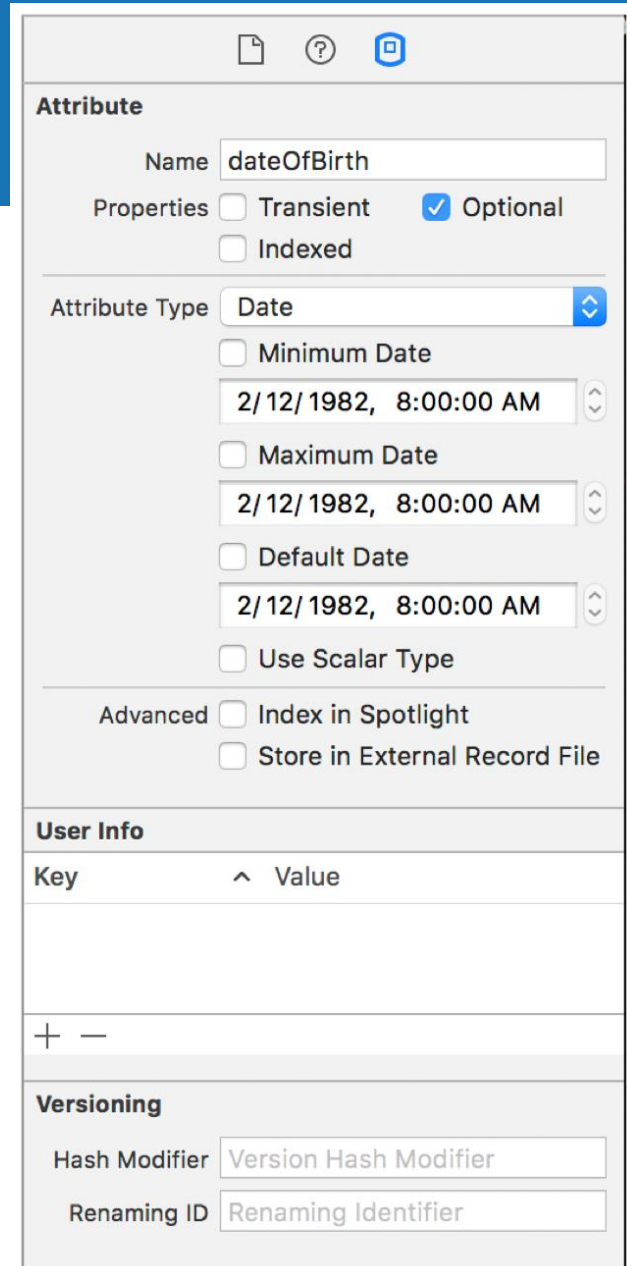
Hash Modifier: Version Hash Modifier

Renaming ID: Renaming Identifier



Attributes

- Core Data supports
 - String (NSString)
 - Date (NSDate)
 - Integer (NSNumber)
 - ...
- An attribute can be optional
 - NULL in a database is not equivalent to an empty string or empty data blob



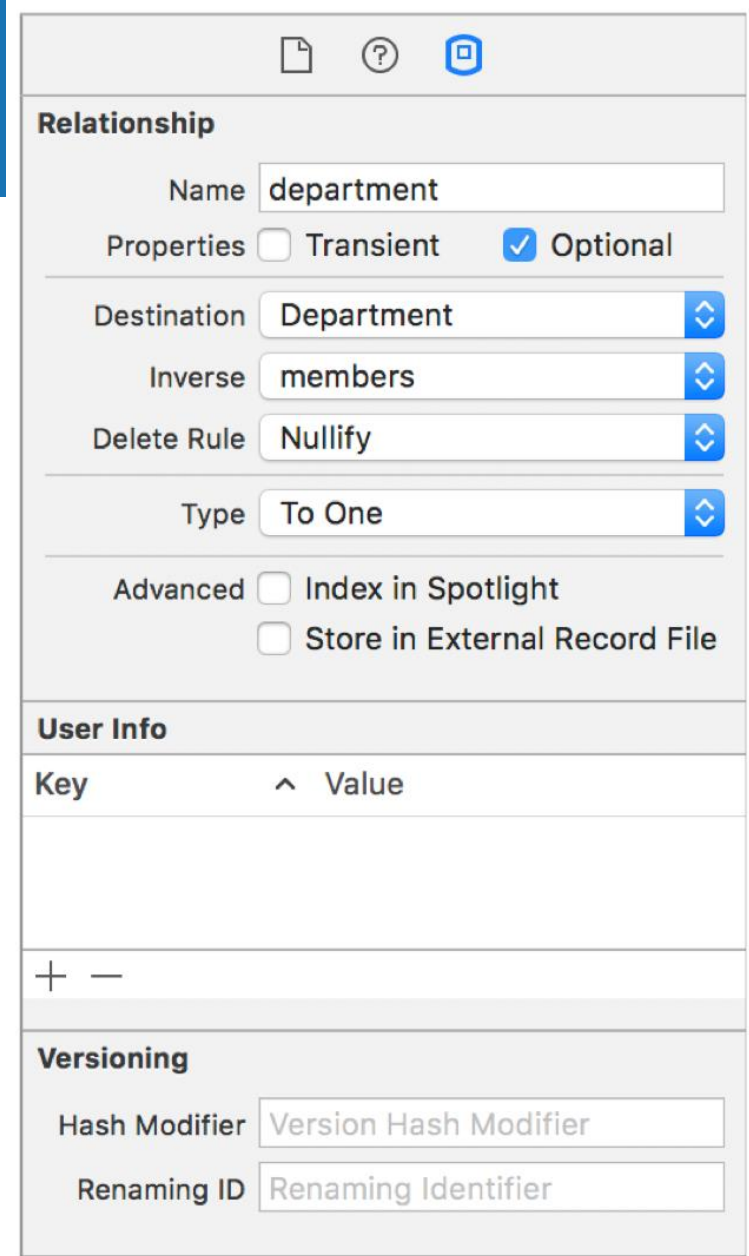
The screenshot shows the 'Attribute' configuration window in Xcode. The 'Name' field is set to 'dateOfBirth'. Under 'Properties', the 'Optional' checkbox is checked, while 'Transient' and 'Indexed' are unchecked. The 'Attribute Type' is set to 'Date'. Below this, there are three date-related fields, each with a 'Minimum Date', 'Maximum Date', and 'Default Date' checkbox, all of which are currently unchecked. The date values shown in the fields are '2/12/1982, 8:00:00 AM'. At the bottom, under 'Advanced', the 'Index in Spotlight' and 'Store in External Record File' checkboxes are also unchecked. Below the attribute configuration, there is a 'User Info' section with a table with two columns: 'Key' and 'Value'. The table is currently empty. At the bottom, there is a 'Versioning' section with two fields: 'Hash Modifier' (containing 'Version Hash Modifier') and 'Renaming ID' (containing 'Renaming Identifier').

Relationship

- Relationships can be defined between entities
 - Ctrl + drag
 - Basically it is a reference to an other NSObject
 - The type of relationship can be 1-N, 1-1 as well
 - The relationship is represented in the code as references in a NSSet
- In case deletion the behavior also can be defined
 - E.g.: Nullify means that the reference is set to NULL

Relationship

- Core Data supports to-one and to-many relationships and fetched properties
 - Fetched properties represent weak, one-way relationships
- A relationship can be
 - Many-to-one type relationship
 - Many-to-many type relationship



The screenshot shows the 'Relationship' configuration window in Xcode. It includes fields for Name, Properties (Transient and Optional), Destination, Inverse, Delete Rule, Type, and Advanced options (Index in Spotlight, Store in External Record File). Below the Relationship section is a 'User Info' table with columns 'Key' and 'Value'. At the bottom is a 'Versioning' section with fields for Hash Modifier and Renaming ID.

Key	Value

Versioning

Hash Modifier	Version Hash Modifier
Renaming ID	Renaming Identifier

Example

```
import UIKit
import CoreData
class DataController: NSObject {
    var managedObjectContext: NSManagedObjectContext
    init(completionClosure: @escaping () -> ()) {
        persistentContainer = NSPersistentContainer(name: "DataModel")
        persistentContainer.loadPersistentStores() { (description, error) in
            if let error = error {
                fatalError("Failed to load Core Data stack: \(error)")
            }
            completionClosure()
        }
    }
}
```

Example

```
let psc = NSPersistentStoreCoordinator(managedObjectModel: mom)

managedObjectContext = NSManagedObjectContext(concurrencyType:
    NSManagedObjectContextConcurrencyType.mainQueueConcurrencyType)
managedObjectContext.persistentStoreCoordinator = psc

let queue = DispatchQueue.global(qos: DispatchQoS.QoSClass.background)
queue.async {
    guard let docURL =
        FileManager.default.urls(for: .documentDirectory,
                                   in: .userDomainMask).last
    else {
        fatalError("Unable to resolve document directory")
    }
    let storeURL = docURL.appendingPathComponent("DataModel.sqlite")
    do {
        try psc.addPersistentStore(ofType: NSSQLiteStoreType,
                                   configurationName: nil, at: storeURL, options: nil)

        DispatchQueue.main.sync(execute: completionClosure)
    } catch {
        fatalError("Error migrating store: \(error)")
    }
}
```

Example

- Creating an object

```
let employee = NSEntityDescription  
    .insertNewObjectForEntityForName("Employee",  
    inManagedObjectContext: managedObjectContext)  
as! AAASEmployeeMO
```


Saving

- Storing an managed object

- Manually

```
do {  
    try managedObjectContext.save()  
} catch {  
    fatalError("Failure to save context:  
\\(error)")  
}
```

Fetching objects

```
let moc = managedObjectContext
let employeesFetch = NSFetchRequest(entityName: "Employee")
do {
    let fetchedEmployees = try
        moc.executeFetchRequest(employeesFetch) as!
                                [AAAEmployeeMO]
} catch {
    fatalError("Failed to fetch employees: \(error)")
}
```

Filtering

```
let firstName = "Trevor",  
fetchRequest.predicate =  
    NSPredicate(format: "firstName == %@",  
    firstName)
```

Properties of objects

```
class MyManagedObject: NSManagedObject {  
    @NSManaged var title: String?  
    @NSManaged var date: NSDate?  
}
```

SQLite in iOS

- We can use SQLite as well
 - `sqlite3_open` – Opening, and creating the database
 - `sqlite3_prepare_v2` – Converting the SQL command from string
 - `sqlite3_step` – Executing the SQL command
 - `sqlite3_column_count` – Number of columns in the response
 - `sqlite3_column_text` – Data of the result table in text
 - `sqlite3_column_name` – Name of the attributes (column)
 - `sqlite3_changes` – Number of affected rows
 - `sqlite3_last_insert_rowids` – ID's of the inserted rows
 - `sqlite3_errmsg` – Text message about the last error
 - `sqlite3_finalize` – Finalization of the created commands
 - `sqlite3_close` – Closing the database, freeing up resources

Example – Open connection

```
func openDatabase() -> OpaquePointer? {  
    var db: OpaquePointer? = nil  
    if sqlite3_open(part1DbPath, &db) == SQLITE_OK {  
        print("\(part1DbPath) has been opened")  
        return db  
    } else {  
        print("Unable to open database.")  
        PlaygroundPage.current.finishExecution()  
    }  
}
```

Example – Create table

```
let createTableString = ""
CREATE TABLE Contact(Id INT PRIMARY KEY NOT NULL, Name CHAR(255));
""

func createTable() {
    var createTableStatement: OpaquePointer? = nil
    if sqlite3_prepare_v2(db, createTableString, -1,
                          &createTableStatement, nil) == SQLITE_OK {
        if sqlite3_step(createTableStatement) == SQLITE_DONE {
            print("Contact table created.")
        } else {
            print("Contact table could not be created.")
        }
    } else {
        print("CREATE TABLE statement could not be prepared.")
    }
    sqlite3_finalize(createTableStatement)
}
```

Example – Insert

```
let insertStatementString = "INSERT INTO Contact (Id, Name) VALUES (?, ?);",
func insert() {
    var insertStatement: OpaquePointer? = nil
    if sqlite3_prepare_v2(db, insertStatementString, -1, &insertStatement, nil) ==
    SQLITE_OK {
        let id: Int32 = 1
        let name: NSString = "Ray"
        sqlite3_bind_int(insertStatement, 1, id)
        sqlite3_bind_text(insertStatement, 2, name.utf8String, -1, nil)
        if sqlite3_step(insertStatement) == SQLITE_DONE {
            print("Successfully inserted row.")
        } else {
            print("Could not insert row.")
        }
    } else {
        print("INSERT statement could not be prepared.")
    }
    sqlite3_finalize(insertStatement)
}
```




Android method

Data storage

- Shared Preferences
- Internal Storage
- External Storage
- SQLite database
- Cloud storage

Android partitions

- /boot, /system, /recovery
 - System partitions, here is the kernel, the system image, and recovery system
- /data
 - User storage
 - Cannot be accessed directly, however the internal storage is also here
- /cache
- /misc
 - Drivers, system settings, ...
- /sdcard
- (/sd-ext, /radio, /wimax, ...)

Shared Preferences

- Storing simple data types, key-value pairs
- boolean, float, int, long, and string
- Stored data is persistent, can be accessed after application restart
- You may use the `PreferenceActivity` for storing application preferences
- To edit an Editor has to be used

```
val sharedPref = activity?.getSharedPreferences(  
    getString(R.string.preference_file_key), Context.MODE_PRIVATE)  
  
with (sharedPref.edit()) {  
    putInt(getString(R.string.saved_high_score_key), newHighScore)  
    commit()  
}  
  
val defaultValue =  
    resources.getInteger(R.integer.saved_high_score_default_key)  
val highScore = sharedPref.getInt(getString(R.string.saved_high_score_key),  
    defaultValue)
```

Files in the APK file

- Project path
 - res/raw/directory.
- To open
 - `openRawResource()`
 - `R.raw.<filename>`
 - Returns an `InputStream`, which can be read

Internal Storage

- Private data
 - After uninstall, the data is deleted
 - ```
String FILENAME = "hello_file";
String string = "hello world!";
FileOutputStream fos =
 openFileOutput(FILENAME,
 Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

# Internal Storage

- `getFilesDir()`
  - The absolute path to the storage of the application
- `getDir()`
  - Create and/or open a directory
- `deleteFile()`
- `fileList()`
  - Returns an array with the file list

# External Storage

- Public storage

- Can be the SD card as well the internal memory (public storage)
- There is no security access
- Example

- `Environment.getExternalStorageDirectory();`

```
String state = Environment.getExternalStorageState();
```

```
if (Environment.MEDIA_MOUNTED.equals(state)) {
```

```
 mExternalStorageAvailable = mExternalStorageWriteable = true;
```

```
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
```

```
 mExternalStorageAvailable = true;
```

```
 mExternalStorageWriteable = false;
```

```
} else {
```

```
 mExternalStorageAvailable = mExternalStorageWriteable = false;
```

```
}
```



# External Storage

- Further features
  - `getExternalFilesDir()` returns the path to the directory
  - `getExternalStoragePublicDirectory()`
  - There are several pre-defined categories
    - Music, Podcasts, Ringtones, Alarms, Notifications, Pictures, Movies, Download
  - `getCacheDir()`
    - Internal storage
  - `getExternalCacheDir()`
    - External storage
  - `getExternalStorageDirectory()`
    - `/Android/data/<package_name>/cache/`

# Temporary files

- A possible way, combining the well-known Java functions

```
File outputDir = context.getCacheDir();
```

```
File outputFile = File.createTempFile("prefix", "extension", outputDir);
```

# Using SQL Database

- SQLite
  - Accessed by name
  - Private to the application
  - You may use the `SQLiteOpenHelper` class and override its `onCreate()` method

# Using SQL Database

- Example

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {
 private static final int DATABASE_VERSION = 2;
 private static final String DICTIONARY_TABLE_NAME =
 "dictionary";
 private static final String DICTIONARY_TABLE_CREATE =
 "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
 KEY_WORD + " TEXT, " +
 KEY_DEFINITION + " TEXT);";

 DictionaryOpenHelper(Context context) {
 super(context, DATABASE_NAME, null, DATABASE_VERSION);
 }

 @Override
 public void onCreate(SQLiteDatabase db) {
 db.execSQL(DICTIONARY_TABLE_CREATE);
 }
}
```

# Using SQL Database

- Once it is done, the database can be accessed by using the `getReadableDatabase()` and `getWritableDatabase()` functions
  - Both returns an `SQLiteDatabase` object
  - Then you can use the `SQLiteDatabase.query(...)` function, which returns a `Cursor`
  - For more complex query you may want to use the `SQLiteQueryBuilder` class
- It is a good practice to create a static database handling class, and then the database access can be hidden (Proxy)
  - Two ways can be used
    - `public static Cursor getAllData()`
      - Optimal as the `Cursor` requires less memory
    - `public static ArrayList<MyData> getAll data`
      - More elegant

# Backup service

- You can backup the persistent data of your application to the Google Cloud services
  - After the factory reset of the phone, or on new device, or after reinstall of the application, these data can be restored
- It is important, that these data are not synchronized automatically, but the API provides functions to initiate the backup and restore
- The process is the following
  - BackupManager retrieves the data from the application and stores/uploads them
  - The BackupManager downloads the data and informs the application to start the restoration process

# Backup – Purpose

- You should not use this service to synchronize between several devices
  - Although Google do not check abuse
- Application settings, user data can be stored
- The backup and the restore is executed automatically, not by request
- It is not guaranteed that it is available on all device/platform
  - You cannot rely on the backup, to provide new service
- Other application cannot access the data
- Communication is not necessarily secured

# Backup in practice

1. Declare in the manifest file
2. Register the application in the backup service
3. Define a backup client, either
  - Subclassing from the BackupAgent class, or
  - Subclassing from the BackupAgentHelper class

```
<manifest ... >
 <application android:label="MyApplication"
 android:backupAgent="MyBackupAgent">
 <activity ... >
 </activity>
 </application>
 </manifest>
```



# Backup – registration

<http://code.google.com/android/backup/signup.html>

```
<application android:label="MyApplication"
 android:backupAgent="MyBackupAgent">

 ...

 <meta-data android:name="com.google.android.backup.api_key"
 android:value=" ..." />
</application>
```

# BackupAgent

- The `onBackup()` and `onRestore()` function have to be overridden
  - You are responsible for the data format, version, coding, decoding
  - You are responsible for selecting the data
  - SQLite data also can be saved
- `onBackup()`
  - 3 parameter: `oldState`, `data`, `newState`

# BackupHelper

- Pre-defined backup functions
  - You do not have to override the `onBackup()` and `onRestore()`
- `SharedPreferencesBackupHelper`
  - ☺
- `FileBackupHelper`

# Usage

```
public class MyPrefsBackupAgent extends BackupAgentHelper {
 // The name of the SharedPreferences file
 static final String PREFS = "user_preferences";

 // A key to uniquely identify the set of backup data
 static final String PREFS_BACKUP_KEY = "prefs";

 // Allocate a helper and add it to the backup agent
 @Override
 public void onCreate() {
 SharedPreferencesBackupHelper helper = new
 SharedPreferencesBackupHelper(this, PREFS);
 addHelper(PREFS_BACKUP_KEY, helper);
 }
}
```

- `dataChanged()` call indicate the change.
  - After the signal the `onBackup()` will be called
  - There is no specific time interval or deadline



# Final Exam!

Next week