Concurrency with Core Data

In many real-world applications, there is a need to load data in the background and use Core Data for storing it. When a serious amount of data is involved, you don't want to process it on the main thread to avoid blocking of the user interface.

The developer documentation from Apple is very clear regarding the recommended pattern for concurrent programming with Core Data. It states, "The pattern recommended for concurrent programming with Core Data is thread confinement: each thread must have its own entirely private managed object context."

There are two possible ways to adopt the pattern:

- Create a separate managed object context for each thread and share a single persistent store coordinator. This is the typically recommended approach.
- Create a separate managed object context and persistent store coordinator for each thread. This option provides for greater concurrency at the expense of greater complexity.

You can find more information in the Apple documentation at `http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/CoreData/Articles/cdConcurrency.html`.

In the source code of this chapter, you can download a project named `ConcurrentThreads` to see a working example.

To adapt to the first approach you have to make changes to the application delegate (if that is the place where you initialize and configure Core Data).

The first step is to add the following two highlighted code lines to the `appContext` method. The first line is calling the `setMergePolicy:` method on the `managedObjectContext`, and the second line configures an Observer with a selector to the method `mocDidSaveNotification:`

```
- (NSManagedObjectContext *)appContext
{
    if (self.managedObjectContext != nil) {
        return self.managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self appStoreCoordinator];
    if (coordinator != nil) {
        _managedObjectContext = [[NSManagedObjectContext alloc] init];
        [self.managedObjectContext setUndoManager:nil];
        [self.managedObjectContext
              setMergePolicy:NSMergeByPropertyObjectTrumpMergePolicy];
            // subscribe to change notifications
        [[NSNotificationCenter defaultCenter] addObserver:self
              selector:@selector(mocDidSaveNotification:)
              name:NSManagedObjectContextDidSaveNotification
              object:nil];
        [self.managedObjectContext
setPersistentStoreCoordinator:coordinator];

    }
    return self.managedObjectContext;
}
```

The implementation of the `mocDidSaveNotification:notification` method is performing the following logic:

- If the change notification is from the main managed object context, ignore it because it will be saved by the `saveContext` method.
- If the `persistentStoreCoordinator` of the managed object context is different that the `persistentStoreCoordinator` of the managed object context belonging to the save notification, again no action is performed.
- In all other cases, the managed object context `mergeChangesFromContextDidSaveNotification:notification` method is called in the main dispatch queue, saving and merging the changes:

```
- (void)mocDidSaveNotification:(NSNotification *)notification
{
    NSManagedObjectContext *savedContext = [notification object];

        // ignore change notifications for the main MOC
    if (self.managedObjectContext == savedContext)
        {
        return;
        }

    if (self.managedObjectContext.persistentStoreCoordinator !=
        savedContext.persistentStoreCoordinator)
        {
            // that's another database
        return;
        }

    dispatch_sync(dispatch_get_main_queue(), ^{

        [self.managedObjectContext
         mergeChangesFromContextDidSaveNotification:notification];
    });
}
```