3つ目のiOSアプリケーション：iCloud

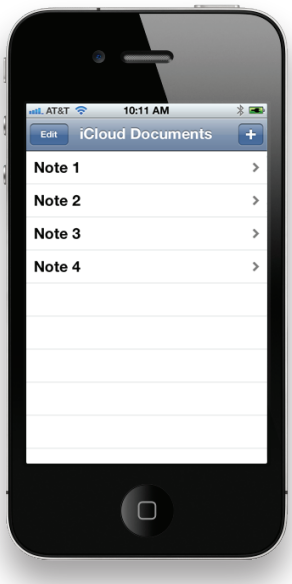
Developer

目次

3つ目のiOSアプリケーションについて

3つ目のiOSアプリケーションでは、iCloudドキュメントストレージAPIを紹介します。これらのAPIを使用して、ユーザのiCloudストレージに対してファイルの保存と操作を行います。

このチュートリアルでは、ユーザのiCloudストレージ内でテキストファイルを作成し、管理するアプリケーションをビルドします。このアプリケーションは、iCloudの基本概念を習得し、簡単なiCloudアプリケーションの作成体験を得るための入門を意図しています。



このチュートリアルを最大限に生かすためには、一般的なコンピュータプログラミングの基礎、特にObjective-C言語に多少慣れていなければなりません。このチュートリアルでは、iCloud統合に焦点を当てるため、アプリケーションの基本的な面は説明しません。したがって、アプリケーションの作成過程に関して基本的な理解を得るために、アプリケーション関係のチュートリアルを済ませておかなければなりません。

初めに

Your Third iOS App は、iCloudへのドキュメントの保存をサポートする機能を組み込む方法を示すため、*Your First iOS App* と *Your Second iOS App: Storyboards* で得た知識に基づいて構築されています。

iCloudを使用するためのプロジェクトの設定

iCloudを使用するアプリケーションは、ユーザのiCloudアカウント内のドキュメントストレージスペースにアクセスできる特別なエンタイトルメントを備えていなければなりません。これらのエンタイトルメントを用意するため、プロジェクトの設定を修正し、XcodeとiOS Provisioning Portalでいくつかの手順を実行します。この設定の一部は開発サイクルの後の方でも実行可能ですが、このチュートリアルでは、プロジェクトが後で適切に機能するように、これらの問題を先に片付けます。

関連する章： “入門” （8 ページ）

ドキュメントオブジェクトを使ってファイルの内容を管理

ドキュメントオブジェクトは、ユーザのiCloudアカウント内に保存するファイルをもっとも簡単に管理する手段です。各ドキュメントオブジェクトは、アプリケーションのデータの個々の部分を管理し、適切な時期にデータをディスクに書き込んだり、メモリに読み出したりする操作を担当します。そのデータとのやり取りはすべて、定義したドキュメントクラスのインターフェイスを通じて行われます。

関連する章： “ドキュメントサブクラスの定義” （19 ページ）

アプリケーションのユーザインターフェイスの構築

アプリケーションのデータを主として管理するのはiCloudですが、アプリケーションにはそのデータを表示するためのユーザインターフェイスが必要です。このチュートリアルでは、2つのView Controllerを使用してコンテンツを表示します。最初のView Controllerは、利用可能なドキュメントのリストを表示し、それらのドキュメントの作成と削除を管理します。2つ目のView Controllerは、ドキュメントの中身を表示し、中身を編集できるようにします。

関連する章: “Master View Controllerの実装” （27 ページ）、“Detail View Controllerの実装” （46 ページ）

iCloud内の既存ドキュメントの検索

アプリケーションが動作していないときにドキュメントのリストが変化する場合があるので、iCloud内のドキュメントの検索機能は必須です。ドキュメントはさまざまなデバイスで作成できる上、不要になったドキュメントをユーザが削除することもあります。このアプリケーションでは、メタデータクエリオブジェクトを使用して、ドキュメントの検索やドキュメントリストの変化の検出を行います。

関連する章： “iCloudドキュメントの検索”（58 ページ）

問題の解決と次のステップの選択

このチュートリアルのタスクを完了する過程で、解決法がわからない問題に遭遇するかもしれません。そのため、よくあるエラーを掲載しているほか、自分のプロジェクトのコードと比較できるように、コードリストも収録してあります。

このチュートリアルを終えたら、アプリケーションの改良や知識の習得に役立つ方法を考察してみてください。

関連する章： “トラブルシューティング”（63 ページ）、“次のステップ”（66 ページ）、“コードリスト”（70 ページ）

必要事項

次のチュートリアルでは、アプリケーション作成時の最善の慣習を紹介してあるので、このチュートリアルを始める前に読むことをお勧めします。

- *Your First iOS App*
- *Your Second iOS App: Storyboards*

関連項目

このチュートリアルでは、アプリケーションのデータモデルとコントローラ層でのiCloudの採用という、アプリケーション開発の1つの面しか取り上げていません。アプリケーション開発のその他の重要な面については、次のドキュメントを参照してください。

- iOSアプリケーションのユーザインターフェイスとユーザ体験の設計に関して推奨する方法については、『*iOS Human Interface Guidelines*』を参照してください。
- フル機能のiOSアプリケーションの作成に関する包括的なガイドについては、『*iOS App Programming Guide*』を参照してください。
- ドキュメントベースのアプリケーションの作成の詳細については、『*Document-Based App Programming Guide for iOS*』を参照してください。
- 自分のアプリケーションをApp Storeに送る準備をする際に実行する必要がある作業については、『*Developing for the App Store*』を参照してください。

入門

このチュートリアルでiOSアプリケーションを作成するには、Xcode 4.2以降、およびiOS SDKが必要です。Xcodeは、iOSとMac OS Xの両方の開発を対象にしたAppleの統合開発環境です。iOS SDKは、iOSプラットフォームのプログラミングインターフェイスを備えています。

XcodeをMacにインストールしていない場合は、iOS Dev Centerを訪れて、Xcode 4.2デベロッパツールセット（iOS 5 SDKが含まれています）をダウンロードしてください。パッケージをダブルクリックして、インストーラの指示に従います。

Important: このチュートリアルを完了するためには、iOS 5が動作するデバイスが必要になるほか、アクティブなiCloudアカウントで設定しなければなりません。iCloud向けのドキュメントは、iOSシミュレータからではなく、デバイスから発生しなければなりません。

プロジェクトの新規作成

アプリケーション開発を開始するため、新しいXcodeプロジェクトを作成します。

To : 新しいプロジェクトを作成するには、以下の手順を実行します。

1. Xcodeを開きます。
2. 「ファイル(File)」＞「新規(New)」＞「新規プロジェクト(New Project)」を選択し、プロジェクトの作成を開始します。
3. iOSセクションで「アプリケーション(Application)」を選択し、テンプレートのリストから「Master-Detail Application」を選んで、「次へ(Next)」をクリックします。

   新しいダイアログが表示され、アプリケーションの名前を入力し、プロジェクトの追加オプションを選択するように指示されます。
4. 「製品名 (Product Name)」、「会社ID(Company Identifier)」、「クラスプレフィックス(Class Prefix)」の各フィールドに入力します。

   下記の値を使用できます。

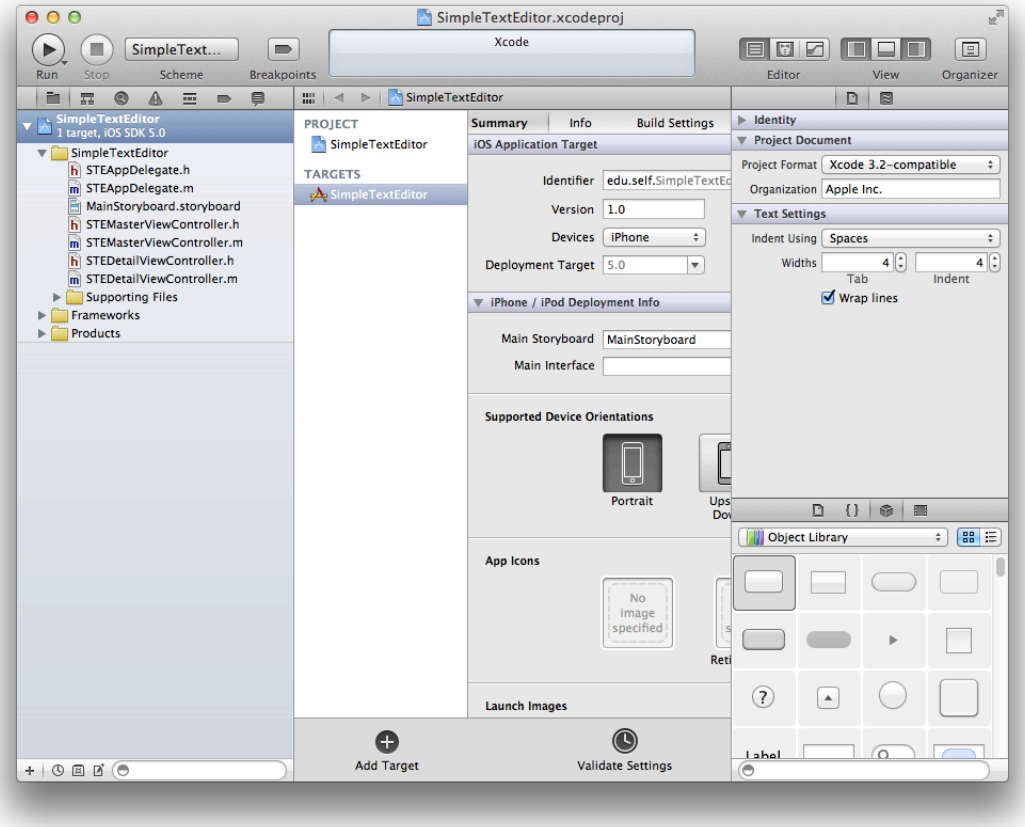
   - Product Name： `SimpleTextEditor`
   - Company Identifier：自社の識別子が存在する場合はその識別子、存在しない場合は`edu.self`

- Class Prefix： STE

5. 「デバイスファミリ(Device Family)」ポップアップメニューで、iPhoneが選択状態になっていることを確認します。
6. 「ストーリーボードを使用(Use Storyboard)」と「自動参照カウントを使用(Use Automatic Reference Counting)」オプションが選択状態になっていること、および「単体テストを含める(Include Unit Tests)」オプションが選択状態になっていないことを確認します（これらはデフォルト値です）。
7. 「次へ(Next)」をクリックします。

   プロジェクトを保存したい場所を指定するように指示する別のダイアログが表示されます。
8. プロジェクトの場所を指定して（「ソース管理（source control）」オプションは未選択状態のまま）、「作成 (Create)」をクリックします。

   次と同様の新しいプロジェクトウインドウが表示されます。



iCloud機能をまだ追加していないので、Xcodeが備えているシミュレータアプリケーションでアプリケーションをビルドし、実行できます。名前が示すとおり、シミュレータでは、iOSベースのデバイスで実行した場合のアプリケーションの外観や動作を確認できます。ただし、プロジェクトの残りの部分に関しては、iOS 5で設定され、アクティブなiCloudアカウントを備えたiOSデバイスで、アプリケーションのビルドと実行を行わなければなりません。

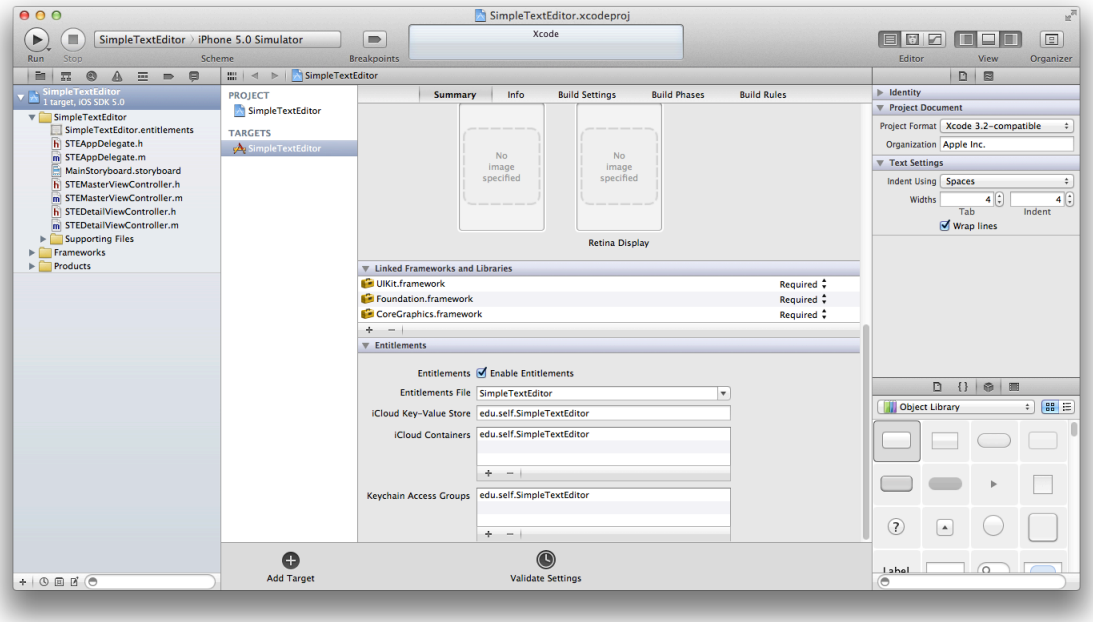
プロジェクトのiCloudエンタイトルメントの設定

iCloudを使用するアプリケーションは、ユーザのアカウントにアクセスできるように、特別なエンタイトルメントで設定されていなければなりません。これらのエンタイトルメントは、Xcodeから直接設定します。

To : iCloudエンタイトルメントを設定するには、以下の手順を実行します。

1. プロジェクトのターゲットを選択します。
2. 「概要(Summary)」タブで、「エンタイトルメント(Entitlements)」セクションまで下にスクロールします。
3. 「エンタイトルメントを有効 (Enable Entitlements)」オプションを選択します。

   Xcodeがエンタイトルメントファイルをプロジェクトに追加し、iCloudにアクセスするためのデフォルト値を自動的に入力します。



このチュートリアルでは、Xcodeが提供するデフォルトのエンタイトルメント値で十分です。独自プロジェクトの場合は、別のアプリケーションのバンドル識別子に合わせて、必要に応じてこれらの値を変更してもかまいません。たとえば、無償版のアプリケーションを作成する場合は、有料版のバンドル識別子を割り当てて、この2つのバージョンのアプリケーションがデータを共有できるようにします。つまり、無償版から有償版にアップグレードしたときに、ユーザが以前のデータにアクセスできなくなることを防ぐわけです。

> **注意:** このチュートリアルでは、「iCloudコンテナ(iCloud Containers)」フィールドに関連付けられたエンタイトルメントしか使用しませんが、「iCloud Key-Value Store」フィールドに値が入っていても害はありません。ただし、出荷するアプリケーションでは、そのアプリケーションが使用しないエンタイトルメントの値を入れておいてはなりません。

iOSプロビジョニングポータルでのアプリケーションIDの設定

iCloudアプリケーションをデバイスにインストールするためには、アプリケーションIDとプロビジョニングプロファイルを作成しなければなりません。アプリケーションIDとプロビジョニングプロファイルの作成は、iOSプロビジョニングポータルの管理者にしか行えません。管理者権限を持っていない場合は、次の手順を実行するようにチームの別のメンバに依頼しなければなりません。

1. アプリケーションIDを作成します。
2. iCloud用アプリケーションIDを構成します。
3. プロビジョニングプロファイルを生成します。

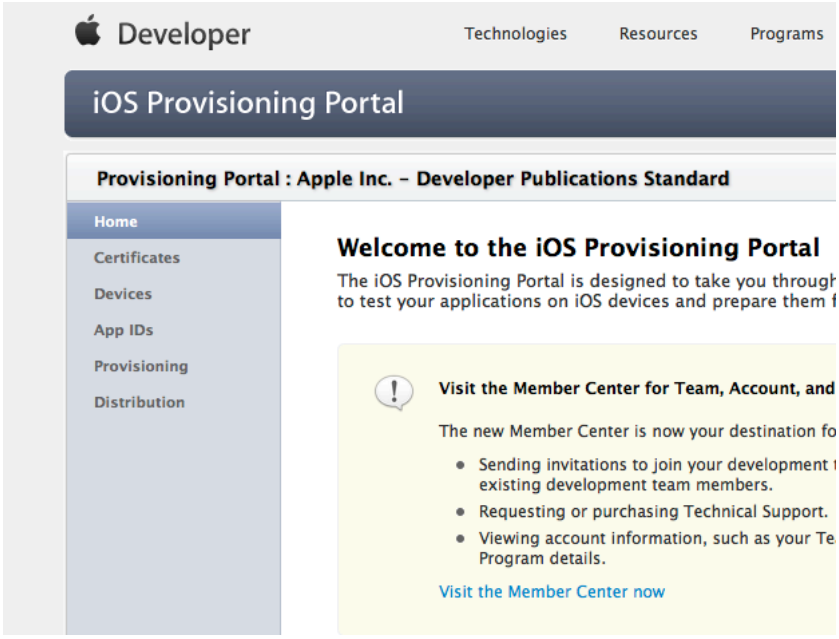
アプリケーションIDの作成

まず、アプリケーション用のアプリケーションIDを作成する必要があります。アプリケーションIDは、デバイスとiCloud内でこのアプリケーションを一意に識別します。

To : アプリケーションIDを作成するには、以下の手順を実行します。

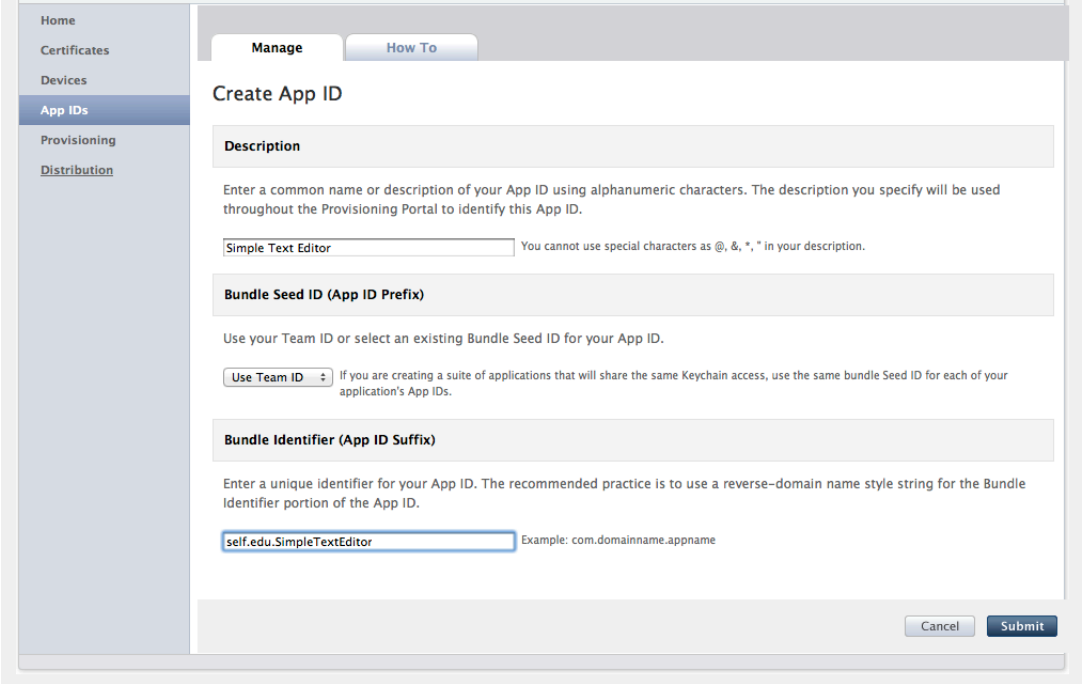
1. Webブラウザを開いて、http://developer.apple.com/ios/manage/overview/index.actionを参照します（必要に応じて、自分のデベロッパ資格情報を使用してログインします）。

2. 左側の欄からアプリケーションIDを選択します。



3. 「App IDs」ページで、「New App ID」をクリックします。
4. 「Create App ID」ページに次の情報を入力します。
   - 説明: `Simple Text Editor`
   - バンドルシードID:Use Team ID（デフォルト）

- バンドル識別子:自分のアプリケーションのバンドル識別子を使用します。これは、会社IDとSimpleTextEditorアプリケーション名を組み合わせたものです。Xcodeでこの値を調べるには、アプリケーションターゲットの「Summary」ペインに移動して、「Identifier」フィールドを見ます。



5. 「Submit」をクリックします。

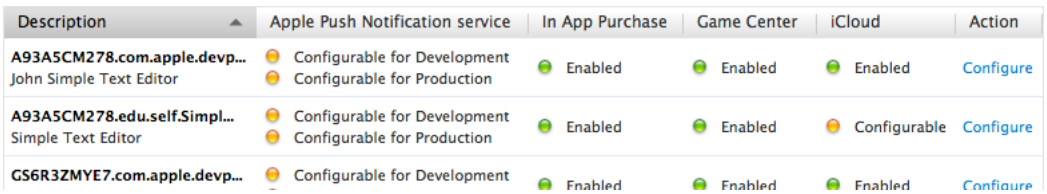
アプリケーションIDを作成しても、まだiCloud用には設定されていないので注意してください。

iCloud用アプリケーションIDの設定

iCloudをサポートするようにアプリケーションIDを明示的に設定しなければなりません。

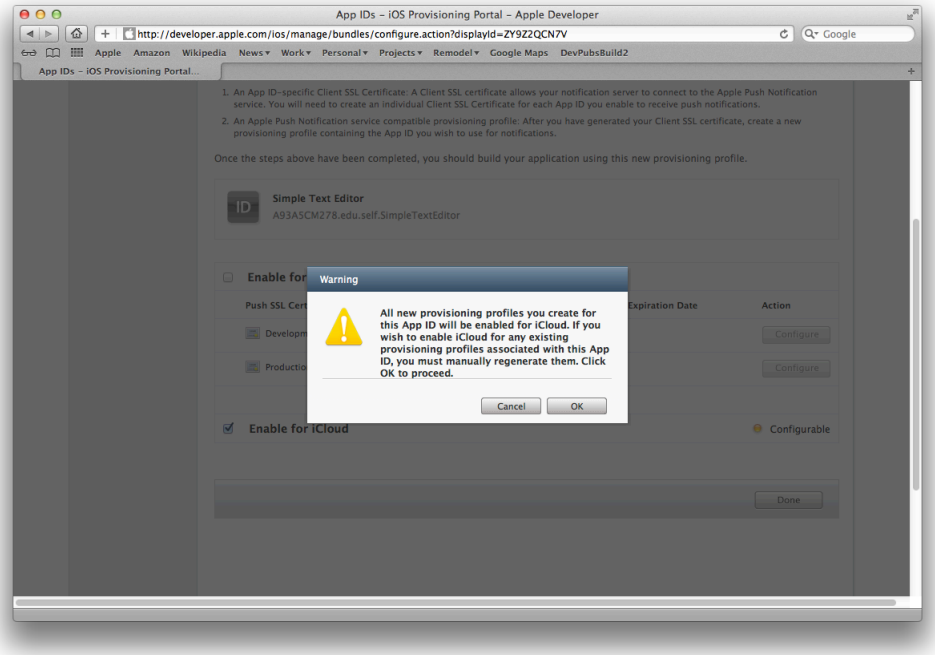
To : アプリケーションIDをiCloud用に設定するには、以下の手順を実行します。

1. 自分のアプリケーションID用の「Configure」ボタンをクリックします。



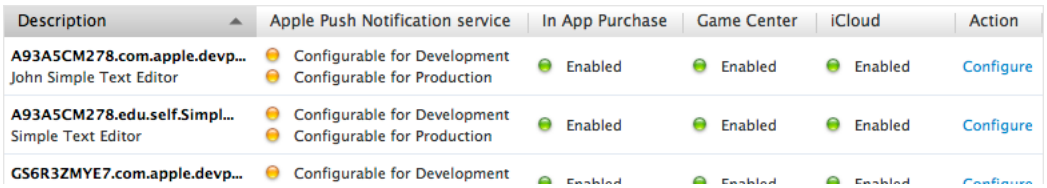
2. 「Enable for iCloud」を選択します。

   このチェックボックスを有効にすると、このアプリケーションに関連付けられたプロビジョニングプロファイルの再生成について、プロビジョニングポータルが警告を発します。



3. 「OK」を押してダイアログを消します。
4. 「Done」をクリックします。

   これで、iCloudをサポートするようにアプリケーションIDが設定されました。



プロビジョニングプロファイルの生成

最後の手順は、開発中に使用できるプロビジョニングプロファイルを生成することです。

To : プロビジョニングプロファイルを生成するには、以下の手順を実行します。

1. iOSプロビジョニングポータルで、左側の欄から「Provisioning」を選択します。

2. 「New Profile」をクリックします。

   プロビジョニングポータルが、自分のプロビジョニングプロファイルに関する情報を指定するための新しい画面を表示します。

3. 自分のプロビジョニングプロファイルに関する情報を指定します。
   - プロファイル名: `My Simple Text Editor Profile`
   - 証明書:ユーザ名を選択します。
   - アプリケーションID:`Simple Text Editor`（先ほど設定したアプリケーションIDです）
   - デバイス:アプリケーションのテストに使用する予定のデバイスを選択します。

4. 「Submit」をクリックします。

   ポータルが、開発プロビジョニングプロファイルのリストが表示された画面を返します。新しいプロビジョニングプロファイルの状態は、当初、「Pending」に設定されています。

5. 左側の欄の「Provisioning link」をクリックして、ページを更新します。

   これで、プロビジョニングプロファイルの状態が「Active」に設定されたはずです。

6. Xcodeで「ウインドウ(Window)」＞「オーガナイザ(Organizer)」を選び、「デバイス(Devices)」タブを選択します。
7. 「ライブラリ(Library)」セクションで、「プロビジョニングプロファイル(Provisioning Profiles)」を選択します。
8. ページの下部にある「更新(Refresh)」ボタンをクリックします。

   リストを更新すると、新しいプロビジョニングプロファイルがリストに表示されるはずです。

マシンが適切な証明書で設定されている場合、プロビジョニングプロファイルがXcodeオーガナイザに有効として表示されるはずです。インストールされた証明書に不一致がある場合は、Xcodeが「オーガナイザ(Organizer)」ウインドウに適切なメッセージを表示します。この情報を使用すると、問題を診断しやすくなります。プロビジョニングプロファイルの問題の診断については、“プロビジョニングプロファイルの問題の診断” （63 ページ）を参照してください。

カスタムプロビジョニングプロファイルを使用するようにプロジェクトを更新

カスタムプロビジョニングプロファイルを生成した後は、そのプロファイルをプロジェクトに関連付けなければなりません。

To : プロビジョニングプロファイルをプロジェクトに関連付けるには、以下の手順を実行します。

1. プロジェクトのターゲットを選択します。
2. 「ビルド設定(Build Settings)」タブで、「コード署名ID(Code Signing Identity)」を探します。
3. 「コード署名ID(Code Signing Identity)」ポップアップから、自分のプロビジョニングプロファイルを選択します。

自分のアプリケーション専用に作成したプロビジョニングプロファイルを指定しなければなりません。汎用のTeam Provisioning Profileを使用してiCloudアプリケーションをテストすることはできません。

アプリケーションのiCloudコンテナの初期化

iCloudを初めて使用する前に、`URLForUbiquityContainerIdentifier:`メソッドを呼び出して、アプリケーションがiCloudを使用できるように準備する機会をシステムに与える必要があります。初めて呼び出されたとき、このメソッドは、アプリケーションのサンドボックスを修正して、コンテナディレクトリにアクセスできるようにします。また、iCloudが実際に使用可能がどうか、および現行バージョンに基づいて設定されているかどうかを確認するためのチェックも行います。これらのチェックにはある程度の時間を要するため、アプリケーションの起動サイクルの中で早めに呼び出して、アプリケーションが何らかのファイルにアクセスする必要が生じる前にチェックが完了するようにしなければなりません。アプリケーションが複数のコンテナディレクトリにアクセスする場合は、コンテナディレクトリごとにメソッドを呼び出さなければなりません。

To : アプリケーションのiCloudコンテナを初期化するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEAppDelegate.m`を選択します。
2. iCloudが使用可能かどうかをチェックする`initializeiCloudAccess`という新規メソッドを定義します。

   iCloudが使用可能かどうかに基づいてSimple Text Editorが設定を変更することはないので、このメソッドは単に結果を記録するだけです。自分独自のアプリケーションでは、この結果を使用して追加の設定手順を実行してもかまいません。

```
- (void)initializeiCloudAccess {


dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
      if ([[NSFileManager defaultManager]
```

```
            URLForUbiquityContainerIdentifier:nil] != nil)
        NSLog(@"iCloud is available\n");
    else
        NSLog(@"This tutorial requires iCloud, but it is not
available.\n");
});
}
```

3. `application:didFinishLaunchingWithOptions:`メソッド内で、`initializeiCloudAccess`メソッドを呼び出します。

```
- (BOOL)application:(UIApplication *)application
      didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    [self initializeiCloudAccess];

    // アプリケーション起動後のカスタマイズを記述する場所。
    return YES;
}
```

独自アプリケーションでこれを使用して早めに`URLForUbiquityContainerIdentifier:`メソッドを呼び出すと、iCloud availability.iCloudが使用可能かどうかに関連するアプリケーション固有のタスクを実行することができます。たとえば、結果を使用して、アプリケーションのデータ構造を異なる設定にすることができます。また、結果を使用して、ユーザインターフェイス用の設定オプションを用意することもできます。このメソッドは、ユーザインターフェイスを構成する前と後のどちらかにその値を返すことができるので、ここからインターフェイスを直接修正しようとしてはなりません。代わりに、フラグを設定して、アプリケーション設定が変化したことを該当するインターフェイスオブジェクトに通知します。その後，これらのオブジェクトが、フラグの値に基づいて、対応するアクションを実行します。

まとめ

この章では、Xcodeを使用して、「Master-Detail」テンプレートに基づく新規プロジェクトを作成しました。次に、プロジェクトに合わせてiCloudエンタイトルメントを設定しました。また、iOSプロビジョニングポータルを使用して、プロジェクト用のアプリケーションIDとプロビジョニングプロファイルを作成し、そのプロビジョニングプロファイルをプロジェクトに関連付けました。最後に、アプリケーションの起動コードでiCloudコンテナディレクトリを初期化しました。

この時点で、プロジェクトの設定は完了し、指定したiOS デバイスにインストールできる状態になっています。次の章では、iCloudのドキュメントサポートを活用する際に必要になるコードの記述を始めます。

ドキュメントサブクラスの定義

iCloud内のファイルを管理する最善の方法は、`UIDocument`クラスに基づくカスタムドキュメントオブジェクトを使用することです。このクラスは、ローカルとiCloud内の両方でファイルを管理する際に必要になる基本機能を提供します。このクラスを使用するためには、このクラスをサブクラス化して、アプリケーションデータの読み書きを担当するメソッドをオーバーライドしなければなりません。

Model-View-Controllerアーキテクチャでは、ドキュメントオブジェクトはModel Controllerです。つまり、アプリケーションのデータモデルの一部に対してコントローラオブジェクトの役割を果たすことがジョブです。ドキュメントオブジェクトを使用すると、通常、アプリケーションのデータ構造と、関連付けられたデータを表すView Controller間のやり取りが容易になります。

UIDocumentサブクラスの作成

Simple Text Editorアプリケーションは、カスタム`UIDocument`サブクラスを使用して、個々のテキストファイルの中身を管理します。このクラスはテンプレートプロジェクトには存在しないので、明示的に作成しなければなりません。

To : ドキュメントサブクラスを作成するには、以下の手順を実行します。

1. Xcodeで、「ファイル(File)」＞「新規(New)」＞「新規ファイル(New File)」を選択します。
2. iOSセクションで、「Cocoa Touch」グループを選択します。
3. ファイルタイプとして「Objective-C」クラスを選択し、「次へ(Next)」をクリックします。
4. サブクラス用に次の情報を指定します。
   - クラス: `STESimpleTextDocument`
   - サブクラス:`UIDocument`（用意されたフィールドにクラス名を入力しなければならない場合があります）
5. 「次へ(Next)」をクリックします。

   ソースファイルを保存するようにXcodeが指示します。
6. プロジェクトディレクトリに移動して、ターゲットのリストでSimpleTextEditorターゲットが選択されていることを確認します。

7. 「作成(Create)」をクリックします。

STESimpleTextDocumentクラスが管理するデータは、ドキュメントに関連付けられたテキストが格納されたNSStringオブジェクトで構成されています。プロパティを使用してこの文字列を宣言します。

To : この文字列データ用のプロパティ宣言を追加するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STESimpleTextDocument.hを選択します。
2. ドキュメントの内容を格納できるように、documentTextという宣言済みプロパティを追加します。

   プロパティ宣言は次のようになります。

```
@property (copy, nonatomic) NSString* documentText;
```

documentTextプロパティの実装を完了するためには、アクセサメソッドを合成するようにコンパイラに指示する必要もあります。ドキュメントの実装ファイルに、そのためのコードを追加します。

To : ドキュメントテキストプロパティ用にアクセサメソッドを合成するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STESimpleTextDocument.mを選択します。
2. @implementation STESimpleTextDocument行の後に、次のコード行を追加します。

```
@synthesize documentText = _documentText;
```

   実際には合成済みのgetterメソッドだけが作成されます。このアプリケーションは、新しい文字列がこのドキュメントに割り当てられたときにいくつかの追加タスクを実行するカスタムsetterメソッドを備えます。

ドキュメントデータを設定するためのメソッドのオーバーライド

STESimpleTextDocumentクラスは、カスタムsetterメソッドをdocumentTextプロパティに使用します。カスタムsetterメソッドは、ドキュメントテキストの設定時に「取り消し」処理追加します。これを実現すると、アプリケーションでドキュメントのテキストに対する変更を取り消せるようになるだけでなく、iCloudに関連するいくつかのメリットも生じます。特に、ドキュメントの自動保存機能が起動して、変更内容がiCloudに送信されます。

To : setDocumentText:メソッドを実装するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STESimpleTextDocument.mを選択します。
2. クラスの実装に次のコードを追加します。

```
- (void)setDocumentText:(NSString *)newText {
    NSString* oldText = _documentText;
    _documentText = [newText copy];

    // 取り消し操作を登録する
    [self.undoManager setActionName:@"Text Change"];
    [self.undoManager registerUndoWithTarget:self
            selector:@selector(setDocumentText:)
            object:oldText];
}
```

setDocumentText:メソッドで、古いテキストが入った文字列を保存し、処理の一部としてそれを取り消しマネージャに渡さなければなりません。この取り消しマネージャは、不要になるまでの間、古い文字列への参照を維持します。

ドキュメントデータを読み書きするメソッドの実装

ドキュメントオブジェクトは、ドキュメントの中身をディスクに書き出し、ディスクから読み込みます。読み書き操作を開始する際に必要になるほぼすべての作業は、UIDocumentクラスによって自動的に処理されます。ただし、データの実際の読み書きはドキュメントクラスに固有なので、カスタムコードをいくらか記述しなければなりません。

Simple Text Editorアプリケーションでは、ドキュメントオブジェクトのデータはNSStringオブジェクトですが、UIDocumentではディスクに文字列を直接書き込むことができません。そのため、ドキュメントオブジェクトが処理できる形、つまりNSDataオブジェクトに、文字列をパッケージ化しなければなりません。これは、ドキュメントサブクラスのcontentsForType:error:メソッドで行います。

To : contentsForType:error:メソッドを実装するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STESimpleTextDocument.mを選択します。
2. contentsForType:error:メソッドを実装します。

   このメソッドは、UIDocumentクラスの宣言済みメソッドです。このメソッドをオーバーライドすると、アプリケーションのデータをドキュメントオブジェクトに提供できるようになります。contentsForType:error:メソッドの実装では、ディスクに書き出すべき有効な文字列があることを確認してから、その文字列をNSDataオブジェクトにパッケージ化してから、返さなければなりません。

```
- (id)contentsForType:(NSString *)typeName error:(NSError **)outError {
    if (!self.documentText)
        self.documentText = @"";

    NSData *docData = [self.documentText
                           dataUsingEncoding:NSUTF8StringEncoding];
    return docData;
}
```

組み込みのドキュメントインフラストラクチャを使用すると、データをディスクに書き出す方法に悩む代わりに、データに集中することができます。データオブジェクトが返されると、ドキュメントオブジェクトは、ファイルコーディネータオブジェクトを作成し、このオブジェクトを使用してデータをディスクに書き出します。ファイルコーディネータを使用すると、アプリケーションがファイルに排他的にアクセスできるため、iCloudコンテナディレクトリにファイルを保存するときにはファイルコーディネータが必要になります。UIDocumentクラスを使用しなかった場合は、ファイルコーディネータオブジェクトを自分で作成しなければならなくなります。

ドキュメントデータを読み込むプロセスも、書き出すプロセスと同様です。loadFromContents:ofType:error:メソッドに渡すべきドキュメントデータをデータオブジェクトから取り出すだけです。書き出しの場合と同様、ファイルコーディネータを作成する必要はなく、提供されたオブジェクトからデータを読み取る以外に何もする必要はありません。

To : loadFromContents:ofType:error:メソッドを実装するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STESimpleTextDocument.m`を選択します。
2. `loadFromContents:ofType:error:`メソッドを実装します。

   `loadFromContents:ofType:error:`メソッドの実装は、データオブジェクトに有効な情報が入っているかどうかを調べて、入っていた場合はその情報を使用して新たな文字列オブジェクトを作成します。何も入っていなかった場合は、ドキュメント内容を空の文字列に初期化します。

```
- (BOOL)loadFromContents:(id)contents
             ofType:(NSString *)typeName
             error:(NSError **)outError {
    if ([contents length] > 0)
        self.documentText = [[NSString alloc]
                    initWithData:contents
                    encoding:NSUTF8StringEncoding];
    else
        self.documentText = @"";

    return YES;
}
```

ドキュメントの更新を報告するデリゲートプロトコルの定義

ドキュメントの内容が変化したら、ドキュメントオブジェクトは、関係するほかのオブジェクトにその変更を通知しなければなりません。これを実行するには、デリゲートオブジェクトを使用する以外に方法はありません。

デリゲートオブジェクトのサポートにおける最初の手順は、そのオブジェクトが実装しなければならないメソッドを備えたプロトコルを定義することです。`STESimpleTextDocument`クラスの場合は、ドキュメントが中身の変更を開始したときに、そのことをデリゲートに知らせる必要があります。このアプリケーションでは、デリゲートは必ずView Controllerなので、ビューを更新する機会をデリゲートメソッドがView Controllerに与えます。

To : ドキュメントのデリゲートプロトコルを宣言するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STESimpleTextDocument.h`を選択します。
2. 新しいプロトコルを宣言して、`STESimpleTextDocumentDelegate`と名付けます。

   このプロトコル宣言を、`STESimpleTextDocument`クラスの宣言の後に配置します。
3. プロトコルに`documentContentsDidChange:`メソッドを追加します。

   このメソッドは、ドキュメントオブジェクトをパラメータとして使用し、戻り値はありません。この時点で、プロトコル定義は次のようになっているはずです。

```
@protocol STESimpleTextDocumentDelegate <NSObject>
@optional
- (void)documentContentsDidChange:(STESimpleTextDocument*)document;
@end
```

`STESimpleTextDocument`クラスの場合、ドキュメントが中身を変更するのは、ディスクからコンテンツを読み込んだときのみです。したがって、デリゲートメソッドを呼び出すように修正しなければならないメソッドは、`loadFromContents:ofType:error:`メソッドのみです。ドキュメント関係のその他の変更はすべてドキュメントの外部で発生するものなので、デリゲートメソッドは呼び出されません。

To : このドキュメントクラスをサポートするデリゲートを追加するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STESimpleTextDocument.h`を選択します。
2. `STESimpleTextDocumentProtocol`の前方宣言をヘッダファイルに追加します。

   この前方宣言を、`STESimpleTextDocument`クラスの宣言の前に配置します。この前方宣言は、コンパイラエラーを表示するために必要です。

```
@protocol STESimpleTextDocumentDelegate;
```

3. `delegate`プロパティを`STESimpleTextDocument`クラスに追加します。

   デリゲートオブジェクトは、`STESimpleTextDocumentDelegate`プロトコルに準拠した`id`タイプのオブジェクトへの弱い参照でなければなりません。したがって、プロパティ宣言は次のようになるはずです。

```
@property (weak, nonatomic) id<STESimpleTextDocumentDelegate> delegate;
```

4. プロジェクトナビゲータで、`STESimpleTextDocument.m`を選択します。
5. デリゲートプロパティを合成します。

```
@synthesize delegate = _delegate;
```

6. `loadFromContents:ofType:error:`メソッドの末尾に、`documentContentsDidChange:`デリゲートメソッドを呼び出すコードを追加します。

   デリゲートメソッドを呼び出すときは、デリゲートオブジェクトの存在を必ず調べ、指定されたセレクタにオブジェクトが応答することを確認します。この時点で、`loadFromContents:ofType:error:`メソッドの実装は次のようになるはずです。

```
- (BOOL)loadFromContents:(id)contents
               ofType:(NSString *)typeName
               error:(NSError *__autoreleasing *)outError {
    if ([contents length] > 0)
        self.documentText = [[NSString alloc] initWithData:contents

encoding:NSUTF8StringEncoding];
    else
        self.documentText = @"";

    // ドキュメントの中身の変化をデリゲートに伝える
    if (self.delegate && [self.delegate respondsToSelector:
                          @selector(documentContentsDidChange:)])
        [self.delegate documentContentsDidChange:self];

    return YES;
}
```

ドキュメントオブジェクト用のコードを入力したら、プロジェクトをビルドして、すべてのコンパイルを確認します。この時点では、ドキュメントオブジェクトがまだ使用されていないため、マスタと詳細が対になったデフォルトインターフェイスしか表示されません。

まとめ

この章では、ドキュメントクラスを宣言して、それを使用してファイルの中身を読み書きする方法を学びました。また、取り消しマネージャオブジェクトが自動保存機能を開始する仕組みと、デリゲートを使用して変更を報告する方法も学びました。最後に、デリゲートプロトコルを使用して、関係するほかのオブジェクトに変更を通知する方法も学びました。次の章では、作成したドキュメントを表示できるように、アプリケーションのインターフェイスの構築を始めます。

Master View Controllerの実装

この時点でアプリケーションはドキュメントクラスを備えているので、次は、ドキュメントを表示するインターフェイスの構築を始めます。Simple Text Editorアプリケーションには、View Controllerとビューをすべて格納したストーリーボードファイルが1つあります。このストーリーボードには、Navigation Controller、Master View Controller、Detail View Controllerが入っています。この章では、Master View Controllerとその関連コードを設定する方法に焦点を当てます。

Master View Controllerの設定は、次の手順を含むプロセスです。

1. ビューをセットアップします。
2. View Controllerのデータ構造体を設定します。
3. いくつかのテーブルデータソースメソッドを実装します。
4. 新しいドキュメントの場所を指定するコードを記述します。
5. ドキュメントのリストを編集するコードを記述します。

ビューの設定

Master View Controllerを設定する最初の手順として、アプリケーションのストーリーボードファイル内のビューを設定します。ストーリーボードファイルには、マスタシーンと詳細シーンが1つずつ入っています。マスタシーンのデフォルトコンテンツには、Table Viewとナビゲーション項目が1つずつ含まれています。

Table Viewは、当初、静的コンテンツを表示するように設定されているので、動的に生成されたコンテンツを表示するように変更しなければなりません。また、テーブルセルとDetail View Controller間のセグエを作成する必要もあります。

To : 動的コンテンツを表示するようにテーブルセルを設定するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`MainStoryboard.storyboard`を選択します。
2. 「Table View」を選択します。これはMaster View Controllerに埋め込まれています。
3. Attributesインスペクタでテーブルの属性を表示します。

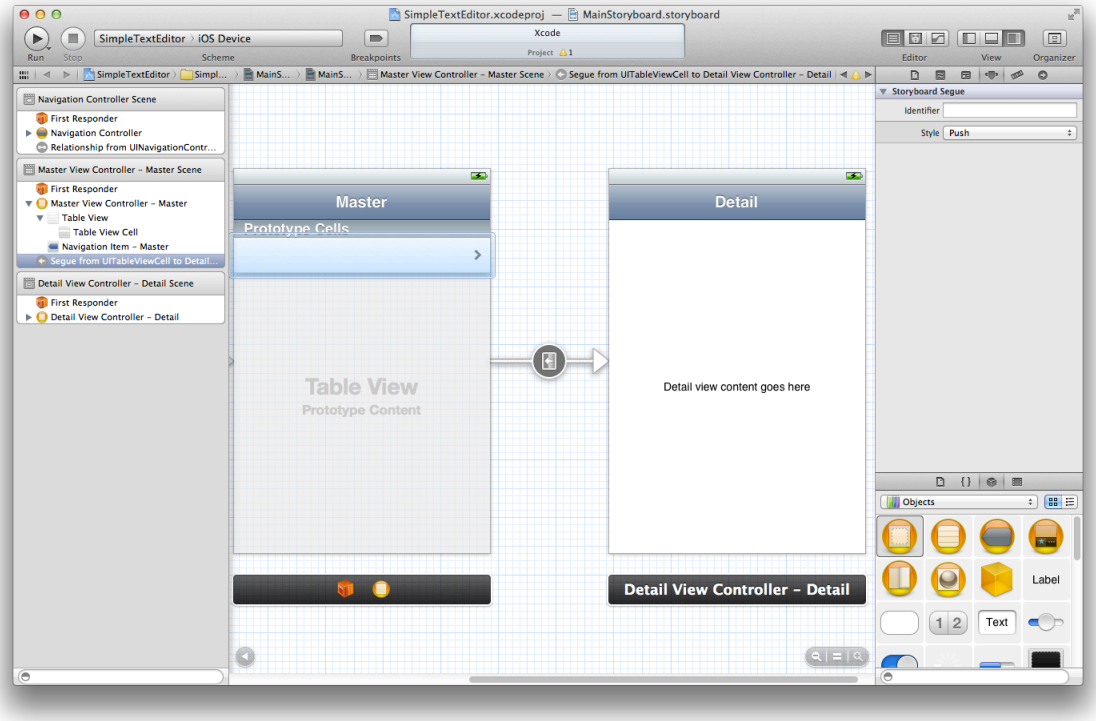
Table Viewは、デフォルトでは静的コンテンツを表示するように設定されています。テーブルデータソースから取得したコンテンツを表示するように、テーブルを変更する必要があります。

4. Contentフィールドに関して、値を「Dynamic Prototypes」に変更します。
5. Table Viewのセルを選択して、Attributesインスペクタを開きます。
6. セルのStyleフィールドを「Basic」に設定します。
7. Identifierフィールドの値を`DocumentEntryCell`に変更します。

   再利用識別子を設定して、セルの新たなインスタンスが作成されるようにしなければなりません。

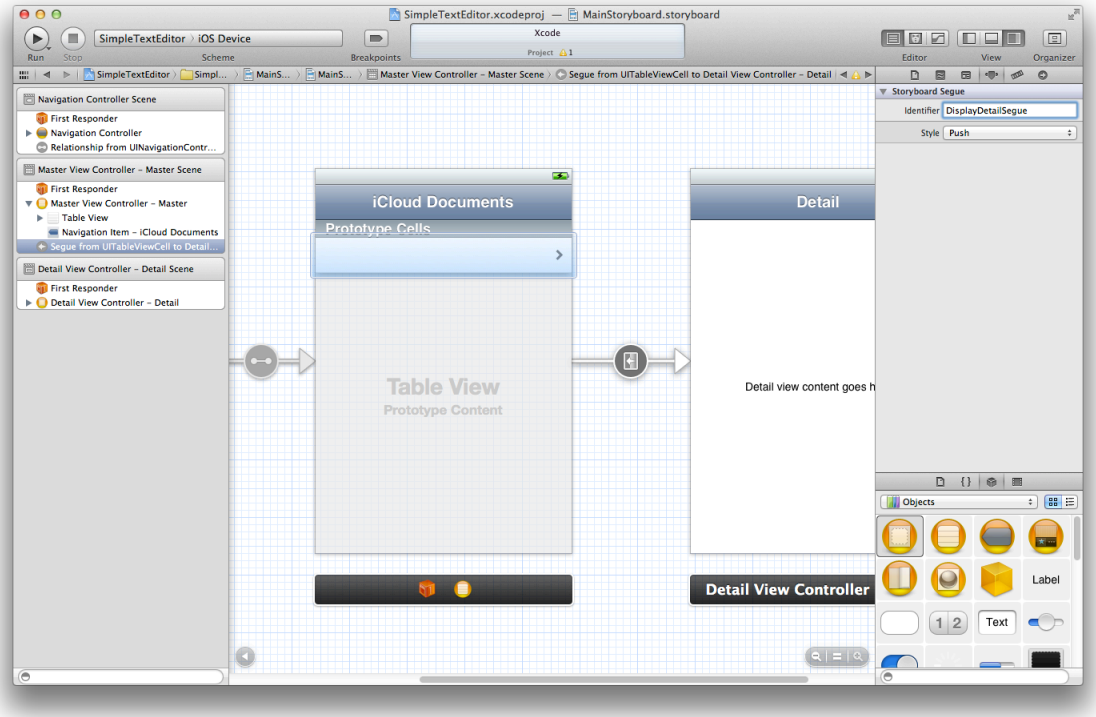
8. Table ViewのセルをControlキーを押しながらクリックし、詳細シーンまでドラッグして、セグエを作成します。

9. プロンプトが表示されたら、セグエのリストから「Push」を選択します。



10. セグエを選択して、Attributesインスペクタを開きます。
11. Identifierフィールドの値を`DisplayDetailSegue`に変更します。

ストーリーボードファイルでは、必ずセグエに名前を付けなければなりません。名前を使用すると、セグエをプログラムによって起動できるほか、同一のView Controllerから作成した複数のセグエを区別できます。

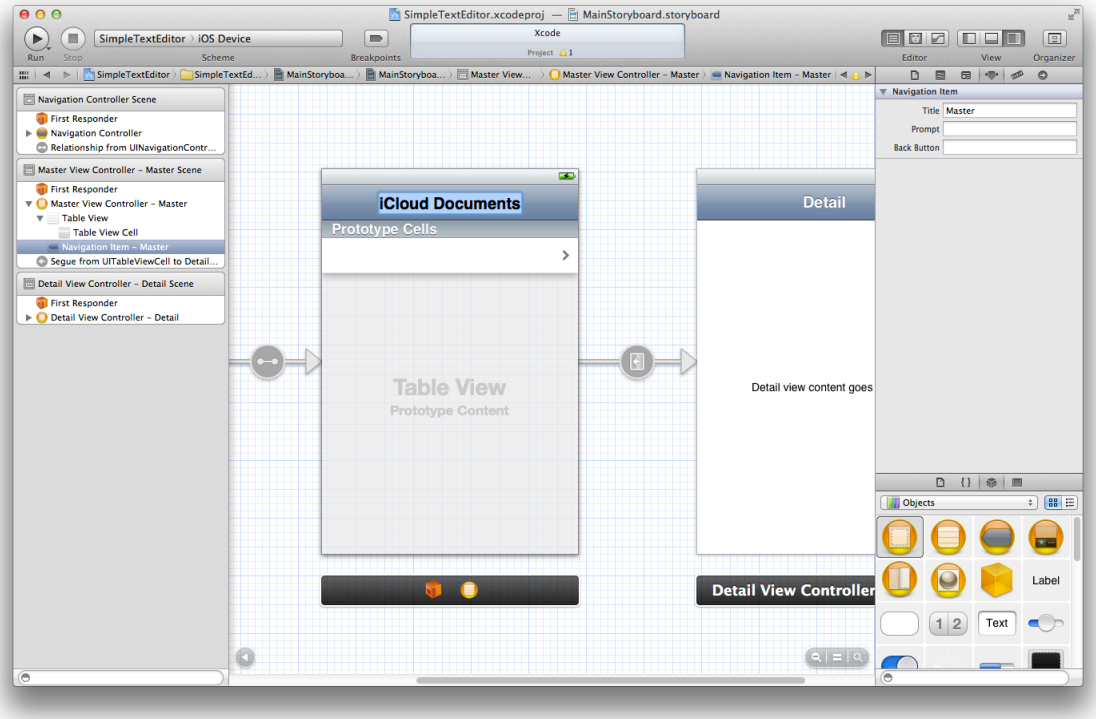


最後に行う細かな作業として、Master View Controllerのナビゲーション項目のタイトルを、このアプリケーションにもっと適したものに変更します。

To : ナビゲーション項目のタイトルを変更するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`MainStoryboard.storyboard`を選択します。
2. マスタシーンで、ナビゲーションバーにあるタイトルをダブルクリックして、編集可能にします。

3. タイトルを`iCloud Documents`に変更します。



アプリケーションのデータ構造体の実装

このアプリケーションでは、iCloud内で見つけたドキュメントのリストをMaster View ControllerのTable Viewに表示します。このリストは時間の経過に伴って変化する可能性があるので、アプリケーションはテーブルの内容を動的に提供しなければなりません。したがって、Master View Controllerは、利用可能なドキュメントを追跡するデータ構造体を管理しなければなりません。

Master View Controllerの基本データ構造体は、`NSURL`オブジェクトの配列で、それぞれのURLは、アプリケーションのiCloudコンテナディレクトリ内でのファイルの場所を表します。この配列は、`documents`というメンバ変数に格納されています。

To:MasterViewController用のデータ構造体を作成するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEMasterViewController.m`を選択します。
2. プライベート`documents`メンバ変数をクラスの実装に追加して、型を`NSMutableArray`に設定します。

   この時点で、クラス実装の先頭部分は次のようになっているはずです。

```
@implementation STEMasterViewController {
    NSMutableArray *documents;
}
```

3. 既存のawakeFromNibメソッドに次のコードを追加します。

```
if (!documents)
   documents = [[NSMutableArray alloc] init];
```

また、View Controllerの初期化メソッド内でこの配列を割り当てることもできます。単純化するため、ここの手順では、テンプレートプロジェクトが提供したawakeFromNibメソッドを使用します。

新規ドキュメントをテーブルに追加するための準備

新しいドキュメントを作成する前に実行しなければならない手順がいくつかあります。

1. 新規ドキュメントの名前を生成します。
2. 新規ドキュメントの場所を表すURLを作成します。
3. 新規ドキュメントボタンをUIに追加します。

これらの手順を済ませれば、アプリケーションのDetail View Controllerに後でドキュメントを作成する必要があるインフラストラクチャが完成します。

新規ドキュメントのデフォルト名の生成

ドキュメントを作成する前に、基本ファイルに付けたい名前を考えておきます。ユーザが作業を素早く始められるようにするため、アプリケーションには、適切なデフォルト名を選択することが期待されます。

Simple Text Editorアプリケーションは、簡単な基本ルールを使用して新規ドキュメントを生成します。newUntitledDocumentNameメソッドによって実装したこの基本ルールでは、動的に選択する整数を静的文字列と組み合わせてドキュメント名の候補を作成します。その後、この名前がアプリケーションの既存ドキュメント名と重複していないかどうかを調べます。このメソッドは、未使用であることが最初に判明した名前を返します。

To : ドキュメントのデフォルト名を生成するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STEMasterViewController.mを選択します。
2. STEDocFilenameExtension文字列定数を定義します。

   この定数は、ソースファイルの先頭で定義します。この定数を使用して、検索対象のファイルのタイプを指定します。今回のアプリケーションでは、カスタムファイル名拡張子を使用します。

```
NSString* STETextFileExtension = @"stedoc";
```

3. newUntitledDocumentNameメソッドを実装します。

```
- (NSString*)newUntitledDocumentName {
    NSInteger docCount = 1;     // 最初は1で、そこから増やしていく
    NSString* newDocName = nil;

    // この時点で、ドキュメントリストは最新の状態でなければならない
    BOOL done = NO;
    while (!done) {
        newDocName = [NSString stringWithFormat:@"Note %d.%@",
                      docCount, STEDocFilenameExtension];

        // 同一名の既存ドキュメントの有無を調べる
       // 同一名の既存ドキュメントが見つかった場合は、docCount値を増やして、同じ処理
を繰り返す
        BOOL nameExists = NO;
        for (NSURL* aURL in documents) {
            if ([[aURL lastPathComponent] isEqualToString:newDocName]) {
                docCount++;
                nameExists = YES;
                break;
            }
        }

        // 名前が見つからなかった場合は、ループを終了する
        if (!nameExists)
```

```
            done = YES;
        }
        return newDocName;
    }
```

このメソッドは、整数変数の値を増やして、`Note 1`、`Note 2`、`Note 3`といった一意のドキュメント名を作成します。このメソッドは、変数の値を連続的に増やす代わりに、必ず1から始めて、未使用の最初の名前を探します。この方法では、ドキュメント名を単純にしておくことができる上、大きな数値が含まれる可能性を排除できます。

新規ドキュメントのURLの構築

これで一意の名前を指定できるようになったので、ドキュメントの関連ファイル用のURLを作成するメソッドが必要です。方法の1つは、ファイルをローカルに作成し、`setUbiquitous:itemAtURL:destinationURL:error:`メソッドを使用してiCloudに移動することです。この方法は、ファイルの作成に有効な場所を確保できるため、出荷するアプリケーションに適しています。このチュートリアルでは、単純にするため、およびiCloud内のファイルしか扱わないため、Simple Text Editorアプリケーションはドキュメントを iCloudコンテナディレクトリ内に直接作成します。

URLの作成後、Master View Controllerは、テーブルを更新して、Detail View Controllerをナビゲーションスタックにプッシュします。次に、Detail View Controllerは、新しいURLに基づいたドキュメントオブジェクトの作成を処理します。`addDocument:`メソッドは、新規ドキュメントのURLの構築を担当するアクションメソッドです。

To : addDocument:メソッドを実装するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEMasterViewController.m`を選択します。
2. ソースファイルの先頭に、`STESimpleTextDocument`クラスを定義するヘッダファイルを加えます。

```
#import "STESimpleTextDocument.h"
```

3. 実行すべきセグエの`DisplayDetailSegue`文字列定数を定義します。

   この定数は、ソースファイルの先頭で定義します。この文字列値は、セグエの「Identifier」フィールドの値と一致していなければなりません。定数宣言は、次のようになっているはずです。

```
NSString* DisplayDetailSegue = @"DisplayDetailSegue";
```

4. iCloudの`Documents`ディレクトリ用に`STEDocumentsDirectoryName`文字列定数を定義します。

   この定数は、ソースファイルの先頭で定義します。定数宣言は、次のようになっているはずです。

```
NSString* STEDocumentsDirectoryName = @"Documents";
```

5. `addDocument:`メソッドをソースファイルに追加します。

   このメソッドは、ボタンにリンクできるように、アクションメソッドとして実装されています。このメソッドは、新規ドキュメントに適したURLを識別して、アプリケーションのデータ構造体を更新し、Detail Viewへのセグエの編集を開始します。

   `addDocument:`メソッドの次の実装をコードで使用します。

```
- (IBAction)addDocument:(id)sender {
    // ドキュメントの作成中、「Add」ボタンを無効にする
    self.addButton.enabled = NO;



dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
        // バックグラウンドキュー上に新たなURLオブジェクトを作成する
        NSFileManager *fm = [NSFileManager defaultManager];
        NSURL *newDocumentURL = [fm
URLForUbiquityContainerIdentifier:nil];


        newDocumentURL = [newDocumentURL
            URLByAppendingPathComponent:STEDocumentsDirectoryName
            isDirectory:YES];
        newDocumentURL = [newDocumentURL
           URLByAppendingPathComponent:[self newUntitledDocumentName]];


        // メインキュー上で残りのタスクを実行する
        dispatch_async(dispatch_get_main_queue(), ^{
            // データ構造体とテーブルを更新する
```

```
            [documents addObject:newDocumentURL];

            // テーブルを更新する
            NSIndexPath* newCellIndexPath =
            [NSIndexPath indexPathForRow:([documents count] - 1)
inSection:0];
            [self.tableView insertRowsAtIndexPaths:[NSArray
arrayWithObject:newCellIndexPath]

withRowAnimation:UITableViewRowAnimationAutomatic];

            [self.tableView selectRowAtIndexPath:newCellIndexPath
                                        animated:YES

scrollPosition:UITableViewScrollPositionMiddle];

            // Detail View Controllerへのセグエの編集を開始する
            UITableViewCell* selectedCell = [self.tableView
                                       cellForRowAtIndexPath:newCellIndexPath];
            [self performSegueWithIdentifier:DisplayDetailSegue
sender:selectedCell];

            // 「Add」ボタンを有効に戻す
            self.addButton.enabled = YES;
        });
    });
}
```

注意: 新しいSTESimpleTextDocumentオブジェクトの作成は、"Detail View Controllerの実装" （46 ページ）で説明するDetail View Controllerが処理します。

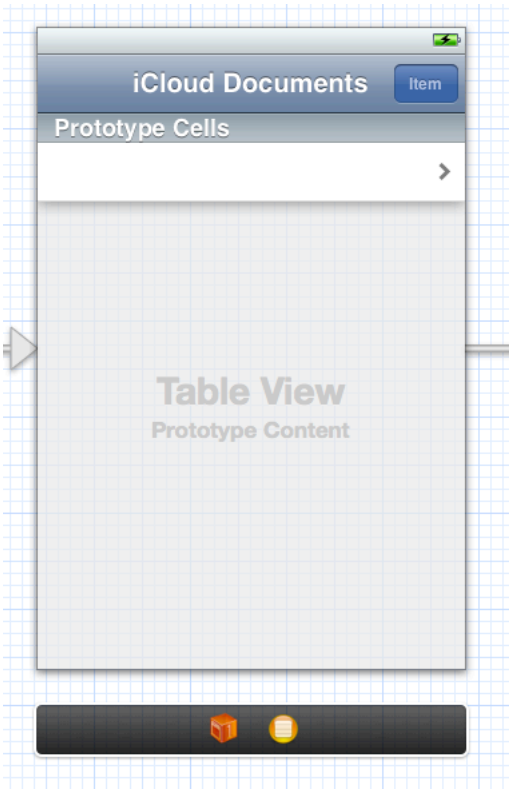
「New Document」ボタンの追加

この時点で、ドキュメント作成用のアクションメソッドを用意できたので、ユーザインターフェイスにボタンを追加して、メソッドを呼び出せるように設定する必要があります。

To :「New Document」ボタンをナビゲーションバーに追加するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`MainStoryboard.storyboard`を選択します。
2. フレームをスクロールして、マスタシーンが見えるようにします。
3. バーボタン項目をライブラリからドラッグし、ナビゲーションバーの右上隅にドロップします。

   この時点で、マスタシーンは次のようになっているはずです。



4. バーボタン項目を選択して、Attributesインスペクタを開きます。
5. ボタンの「Identifier」ポップアップメニューの値を「Add」に変更します。

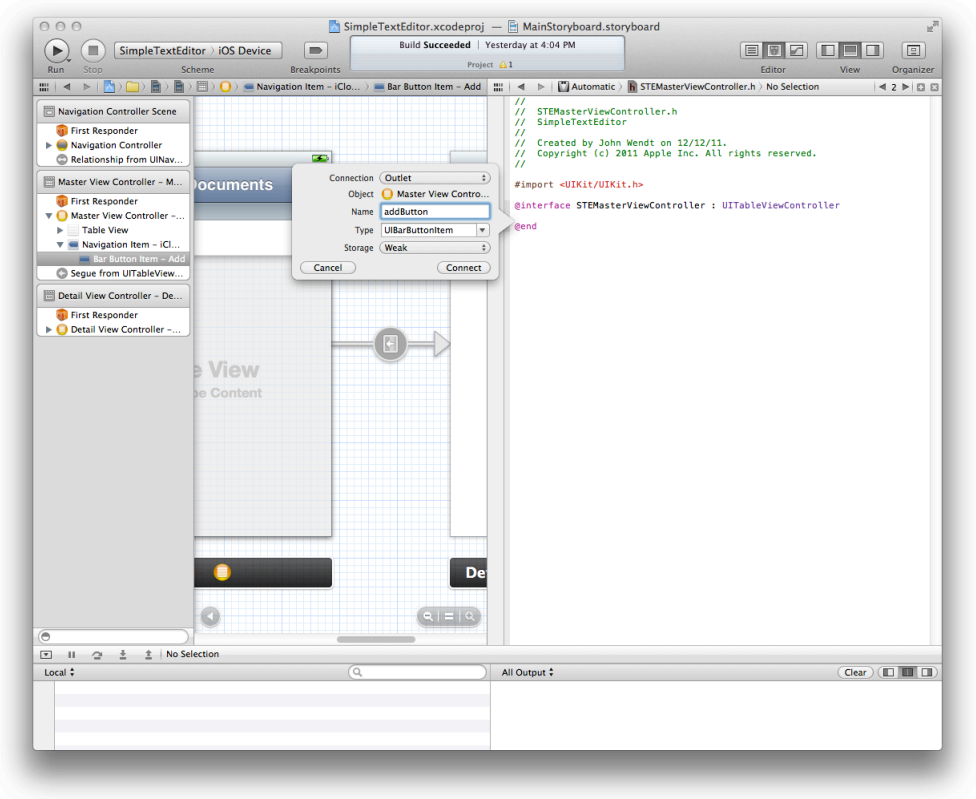
「Add」ボタンのタイプは、一般的なタスク用にシステムが備えている標準ボタンタイプの1つです。このタイプでは、ボタンの上に表示されるイメージが自動的にプラス記号に設定されます。

6. Assistant editorを表示します。

   プロジェクト上の「Assistant editor」ボタンをクリックすると、エディタが、ストーリーボードファイルと共に`STEMasterViewController.h`を表示します。

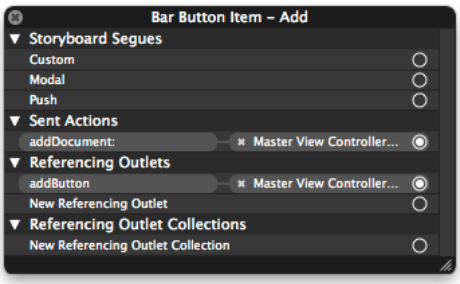
7. バーボタン項目をControlキーを押しながらクリックし、ヘッダファイルまでドラッグして、ボタン用のアウトレットを作成します。

アウトレットには`addButton`という名前を使用します。



8. 「Add」ボタンのアクションセレクタを、Master View Controllerの`addDocument:`メソッドに接続します。

   この時点で、「Add」ボタンの接続は次のようになっているはずです。



テーブルデータソースメソッドの実装

新規ドキュメントを作成することはできますが、マスタシーンのTable Viewにそのドキュメントは表示されません。Table Viewはデータを動的に取得するので、テーブルのデータソースメソッド、特に`tableView:numberOfRowsInSection:`と`tableView:cellForRowAtIndexPath:`データソースメソッドを実装する必要があります。最初のメソッドは利用可能なドキュメントの数を報告し、2つ目のメソッドは、表示すべき実際のテーブルセルオブジェクトを提供します。

Simple Text Editorでは、テーブル内の行数は、`documents`配列内のエントリ数に等しくなります。したがって、`tableView:numberOfRowsInSection:`メソッドの実装は、配列内のエントリ数を返すだけで済みます。

To : tableView:numberOfRowsInSection:メソッドを実装するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEMasterViewController.m`を選択します。
2. `tableView:numberOfRowsInSection:`メソッドを次のように実装します。

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section {
    return [documents count];
}
```

データソースは、行ごとに、ドキュメントの名前を表示するTable Viewセルを提供します。Simple Text Editorアプリケーションは、テーブルのセルにデフォルトスタイルを使用します。

To : tableView:cellForRowAtIndexPath:メソッドを実装するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEMasterViewController.m`を選択します。
2. Table Viewのセルの再利用識別子の`DocumentEntryCell`文字列定数を定義します。

   この定数は、これまでの手順で定義したその他の定数と共に、ソースファイルの先頭で定義します。この定数の値は、必ず文字列`DocumentEntryCell`です。この文字列は、プロトタイプセルの「Identifier」フィールドに割り当てた値と一致しなければなりません。定数宣言は、次のようになっているはずです。

```
NSString* DocumentEntryCell = @"DocumentEntryCell";
```

3. `tableView:cellForRowAtIndexPath:`メソッドを次のように実装します。

```
- (UITableViewCell*)tableView:(UITableView *)tableView
                     cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *newCell = [tableView
dequeueReusableCellWithIdentifier:DocumentEntryCell];
    if (!newCell)
        newCell = [[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault

reuseIdentifier:DocumentEntryCell];


    if (!newCell)
        return nil;


    // 指定された行にあるドキュメントを取得する
    NSURL *fileURL = [documents objectAtIndex:[indexPath row]];


    // セルを設定する
    newCell.textLabel.text = [[fileURL lastPathComponent]
stringByDeletingPathExtension];
    return newCell;
}
```

このメソッドを実装する標準的な方法は、既存のテーブルセルをリサイクルし（または新しいセルを1つ作成し）、そのセルの値を設定して、返すことです。このアプリケーションの場合、セルのインデックスは、必ず、`documents`配列内の該当URLのインデックスに対応します。この対応関係のおかげで、URLを取得して、対応するファイル名をセルのラベルに割り当てる処理が容易になります。

ドキュメントのリストの編集

Master View Controllerに追加すべき最後の機能が1つ残っています。テーブルの行を削除する機能のサポートです。Table Viewに組み込まれている編集機能を使用して、行を削除します。この機能のおかげで、後でアプリケーションを簡単にテストできます。

最初の手順は、ユーザがタップするとテーブルを編集モードにできるボタンを追加することです。UIViewControllerクラスは、使用すると「Edit」ボタンを実装できる事前設定済みのバーボタン項目を備えています。

To :「Edit」ボタンをナビゲーションバーに追加するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STEMasterViewController.mを選択します。
2. awakeFromNibメソッドに次のコード行を追加します。

```
self.navigationItem.leftBarButtonItem = self.editButtonItem;
```

ユーザが「Edit」ボタンをタップすると、ボタンがView Controllerのsetediting:animated:メソッドを呼び出して、編集プロセスを開始します。同時に、ボタンの外観が「Done」ボタンの外観に変化し、編集モードでの変更に関するフィードバックをユーザに提供します。「Done」ボタンをタップすると、編集モードは終了します。

編集モードになっているとき、Master View Controllerは、テーブルにコントロールを追加して、個々の行を削除できるようにします。コントロールが表示され、ユーザがタップできる状態ですが、実際にタップしても、基本ファイルはまだ削除されません。ファイルの削除を処理するためには、tableView:commitEditingStyle:forRowAtIndexPath:データソースメソッドを実装する必要があります。このメソッドの実装は、ファイルを削除して、テーブルのデータ構造体を更新しなければなりません。

To : テーブルの行の削除を処理するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STEMasterViewController.mを選択します。
2. tableView:commitEditingStyle:forRowAtIndexPath:メソッドの実装をコードに追加します。

   Xcodeプロジェクトは、すぐに作業を始められるように、このメソッドのコメントアウトバージョンを備えています。コメントアウトバージョンを、ここで作成した実装に置き換えてください。この実装は、ファイルコーディネータを使用して、バックグラウンドのスレッドにあるファイルを削除し、削除をテーブルに通知する前にアプリケーションのデータ構造体を更新します。Table Viewは、行を削除するように指示されたときまでに、データソースが最新状態になっていると期待するため、最後の2つの処理の順序は重要です。

```
- (void)tableView:(UITableView *)tableView
        commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
        forRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        NSURL *fileURL = [documents objectAtIndex:[indexPath row]];

        // アプリケーションのメインキューでファイルコーディネータを使用してはならな
い

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
 0), ^{
            NSFileCoordinator *fc = [[NSFileCoordinator alloc]
                                     initWithFilePresenter:nil];
            [fc coordinateWritingItemAtURL:fileURL
                options:NSFileCoordinatorWritingForDeleting
                error:nil
                byAccessor:^(NSURL *newURL) {
                    NSFileManager *fm = [[NSFileManager alloc] init];
                    [fm removeItemAtURL:newURL error:nil];
            }];
        });

        // ドキュメントの配列からURLを削除する
        [documents removeObjectAtIndex:[indexPath row]];

        // テーブルUIを更新する。これは、
        // ドキュメント配列の更新後でなければならない
        [tableView deleteRowsAtIndexPaths:[NSArray
arrayWithObject:indexPath]
                    withRowAnimation:UITableViewRowAnimationAutomatic];
    }
}
```

iCloudに格納されたファイルを削除または移動するときは、必ずNSFileCoordinatorオブジェクトを使用してください。ファイルコーディネータオブジェクトを自分で作成しなければならない機会は少数ですが、これはその1つです。それ以外のほとんどの場合は、ドキュメントオブジェクトがこのオブジェクトを作成し、それを使用して、該当するファイル関連操作を実行してくれます。ファイルコーディネータを使用すると、ファイルを削除している間にiCloudサービスがそのファイルを修正す

るということがなくなります。また、ファイルを削除したときに、そのことが必ずiCloudサービスに通知されます。ファイルコーディネータの使い方については『*File System Programming Guide*』を参照してください。

まとめ

この章では、Table ViewをMaster View Controller用に設定し、ドキュメントのリストを作成および編集できるようにするコントロールを追加しました。また、iCloud内のドキュメントファイルの作成と削除に必要なコードを実装しました。最後に、Detail View Controllerを表示することでドキュメントの選択内容に応答するストーリーボードセグエを設定しました。次の章では、Detail View Controllerを使用してドキュメントの中身を表示および編集する方法を学びます。

Detail View Controllerの実装

Master View ControllerでURLを生成できるようになりましたが、これらのURLで実際のファイルを作成および管理する手段が必要です。このXcodeプロジェクトには、Master View Controllerのテーブルで選択された項目を表示するために、Detail View Controllerがあります。この章では、Detail View Controllerを使用してドキュメントオブジェクトを作成し、ドキュメントの現在のテキストを表示し、そのテキストを編集する手段を提供し、テキストをディスクに保存します。

Detail Viewの設定

Detail View Controller用に用意されたデフォルトビューには、単一のラベルオブジェクトが入っています。テキスト編集の場合、このラベルは十分ではなく、複数行のテキスト入力をサポートしたText Viewに置き換える必要があります。

ビュー自体をセットアップする前に、Detail View Controllerのソースファイルを用意する必要があります。クラスを`STESimpleTextDocumentDelegate`プロトコルに準拠させる必要があるほか、ラベルオブジェクトを使用しないことを考慮して、いくつかのクリーンアップ処理も行う必要があります。

To : Text View用のソースファイルを用意するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEDetailViewController.h`を選択します。
2. `STESimpleTextDocument.h`ヘッダファイル用のインポートステートメントを追加します。

```
#import "STESimpleTextDocument.h"
```

3. Detail View Controllerを`STESimpleTextDocumentDelegate`プロトコルに準拠させます。
4. 既存の`detailItem`プロパティのタイプを`id`から`NSURL*`に変更します。

   Master View Controllerは、`NSURL`オブジェクトをDetail View Controllerに渡します。Detail View Controllerは、このURLを使用して、対応するドキュメントオブジェクトを作成します。

5. プロジェクトナビゲータで、`STEDetailViewController.m`を選択します。
6. 型が`STESimpleTextDocument`クラスのプライベート`_document`メンバ変数を追加します。

   このメンバ変数は、ドキュメントオブジェクトを格納するためにプライベートに使用されます。この時点で、クラス実装の先頭部分は次のようになっているはずです。

```
@implementation STEDetailViewController {
    STESimpleTextDocument* _document;
}
@synthesize detailItem = _detailItem;
@synthesize detailDescriptionLabel = _detailDescriptionLabel;
```

7. `setDetailItem:`メソッドの実装をコメントアウトします。

   `setDetailItem:`アクセサメソッドのカスタム実装によって、ラベルが更新されます。このラベルは使用しないので、このカスタムメソッドをコメントアウトして、コンパイラが生成するアクセサメソッドのみを使用してもかまいません。

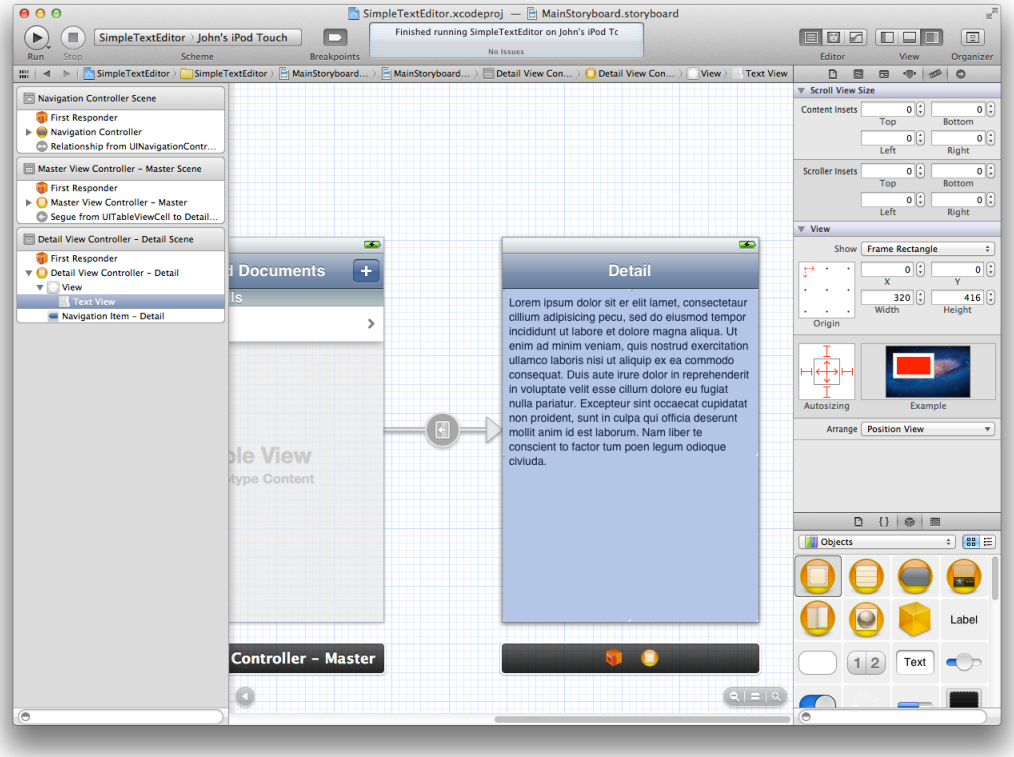
これでソースファイルの設定が終わったので、Text Viewをストーリーボードファイルに追加して、適切なアウトレットを接続することができます。Detail View Controllerは、Text View用のアウトレットを備えているだけでなく、Text Viewのデリゲートの役割も果たします。ストーリーボードには、両方のアウトレットを設定しなければなりません。

To : Text Viewをストーリーボードファイルに追加するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`MainStoryboard.storyboard`を選択します。
2. 詳細シーンで、ビューに付属しているデフォルトラベルを削除します。

   このチュートリアルでは、ラベルは不要です。
3. Text Viewオブジェクトを Detail Viewまでドラッグします。

基盤のContent Viewの端に合わせて、Text Viewを配置します。ナビゲーションバーの下にあるスペース全体が埋まるように、Text Viewを配置します。Text Viewオブジェクトの自動サイズ設定オプションが、基盤のContent Viewに合わせてサイズ変更されるようにすでに設定されているはずです。

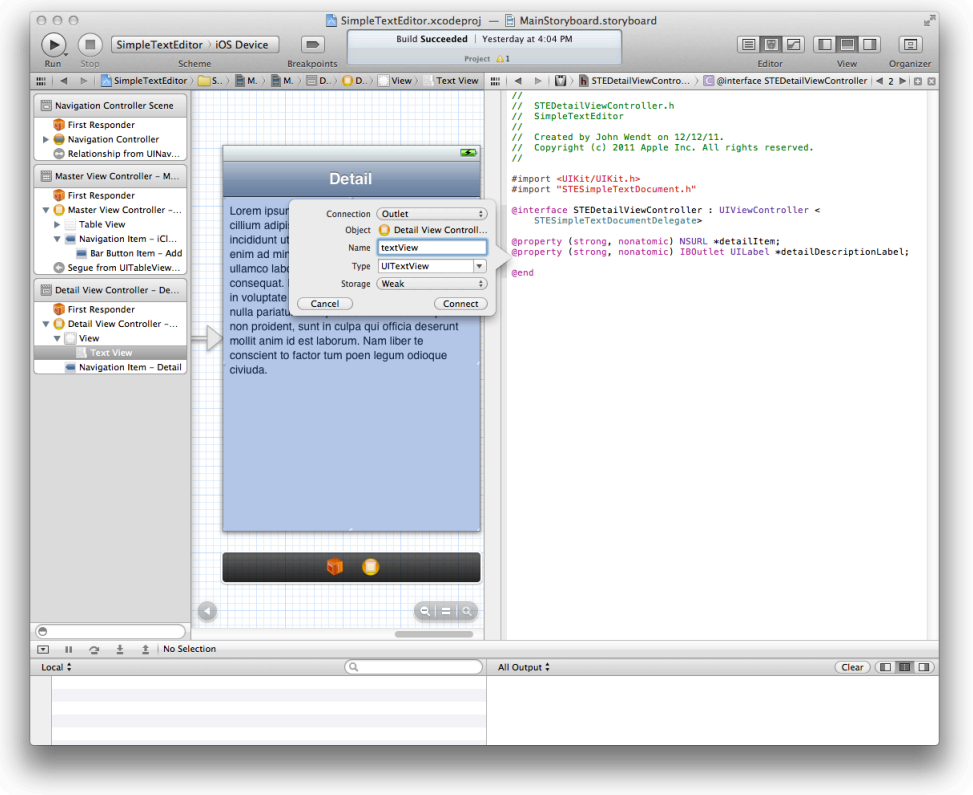


4. Assistant editorを表示します。

   プロジェクト上の「Assistant editor」ボタンをクリックすると、エディタが、ストーリーボードファイルと共に`STEMasterViewController.h`を表示します。

5. Text ViewをControlキーを押しながらクリックし、ヘッダファイルまでドラッグして、このビュー用のアウトレットを作成します。

アウトレットにはtextViewという名前を使用します。



6. Text Viewオブジェクトのdelegateアウトレットを`STEDetailViewController`クラスに接続します。

   `STEDetailViewController`オブジェクトは、Text Viewのデリゲートの役割を果たします。

セグエの用意

Master View Controllerのテーブルにあるセルをユーザがタップすると、関連付けられたセグエが新しいDetail View Controllerをロードして、ユーザに表示します。ただし、Detail View Controllerが実際に表示される前に、Master View Controllerの`prepareForSegue:sender:`メソッドが呼び出されて、必要なデータをDetail View Controllerに渡す機会を与えます。

Simple Text Editorアプリケーションは、`prepareForSegue:sender:`メソッドを使用して、選択されたURLオブジェクトをDetail View Controllerに割り当てます。

To : セグエ中にDetail View Controllerを設定するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEMasterViewController.m`を選択します。

2. インポートされたヘッダのリストに`STESimpleTextDocument.h`ヘッダファイルを追加します。

```
#import "STEDetailViewController.h"
```

3. `prepareForSegue:sender:`メソッドの実装を自分の実装に追加します。

   このメソッドの実装は、`sender`パラメータに渡されたTable Viewのセルを使用して、ドキュメントのURLを識別します。テーブル内でのそのセルの位置によって、`documents`配列から取り出すURLを判断し、Detail View Controllerの`detailItem`プロパティに割り当てます。

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if (![segue.identifier isEqualToString:DisplayDetailSegue])
        return;

    // Detail View Controllerを取得する
    STEDetailViewController* destVC =
           (STEDetailViewController*)segue.destinationViewController;

    // ドキュメント配列から正しいディクショナリを検索する
    NSIndexPath *cellPath = [self.tableView indexPathForSelectedRow];
    UITableViewCell *theCell = [self.tableView
cellForRowAtIndexPath:cellPath];
    NSURL *theURL = [documents objectAtIndex:[cellPath row]];

    // URLをDetail View Controllerに割り当て、
    // View Controllerのタイトルをドキュメント名に設定する
    destVC.detailItem = theURL;
    destVC.navigationItem.title = theCell.textLabel.text;
}
```

これで、Detail View Controllerが適切なURLへの参照を備えましたが、ドキュメントを作成して、その中身を開くか新規作成する必要があります。View Controllerは、そのビューが画面に表示されるときにこの処理を行います。

To : Text View用の初期テキストを設定するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEDetailViewController.m`を選択します。
2. `viewWillAppear:`メソッドを更新して、ドキュメントオブジェクトを作成します。

   Text Viewを削除した後に行う最初の手順は、ドキュメントオブジェクトを作成して、Detail View Controllerをドキュメントのデリゲートとして割り当てることです。その後のドキュメント作成プロセスは、既存ドキュメントを開くのか新規ドキュメントを作成するのかによって異なります。既存ドキュメントの場合は、`openWithCompletionHandler:`メソッドを呼び出して、そのドキュメントの既存の内容をロードします。新規ドキュメントの場合は、`saveToURL:forSaveOperation:completionHandler:`メソッドを呼び出して、ディスク上にファイルを作成します。

   `viewWillAppear:`メソッドの実装は、次のように見えるはずです。

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    // Text Viewの内容を削除します。
    self.textView.text = @"";

    // ドキュメントを作成して、デリゲートを割り当てる
    _document = [[STESimpleTextDocument alloc]
initWithFileURL:self.detailItem];
    _document.delegate = self;

    // ファイルが存在する場合はそれを開き、それ以外の場合は作成する
    NSFileManager *fm = [NSFileManager defaultManager];
    if ([fm fileExistsAtPath:[self.detailItem path]])
        [_document openWithCompletionHandler:nil];
    else
        // 新しいドキュメントをディスクに保存する
        [_document saveToURL:self.detailItem
                  forSaveOperation:UIDocumentSaveForCreating
                  completionHandler:nil];
}
```

既存ドキュメントを開くときは、通常、そのドキュメントの中身をディスクから読み込むのに少し時間がかかります。Detail View Controllerは、ドキュメントのロード中、対応するビューの表示を遅らせる代わりに、そのビューを通常どおりに表示します。当初、Text Viewには中身がありません。ただし、Detail View Controllerが自らをドキュメントのデリゲートとして割り当てるので、ドキュメントの内容のロードが完了したら、そのことが通知されます。Detail View Controllerは、次に、`documentContentsDidChange:`デリゲートメソッドを使用して、Text Viewを更新します。

To : ドキュメントのロードが完了したときにText Viewを更新するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEDetailViewController.m`を選択します。
2. `documentContentsDidChange:`メソッドの実装を追加します。

   このメソッドが呼び出されるときまでに、変更後のテキストがドキュメントオブジェクトに入ります。したがって、このメソッドの実装は、そのテキストをText Viewに割り当てなければなりません。このデリゲートメソッドはどのスレッドでも呼び出せるので、このメソッドには、アプリケーションのメインキューからコードが実行されるようにするための明示的なディスパッチ呼び出しが入っています。

```
- (void)documentContentsDidChange:(STESimpleTextDocument *)document {
    dispatch_async(dispatch_get_main_queue(), ^{
        self.textView.text = document.documentText;
    });
}
```

ユーザが編集を終えたときのドキュメントの保存

ユーザが戻るボタンをタップしてドキュメントのリストに戻ると、Detail View Controllerは、現在の変更内容をドキュメントオブジェクトに書き戻してから、ドキュメントを閉じます。ドキュメントが閉じると、変更内容が非同期にディスクに書き出されます。

To : 変更内容をドキュメントに保存するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEDetailViewController.m`を選択します。
2. `viewWillDisappear:`メソッドの実装を更新して、現在のテキストをドキュメントオブジェクトにコピーし、ドキュメントを閉じます。

```
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];

    NSString* newText = self.textView.text;
    _document.documentText = newText;

    // ドキュメントを閉じる
    [_document closeWithCompletionHandler:nil];
```

`viewWillDisappear:`メソッドは、Navigation ControllerのスタックからDetail View Controllerが削除される直前に呼び出されます。

キーボード通知の処理

キーボードが表示されると、そのキーボードはアプリケーションのウインドウの上部にスライドし、その下にあるコンテンツを隠します。Simple Text Editorアプリケーションでは、Text Viewの大きな部分が隠れてしまします。キーボードの下にカーソルが移動するほどの量のテキストをユーザが入力した場合は、これが問題になります。ユーザのテキストがすべて見えるようにするため、キーボードが表示されたときに、Text Viewのコンテンツ挿入枠を調整することができます。

まず、キーボードの表示と消去のタイミングを判断します。そのため、キーボード通知を受け取るように登録します。

To : キーボード通知を受け取るように登録するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEDetailViewController.m`を選択します。
2. `viewWillAppear:`メソッドで、キーボード通知を受け取るように登録します。

   変更をキーボードアニメーションと同期できるようにするため、Detail View Controllerは、`UIKeyboardWillShowNotification`と`UIKeyboardWillHideNotification`の両方の通知を受け取るように登録する必要があります。`viewWillAppear:`メソッドの末尾で、キーボード通知を受け取るように登録します。この時点で、メソッドの実装は次のようになっているはずです。

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
```

```
    // Text Viewの内容を削除します。
    self.textView.text = @"";

    // ドキュメントを作成して、デリゲートを割り当てる
    _document = [[STESimpleTextDocument alloc]
initWithFileURL:self.detailItem];
    _document.delegate = self;

    // ファイルが存在する場合はそれを開き、それ以外の場合は作成する
    NSFileManager *fm = [NSFileManager defaultManager];
    if ([fm fileExistsAtPath:[self.detailItem path]])
        [_document openWithCompletionHandler:nil];
    else
        // 新しいドキュメントをディスクに保存する
        [_document saveToURL:self.detailItem
                  forSaveOperation:UIDocumentSaveForCreating
                  completionHandler:nil];

    // 通知の受け取りを登録する
    [[NSNotificationCenter defaultCenter] addObserver:self
                        selector:@selector(keyboardWillShow:)
                        name:UIKeyboardWillShowNotification
                        object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
                        selector:@selector(keyboardWillHide:)
                        name:UIKeyboardWillHideNotification
                        object:nil];
}
```

3. ビューが消えたら、キーボード通知の受け取りを登録解除します。

   View Controllerが存在しないときに通知センターがコードを呼び出すのを防止するため、キーボード通知のオブザーバを削除する必要があります。`viewWillDisappear:`メソッドで、キーボード通知の受け取りを登録解除します。この時点で、実装は次のようになっているはずです。

```
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];

    NSString* newText = self.textView.text;
    _document.documentText = newText;

    // ドキュメントを閉じる
    [_document closeWithCompletionHandler:nil];

    // 通知の受け取りを登録解除する
    [[NSNotificationCenter defaultCenter] removeObserver:self
          name:UIKeyboardWillShowNotification
          object:nil];
    [[NSNotificationCenter defaultCenter] removeObserver:self
          name:UIKeyboardWillHideNotification
          object:nil];
}
```

`UIKeyboardWillShowNotification`通知は、キーボードが所定の位置にアニメーション化される直前に送信されます。Detail View Controllerクラスは、この通知を使用して、Text Viewのコンテンツ挿入枠の変化をアニメーション化します。下部の挿入枠を追加すると、Text Viewの上部に向かってテキストが移動します。

To : キーボードが表示されたときにText Viewを調整するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEDetailViewController.m`を選択します。
2. `keyboardWillShow:`メソッドの次の実装をコードに追加します。

   必ず、通知が提供するキーボードサイズを使用してください。キーボードの位置とサイズは、現在の言語と入力方式によって異なります。このメソッドは下部挿入枠に高さを追加し、それに応じてテキストが上方移動します。キーボードアニメーションと同じ速度で、挿入枠への変更が所定の場所でアニメーション化されます。

```
- (void)keyboardWillShow:(NSNotification*)aNotification {
    NSDictionary* info = [aNotification userInfo];
    CGRect kbSize = [[info objectForKey:UIKeyboardFrameEndUserInfoKey]
```

```
                                CGRectValue];
    double duration = [[info
objectForKey:UIKeyboardAnimationDurationUserInfoKey]
                                doubleValue];

    UIEdgeInsets insets = self.textView.contentInset;
    insets.bottom += kbSize.size.height;

    [UIView animateWithDuration:duration animations:^{
        self.textView.contentInset = insets;
    }];
}
```

.Text View全体にコンテンツを表示できるようにするため、キーボードが消える前に、コンテンツ挿入枠を元の値に戻す必要があります。

To : キーボードが非表示になったときにText Viewのコンテンツ挿入枠を元に戻すには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEDetailViewController.m`を選択します。
2. `keyboardWasShown:`メソッドの次の実装をコードに追加します。

   このメソッドが下部挿入枠を0に戻すので、Text Viewの下端までテキストが広がります。

```
- (void)keyboardWillHide:(NSNotification*)aNotification {
    NSDictionary* info = [aNotification userInfo];
    double duration = [[info
objectForKey:UIKeyboardAnimationDurationUserInfoKey]
                                   doubleValue];

    // Text Viewの下部コンテンツ挿入枠をリセットする
    UIEdgeInsets insets = self.textView.contentInset;
    insets.bottom = 0;

    [UIView animateWithDuration:duration animations:^{
        self.textView.contentInset = insets;
```

```
    }];
}
```

ビューを変更してキーボードを表示するときは、必ずアニメーションを使用することをお勧めします。このようにすると、コンテンツの外観が急に変化するのを避けることができます。

まとめ

この章では、ドキュメントオブジェクトを作成し、それを使用して、対応するファイルを開くか作成しました。また、変更内容をドキュメントに保存し、セグエを用意する方法も学びました。次の章では、作成済みのドキュメントを検索する方法を学びます。

iCloudドキュメントの検索

あるデバイス上のアプリケーションが作成したドキュメントは、そのユーザのほかのデバイスで実行されている同一アプリケーションでも見つけ出せなければなりません。アプリケーションは、`NSMetadataQuery`クラスのインスタンスであるメタデータクエリオブジェクトを使用して、iCloudコンテナディレクトリ内のドキュメントを見つけ出します。メタデータクエリオブジェクトは、指定した基準に合ったファイルを求めて、利用可能なiCloudコンテナディレクトリを検索します。

ドキュメントはコンテナディレクトリにいつでも表示できるので（またはこのディレクトリからいつでも非表示にできるので）、たいていの場合は、クエリを開始して実行したままにしておくと便利です。その場合は、アプリケーションの中央部分にクエリオブジェクトを作成して、アプリケーションが実行されている間ずっと、そのオブジェクトへの参照を維持しなければなりません。

ドキュメントの検索の開始

`STEMasterViewController`クラスは、アプリケーションの実行中の早い時期にデータクエリオブジェクトを作成して、起動します。このクエリは、ドキュメントの初期リストを即座に返し、後で変化が発生すると更新内容を送信します。

このクエリは、アプリケーションの実行中ずっと動作しているので、`STEMasterViewController`クラスは、インスタンス変数を使用してクエリオブジェクトを常に格納します。

To : メタデータクエリオブジェクトのストレージを宣言するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEMasterViewController.m`を選択します。
2. `_query`というプライベートインスタンス変数を追加して、メタデータクエリオブジェクトを格納できるようにします。

   この変数の型は、`NSMetadataQuery`へのポインタです。この時点で、クラス実装の先頭部分は次のようになっているはずです。

```
@implementation STEMasterViewController {
    NSMutableArray *documents;
    NSMetadataQuery *_query;
```

```
}
@synthesize addButton;
```

アプリケーションの起動サイクル中に、_queryインスタンス変数を初期化します。

To : 初期化時に検索クエリを設定および起動するには、以下の手順を実行します。

1. プロジェクトナビゲータで、STEMasterViewController.mを選択します。
2. textDocumentQueryというメソッドを定義して、クエリオブジェクトを作成できるようにします。

   このメソッドの実装は、新しいNSMetadataQueryオブジェクトを作成し、アプリケーションのiCloudコンテナのDocumentsディレクトリにあるテキストドキュメントのみを検索するように設定します。このメソッドの実装は次のようになります。

```
- (NSMetadataQuery*)textDocumentQuery {
    NSMetadataQuery* aQuery = [[NSMetadataQuery alloc] init];
    if (aQuery) {
        // Documentsサブディレクトリのみを検索する
        [aQuery setSearchScopes:[NSArray

arrayWithObject:NSMetadataQueryUbiquitousDocumentsScope]];


        // ドキュメントを検索するための述語を追加する
        NSString* filePattern = [NSString stringWithFormat:@"*.%@",
                    STEDocFilenameExtension];
        [aQuery setPredicate:[NSPredicate predicateWithFormat:@"%K LIKE
 %@",
                    NSMetadataItemFSNameKey, filePattern]];
    }


    return aQuery;
}
```

3. setupAndStartQueryという新しいメソッドを定義して、クエリを開始できるようにします。

   setupAndStartQueryメソッドは、メタデータクエリオブジェクトを取得し、通知ハンドラを設定して、クエリを開始します。このメソッドの実装は次のようになります。

```
- (void)setupAndStartQuery {
    // まだ存在しない場合は、クエリオブジェクトを作成する
    if (!_query)
        _query = [self textDocumentQuery];

    // メタデータクエリ通知を受け取るように登録する
    [[NSNotificationCenter defaultCenter] addObserver:self
          selector:@selector(processFiles:)
          name:NSMetadataQueryDidFinishGatheringNotification
          object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
          selector:@selector(processFiles:)
          name:NSMetadataQueryDidUpdateNotification
          object:nil];

    // クエリを開始して、実行状態のままにする
    [_query startQuery];
}
```

4. `awakeFromNib`メソッドを更新して、`setupAndStartQuery`メソッドを呼び出すようにします。この時点で、`awakeFromNib`メソッドは次のようになっているはずです。

```
- (void)awakeFromNib {
    [super awakeFromNib];

    if (!documents)
        documents = [[NSMutableArray alloc] init];

    self.navigationItem.leftBarButtonItem = self.editButtonItem;
    [self setupAndStartQuery];
}
```

検索結果の処理

検索結果は、2つのフェーズで生成されます。最初の結果収集フェーズと、更新フェーズです。最初の結果収集フェーズは、メタデータクエリの開始直後に発生し、返されるべき最初の結果セットです。その後、このクエリオブジェクトは、ディレクトリ内のファイルが変化したときのみ、更新通知を送信します。最初の結果収集フェーズと更新フェーズのどちらでも、メタデータクエリの結果には、指定された場所でそれまでに発見したすべてのファイルが含まれます。

Simple Text Editorアプリケーションは、メタデータクエリ通知を使用してドキュメントのリストを更新し、それに応じてユーザインターフェイスを更新します。

To : ドキュメントのリストを処理するには、以下の手順を実行します。

1. プロジェクトナビゲータで、`STEMasterViewController.m`を選択します。
2. カスタム`processFiles:`メソッドの次の実装を追加します。

   このメソッドは、隠すべきファイルを除外した上で、発見されたファイルのリストを作成します。検索クエリの結果の処理時、ファイルのリストを処理している間は、更新を一時的に無効にします。このようにすると、検索の反復中に検索結果が変化するのを防止できます。

```
- (void)processFiles:(NSNotification*)aNotification {
    NSMutableArray *discoveredFiles = [NSMutableArray array];

    // 結果の処理時には更新を必ず無効にする
    [_query disableUpdates];

    // クエリは、常に、発見したすべてのファイルを報告する
    NSArray *queryResults = [_query results];
    for (NSMetadataItem *result in queryResults) {
       NSURL *fileURL = [result valueForAttribute:NSMetadataItemURLKey];
        NSNumber *aBool = nil;

        // 隠しファイルを除外する
        [fileURL getResourceValue:&aBool forKey:NSURLIsHiddenKey
error:nil];
        if (aBool && ![aBool boolValue])
            [discoveredFiles addObject:fileURL];
    }
```

```
    // ドキュメントのリストを更新する
    [documents removeAllObjects];
    [documents addObjectsFromArray:discoveredFiles];
    [self.tableView reloadData];

    // クエリの更新を有効に戻す
    [_query enableUpdates];
}
```

アプリケーションのビルドと実行

この時点で、完全なアプリケーションのビルドと実行に必要なすべての作業が完了しました。iCloudで、新しいドキュメントを作成し、既存ドキュメントを開き、それらのドキュメントの中身を編集することができるようになりました。プロビジョニングプロファイルに関連付けられたデバイスが複数ある場合は、各デバイスを接続して、ほかのデバイスで作成したファイルを表示できるようになったはずです。

アプリケーションが期待どおりに動作しない場合は、“トラブルシューティング”（63 ページ）内の情報を使用して問題を解決できます。

まとめ

この時点で、検索機能の実装は完了し、3番目のiOSアプリケーションは完成しました。おめでとうございます。

少し時間を取って、iCloudがアプリケーション全体のアーキテクチャにどのような影響を与えたかを考えてみてください。iCloudに対応するための処理は、多くがアプリケーションのデータ部分に対するものです。`UIDocument`オブジェクトを使用してファイルを管理すれば、実行する必要がある作業は実際には少なくて済みます。

トラブルシューティング

アプリケーションが正しく機能しないトラブルが発生した場合は、この章に記載された問題解決のアプローチを試します。

プロビジョニングプロファイルの問題の診断

コードのコンパイルはできるのに、ビルドをデバイスにインストールする作業に失敗する場合は、アプリケーションのプロビジョニング情報を調べる必要があります。

- アプリケーションをデバイスにインストールできない場合は、「コード署名ID (Code Signing Identity)」ビルド設定の値を調べて、正しいプロビジョニングプロファイルに設定されていることを確認します。
- プロビジョニングプロファイルが無効として表示されている場合は、正しい開発証明書が現在のマシンにインストールされていることを確認します。キーチェーンアクセスアプリケーションを使用すると、利用可能な証明書を確認できます。
- 「コード署名ID (Code Signing Identity)」ポップアップメニューでプロビジョニングプロファイルが無効として表示されている場合は、プロジェクトのバンドル識別子がプロビジョニングプロファイル内のバンドル識別子と一致していることを確認します。インストールされている証明書が、プロビジョニングプロファイル用に選択された証明書と一致しない場合にも、プロファイルは無効と表示されます。

iCloud使用可能性の問題の診断

すべてが適切に設定されていれば、アプリケーションがiCloudコンテナディレクトリを取得するために`URLForUbiquityContainerIdentifier:`メソッドを呼び出したときに、そのメソッドは、有効なURLを返すはずです。メソッドが`nil`を返した場合は、次の原因のいずれかによる可能性があります。

- デバイスがiCloudアカウントで設定されていないか、「ドキュメントとデータ(Documents & Data)」オプションが無効になっています。
- アプリケーションのエンタイトルメントが正しく設定されていません。

開発中に発生する可能性がもっとも大きな問題は、アプリケーションのエンタイトルメントが正しく設定されていないことです。“入門”（8 ページ）に記載された手順をダブルチェックして、エンタイトルメントが正しく設定されていること、およびアプリケーションIDとプロビジョニングプロファイルが最新の状態で、Xcodeプロジェクトが使用中であることを確認します。

コードおよびコンパイラの警告

期待どおりに機能しない部分がある場合は、まず、自分のコードと“コードリスト”（70 ページ）に示す完全なリストを比較します。

コードは、どのような警告も表示されることなく、コンパイルされなければなりません。警告が表示された場合は、エラーと同等に扱うことをお勧めします。Objective-C言語は非常に柔軟な言語なので、コンパイラから出力されるメッセージがせいぜい警告どまりだからです。

あるメソッドがほかのメソッドを呼び出すときは、呼び出される側のメソッドを呼び出す側のメソッドより前に定義しなければなりません。クラスが所定のセレクタ名でメソッドを宣言していないというエラーをコンパイラが報告した場合は、この問題を解決するために、ソースファイル内でのメソッドの位置を再調整してみてください。別の指示がない限り、このチュートリアルのメソッドをクラス宣言に含める必要はありません。

ストーリーボードファイルのチェック

デベロッパとして、期待どおりに機能しない部分がある場合は、ソースコードにバグがないかどうかを調べたくなるのが本能です。しかし、Cocoa Touchを使用しているときは、別の面が加わります。アプリケーションの設定の多くの部分が、ストーリーボードファイル内に「エンコード」されている場合があるのです。正しく接続していなかった場合、アプリケーションは期待どおりの動作をしません。

- 明示的にデータを追加してもテーブルにデータが何も表示されない場合は、Master View ControllerをTable Viewのデリゲートとして設定するのを忘れている可能性があります。
- セルをタップしてもDetail Viewが表示されない場合は、セグエの設定を確認します。テーブルのコンテンツタイプが変化すると、プロジェクト内の既存のセグエは削除されるため、作成し直さなければなりません。
- 「Add」ボタンをタップしても希望どおりの結果にならない場合は、アクションメソッドを接続し忘れている可能性があります。

通知メソッドの名前

通知に関して犯しやすいミスは、通知ハンドラのメソッド名のスペルを間違えることです。通知オブザーバの登録時に、実際のメソッド名とは異なる名前を付けてしまうと、正しいメソッドが呼び出されません。ハンドラを指定するときは、通常、メソッド名をコピーアンドペーストする方が無難です。

次のステップ

このチュートリアルでは、iCloudを使用してドキュメントを保存する高度なiOSアプリケーションを作成しました。iCloudをサポートするアプリケーションを設計する際には、さまざまな判断をすることになりますが、このチュートリアルではほんの表面をなぞったに過ぎません。この章では、アプリケーションへのiCloudの統合に関する学習を続けるために、次に進むべき方向をいくつか提案します。

ドキュメントの版の食い違い解消

ドキュメントをiCloudに格納するアプリケーションは、そのドキュメントの複数のバージョン間の食い違いを処理できる機能を備えていなければなりません。食い違いが発生する恐れがあるのは、2台の異なるデバイスで同一ドキュメントに変更を加えたときです。たとえば、機内モードになっている2台の異なるデバイスでユーザが同一ドキュメントを編集したときは、iCloudサーバに変更内容を転送することができないため、食い違いが発生する恐れがあります。

食い違いはそれほど頻繁に発生するわけではありませんが、それを処理する機能をアプリケーションに用意する必要があります。ドキュメントベースのアプリケーションで食い違いを検出するためには、`UIDocument`クラスの状態変化通知を受け取るように登録しなければなりません。ドキュメントが`UIDocumentStateInConflict`状態になったとき、アプリケーションは、食い違っているドキュメントの版を取得して、最善の処理方法を判断する必要があります。

版の食い違い解消の詳細については、『*iOS App Programming Guide*』の"iCloud Storage"を参照してください。

アップロードとダウンロードの進行状況のユーザへの表示

ドキュメントが大きくなる場合は、iCloudとの間で送受信する際に、ユーザに何らかのフィードバックを表示することをお勧めします。`NSURL`クラスのインスタンスは、基盤のファイルの現在の転送状態を通知する属性を維持します。これらの値を使用すると、ファイルがローカルデバイスにダウンロードされたかどうか、および変更内容がiCloudにアップロードされたかどうかを判断できます。また、これらの値を使用して、ダウンロードとアップロード処理の現在の進行状況を調べることできます。

iCloudステータス属性へのアクセスの詳細については、*NSURL Class Reference*を参照してください。

iCloudが使用不能の場合の処理

起動の直後に`URLForUbiquityContainerIdentifier:`メソッドを呼び出した場合の利点の1つは、アプリケーションの実行期間の早い段階でiCloudが使用可能かどうかを判断できることです。メソッドから`nil`の値が返された場合は、そのiCloudコンテナディレクトリに到達できなかったということです。これは、通常、ユーザのデバイスがiCloudを使用するように設定されていないせいで発生します（開発中、iCloudへのアクセスが不可能になった場合は、通常、アプリケーションのiCloudエンタイトルメントの設定にミスがあるということです）。

iCloudが使用不能になっている場合、アプリケーションはスムーズなフォールバック位置を提供しなければなりません。たとえば、新しいユーザドキュメントをローカルなサンドボックスに格納し、最初に訪れた機会にそれらをiCloudに転送する場合があります。これは、ユーザを悩ませずに、内部のみで処理しなければなりません。

また、「ドキュメントとデータ(Documents & Data)」オプションをオフにするか、現在のiCloudアカウントを削除すると、アプリケーションからiCloudへのアクセスをユーザが完全に遮断できる点にも注意する必要があります。これは頻繁に発生するわけではありませんが、アプリケーションが一時停止しているかバックグラウンドで実行されている間にこの種の変更が実行されないように、アプリケーションのコーディングで対策を講じなければなりません。特に、バックグラウンドに移行した後でフォアグラウンドに戻った後は、アプリケーションのコンテナディレクトリ内のファイルやドキュメントへのアクセスを試みる前に、`URLForUbiquityContainerIdentifier:`メソッドを呼び出して、iCloudがまだ使用可能かどうかを確認してください。

ドキュメント表示インターフェイスの改善

Simple Text Editorアプリケーションは、ドキュメントを検索して、発見した順に表示します。ただし、アプリケーションにドキュメントを表示するもっと確実な方法があれば好都合です。たとえば、アルファベット順に表示したり、現在の順序を記録してその情報もiCloudに書き込めたりすると便利です。

ドキュメントの動的命名機能のサポート

このチュートリアルでは、静的な名前と動的な数値を組み合わせたものを新規ドキュメントの名前として使用しましたが、もっと柔軟に命名できれば便利です。次に示すのは、優れたドキュメント名を作成するためのヒントです。

- アプリケーションが作成する内容を連想できる初期名を使用します。単なる「無題」のドキュメントを作成してはなりません。もっと具体的なドキュメント名にします。たとえば、ペイントプログラムなら、「マイクリエイション」や「キャンバス」を基本ドキュメント名にします。

- ドキュメント名を変更する簡単で控えめな手段をユーザに提供します。ユーザがドキュメント名をタップして編集できるようにします。警報を発したり、現在のコンテキストからユーザを引きずり出すようなインターフェイスを使用したりしてはなりません。
- コンテンツがテキストのドキュメントでは、（ファイル名ではなく）そのコンテンツを使用してドキュメントを識別することを検討します。このアプローチは、「メモ(Notes)」アプリケーションが情報を表示する方法と同じです。ユーザは基盤になるファイルシステムに直接アクセスするわけではないので、たいていの場合は、ファイル名を表示するより、ドキュメントの最初の方の内容を表示する方が便利です。

キー値の保存機能のサポート

アプリケーションで環境設定などのあまり重要ではない設定データを共有したい場合は、`NSUbiquitousKeyValueStore`クラスを使用して、iCloudによって実現します。このクラスは、`NSUserDefaults`クラスと同様に振る舞います。キー値のペアデータをiCloudで設定するための単純なインターフェイスを提供します。

`NSUbiquitousKeyValueStore`クラスの使い方については、『*Preferences and Settings Programming Guide*』を参照してください。

iCloudでのCore Dataの使用

Core Dataは、アプリケーションのデータ構造体をモデル化し、それらをアプリケーションで効率的に管理できる、一連の高度なツールを提供します。ライブSQLiteデータベースとは異なり、Core Dataストアは、iCloudを通じてユーザのデバイス間で共有できます。Core Dataは、変更内容だけをiCloudに送信することでこの処理を管理するので、各デバイスのローカルデータベースに組み込むことができます。

Core DataとiCloudの併用の詳細については、『*iOS App Programming Guide*』の"iCloud Storage"を参照してください。

ファイルコーディネータに関する知識の強化

ある時点で、iCloudでファイルコーディネータが果たす役割について、知識を強化したくなるかもしれません。`UIDocument`クラスは多数のアクションを対象にしたファイルコーディネータを備えていますが、一部のアクションではファイルコーディネータを自分で作成する必要があります。たとえ

ば、Simple Text Editorでファイルを削除するときは、専用のファイルコーディネータを作成し、それを使用して操作を実行する必要があります。したがって、ファイルコーディネータが自分のアプリケーションで果たす役割を理解し、それを使用すべきときを判断することが重要です。

ファイルコーディネータの使い方については、『*File System Programming Guide*』を参照してください。

コードリスト

この付録には、STEAppDelegate、STEMasterViewController、STEDetailViewController、STESimpleTextDocumentの各クラスのインターフェイスと実装のリストを掲載してあります。ただし、ファイルテンプレートからの変更がないメソッド実装は省いてあります。

STEAppDelegate.h

```
#import <UIKit/UIKit.h>

@interface STEAppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@end
```

STEAppDelegate.m

```
#import "STEAppDelegate.h"

@implementation STEAppDelegate
@synthesize window = _window;

- (void)initializeiCloudAccess {
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
 ^{
       if ([[NSFileManager defaultManager] URLForUbiquityContainerIdentifier:nil])
            NSLog(@"iCloud is available.\n");
        else
            NSLog(@"iCloud is not available.\n");
    });
}
```

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [self initializeiCloudAccess];
    // アプリケーション起動後のカスタマイズを記述する場所。
    return YES;
}
@end
```

STEMasterViewController.h

```
#import <UIKit/UIKit.h>

@interface STEMasterViewController : UITableViewController
@property (weak, nonatomic) IBOutlet UIBarButtonItem *addButton;
@end
```

STEMasterViewController.m

```
#import "STEMasterViewController.h"
#import "STESimpleTextDocument.h"
#import "STEDetailViewController.h"

NSString *STEDocFilenameExtension = @"stedoc";
NSString *DisplayDetailSegue = @"DisplayDetailSegue";
NSString *STEDocumentsDirectoryName = @"Documents";
NSString *DocumentEntryCell = @"DocumentEntryCell";


@implementation STEMasterViewController {
    NSMutableArray *documents;
    NSMetadataQuery *_query;
```

```
}
@synthesize addButton;

- (NSMetadataQuery*)textDocumentQuery {
    NSMetadataQuery* aQuery = [[NSMetadataQuery alloc] init];
    if (aQuery) {
        // Documentsサブディレクトリのみを検索する
        [aQuery setSearchScopes:[NSArray
                    arrayWithObject:NSMetadataQueryUbiquitousDocumentsScope]];

        // ドキュメントを検索するための述語を追加する
        NSString* filePattern = [NSString stringWithFormat:@"*.%@",
                    STEDocFilenameExtension];
        [aQuery setPredicate:[NSPredicate predicateWithFormat:@"%K LIKE %@",
                    NSMetadataItemFSNameKey, filePattern]];
    }

    return aQuery;
}

- (void)setupAndStartQuery {
    // まだ存在しない場合は、クエリオブジェクトを作成する
    if (!_query)
        _query = [self textDocumentQuery];

    // メタデータクエリ通知を受け取るように登録する
    [[NSNotificationCenter defaultCenter] addObserver:self
                    selector:@selector(processFiles:)
                    name:NSMetadataQueryDidFinishGatheringNotification
                    object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
                    selector:@selector(processFiles:)
                    name:NSMetadataQueryDidUpdateNotification
                    object:nil];
```

```
    [_query startQuery];
}

- (void)awakeFromNib
{
    [super awakeFromNib];

    if (!documents)
        documents = [[NSMutableArray alloc] init];

    self.navigationItem.leftBarButtonItem = self.editButtonItem;
    [self setupAndStartQuery];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // キャッシュ内のデータやイメージなどのうち、未使用のものを解放する
}

#pragma mark Document Management

- (NSString*)newUntitledDocumentName {
    NSInteger docCount = 1;   // 最初は1で、そこから増やしていく
    NSString *newDocName = nil;

    // この時点で、ドキュメントリストは最新の状態でなければならない
    BOOL done = NO;
    while (!done) {
        newDocName = [NSString stringWithFormat:@"Note %d.%@",
                        docCount, STEDocFilenameExtension];

        // 同一名の既存ドキュメントの有無を調べる同一名の既存ドキュメントが見つかった場合は、
        // docCount値を増やして、同じ処理を繰り返す
```

```
        BOOL nameExists = NO;
        for (NSURL* aURL in documents) {
            if ([[aURL lastPathComponent] isEqualToString:newDocName]) {
                docCount++;
                nameExists = YES;
                break;
            }
        }

        // 名前が見つからなかった場合は、ループを終了する
        if (!nameExists)
            done = YES;
    }
    return newDocName;
}

- (IBAction)addDocument:(id)sender {
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
 ^{
        // バックグラウンドキュー上に新たなURLオブジェクトを作成する
        NSFileManager *fm = [NSFileManager defaultManager];
        NSURL *newDocumentURL = [fm URLForUbiquityContainerIdentifier:nil];

        newDocumentURL = [newDocumentURL
            URLByAppendingPathComponent:STEDocumentsDirectoryName
            isDirectory:YES];
        newDocumentURL = [newDocumentURL
            URLByAppendingPathComponent:[self newUntitledDocumentName]];

        // メインキュー上で残りのタスクを実行する
        dispatch_async(dispatch_get_main_queue(), ^{
            // データ構造体とテーブルを更新する
            [documents addObject:newDocumentURL];

            // テーブルを更新する
```

```
            NSIndexPath* newCellIndexPath =
            [NSIndexPath indexPathForRow:([documents count] - 1) inSection:0];
            [self.tableView insertRowsAtIndexPaths:[NSArray
arrayWithObject:newCellIndexPath]
                                withRowAnimation:UITableViewRowAnimationAutomatic];

            [self.tableView selectRowAtIndexPath:newCellIndexPath
                                        animated:YES
                                        scrollPosition:UITableViewScrollPositionMiddle];

            // Detail View Controllerへのセグエの編集を開始する
            UITableViewCell* selectedCell = [self.tableView
                                    cellForRowAtIndexPath:newCellIndexPath];
          [self performSegueWithIdentifier:DisplayDetailSegue sender:selectedCell];
        });
    });
}

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if (![segue.identifier isEqualToString:DisplayDetailSegue])
        return;

    // Detail View Controllerを取得する
    STEDetailViewController* destVC =
           (STEDetailViewController*)segue.destinationViewController;

    // ドキュメント配列から正しいディクショナリを検索する
    NSIndexPath *cellPath = [self.tableView indexPathForSelectedRow];
    UITableViewCell *theCell = [self.tableView cellForRowAtIndexPath:cellPath];
    NSURL *theURL = [documents objectAtIndex:[cellPath row]];

    // URLをDetail View Controllerに割り当て、
    // View Controllerのタイトルをドキュメント名に設定する
    destVC.detailItem = theURL;
    destVC.navigationItem.title = theCell.textLabel.text;
```

```
}



#pragma mark Table View methods


- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section {
    return [documents count];
}


- (UITableViewCell*)tableView:(UITableView *)tableView
                    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *newCell = [tableView
dequeueReusableCellWithIdentifier:DocumentEntryCell];
    if (!newCell)
       newCell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                                            reuseIdentifier:DocumentEntryCell];


    if (!newCell)
        return nil;


    // 指定された行にあるドキュメントを取得する
    NSURL *fileURL = [documents objectAtIndex:[indexPath row]];


    // セルを設定する
    newCell.textLabel.text = [[fileURL lastPathComponent]
stringByDeletingPathExtension];
    return newCell;
}


 - (void)tableView:(UITableView *)tableView
         commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
         forRowAtIndexPath:(NSIndexPath *)indexPath {
     if (editingStyle == UITableViewCellEditingStyleDelete) {
         NSURL *fileURL = [documents objectAtIndex:[indexPath row]];
```

```
        // アプリケーションのメインキューでファイルコーディネータを使用してはならない
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
0), ^{
            NSFileCoordinator *fc = [[NSFileCoordinator alloc]
                                     initWithFilePresenter:nil];
            [fc coordinateWritingItemAtURL:fileURL
                options:NSFileCoordinatorWritingForDeleting
                error:nil
                byAccessor:^(NSURL *newURL) {
                    NSFileManager *fm = [[NSFileManager alloc] init];
                    [fm removeItemAtURL:newURL error:nil];
            }];

        });

        // ドキュメントの配列からURLを削除する
        [documents removeObjectAtIndex:[indexPath row]];

        // テーブルUIを更新する。これは、
        // ドキュメント配列の更新後でなければならない
        [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                    withRowAnimation:UITableViewRowAnimationAutomatic];
    }
}


#pragma mark Query Method

- (void)processFiles:(NSNotification*)aNotification {
    NSMutableArray *discoveredFiles = [NSMutableArray array];

    // 結果の処理時には更新を必ず無効にする
    [_query disableUpdates];
```

```
    // クエリは、常に、発見したすべてのファイルを報告する
    NSArray *queryResults = [_query results];
    for (NSMetadataItem *result in queryResults) {
        NSURL *fileURL = [result valueForAttribute:NSMetadataItemURLKey];
        NSNumber *aBool = nil;

        // 隠しファイルを除外する
        [fileURL getResourceValue:&aBool forKey:NSURLIsHiddenKey error:nil];
        if (aBool && ![aBool boolValue])
            [discoveredFiles addObject:fileURL];
    }

    // ドキュメントのリストを更新する
    [documents removeAllObjects];
    [documents addObjectsFromArray:discoveredFiles];
    [self.tableView reloadData];

    // クエリの更新を有効に戻す
    [_query enableUpdates];
}
```

STEDetailViewController.h

```
#import <UIKit/UIKit.h>
#import "STESimpleTextDocument.h"

@interface STEDetailViewController : UIViewController <STESimpleTextDocumentDelegate>

@property (strong, nonatomic) NSURL *detailItem;
@property (strong, nonatomic) IBOutlet UILabel *detailDescriptionLabel;
@property (weak, nonatomic) IBOutlet UITextView *textView;

@end
```

STEDetailViewController.m

```
#import "STEDetailViewController.h"

@interface STEDetailViewController ()
- (void)configureView;
@end

@implementation STEDetailViewController {
    STESimpleTextDocument *_document;
}
@synthesize detailItem = _detailItem;
@synthesize detailDescriptionLabel = _detailDescriptionLabel;
@synthesize textView = _textView;

#pragma mark View Management

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

    // Text Viewの内容を削除する
    self.textView.text = @"";

    // ドキュメントを作成して、デリゲートを割り当てる
    _document = [[STESimpleTextDocument alloc] initWithFileURL:self.detailItem];
    _document.delegate = self;

    // ファイルが存在する場合はそれを開き、それ以外の場合は作成する
    NSFileManager *fm = [NSFileManager defaultManager];
    if ([fm fileExistsAtPath:[self.detailItem path]])
        [_document openWithCompletionHandler:nil];
    else
        // 新しいドキュメントをディスクに保存する
        [_document saveToURL:self.detailItem
                forSaveOperation:UIDocumentSaveForCreating
                completionHandler:nil];
```

```
    // 通知の受け取りを登録する
    [[NSNotificationCenter defaultCenter] addObserver:self
                        selector:@selector(keyboardWillShow:)
                        name:UIKeyboardWillShowNotification
                        object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
                        selector:@selector(keyboardWillHide:)
                        name:UIKeyboardWillHideNotification
                        object:nil];
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];

    NSString *newText = self.textView.text;
    _document.documentText = newText;

    // ドキュメントを閉じる
    [_document closeWithCompletionHandler:nil];

    [[NSNotificationCenter defaultCenter] removeObserver:self
                        name:UIKeyboardWillShowNotification
                        object:nil];
    [[NSNotificationCenter defaultCenter] removeObserver:self
                        name:UIKeyboardWillHideNotification
                        object:nil];
}


#pragma mark Keyboard Handlers

- (void)keyboardWillShow:(NSNotification*)aNotification {
    NSDictionary *info = [aNotification userInfo];
```

```
    CGRect kbSize = [[info objectForKey:UIKeyboardFrameEndUserInfoKey]
                            CGRectValue];
    double duration = [[info objectForKey:UIKeyboardAnimationDurationUserInfoKey]
                            doubleValue];

    UIEdgeInsets insets = self.textView.contentInset;
    insets.bottom += kbSize.size.height;

    [UIView animateWithDuration:duration animations:^{
        self.textView.contentInset = insets;
    }];
}

- (void)keyboardWillHide:(NSNotification*)aNotification {
    NSDictionary *info = [aNotification userInfo];
    double duration = [[info objectForKey:UIKeyboardAnimationDurationUserInfoKey]
                            doubleValue];

    // Text Viewの下部コンテンツ挿入枠をリセットする
    UIEdgeInsets insets = self.textView.contentInset;
    insets.bottom = 0;

    [UIView animateWithDuration:duration animations:^{
        self.textView.contentInset = insets;
    }];
}
```

STESimpleTextDocument.h

```
#import <UIKit/UIKit.h>

@protocol STESimpleTextDocumentDelegate;

@interface STESimpleTextDocument : UIDocument
```

```
@property (copy, nonatomic) NSString* documentText;
@property (weak, nonatomic) id<STESimpleTextDocumentDelegate> delegate;
@end

@protocol STESimpleTextDocumentDelegate <NSObject>
@optional
- (void)documentContentsDidChange:(STESimpleTextDocument*)document;
@end
```

STESimpleTextDocument.m

```
#import "STESimpleTextDocument.h"

@implementation STESimpleTextDocument
@synthesize documentText = _documentText;
@synthesize delegate = _delegate;

- (void)setDocumentText:(NSString *)newText {
    NSString* oldText = _documentText;
    _documentText = [newText copy];

    // 取り消し操作を登録する
    [self.undoManager setActionName:@"Text Change"];
    [self.undoManager registerUndoWithTarget:self
            selector:@selector(setDocumentText:)
            object:oldText];
}

- (id)contentsForType:(NSString *)typeName
      error:(NSError *__autoreleasing *)outError {
    if (!self.documentText)
        self.documentText = @"";
```

```
    NSData *docData = [self.documentText dataUsingEncoding:NSUTF8StringEncoding];

    return docData;
}

- (BOOL)loadFromContents:(id)contents
        ofType:(NSString *)typeName
        error:(NSError *__autoreleasing *)outError {
    if ([contents length] > 0)
        self.documentText = [[NSString alloc]
                initWithData:contents
                encoding:NSUTF8StringEncoding];
    else
        self.documentText = @"";

    // ドキュメントの中身の変化をデリゲートに伝える
    if (self.delegate && [self.delegate respondsToSelector:
                          @selector(documentContentsDidChange:)])
        [self.delegate documentContentsDidChange:self];

    return YES;
}

@end
```

書類の改訂履歴

この表は「*3つ目のiOSアプリケーション：iCloud*」の改訂履歴です。

日付	メモ
2012-01-09	iCloudをiOSアプリケーションに組み込む方法を示す新規ドキュメント

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3 丁目20 番2 号
東京オペラシティタワー
http://www.apple.com/jp/