

iOS Swift - Network 400



iOS Swift - Network 400



Copyright © Quaddro Treinamentos Empresariais LTDA

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Quaddro Treinamentos Empresariais Ltda, estado o contrafator sujeito a responder por crime de Violação do Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos o logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

Autoria:

Tiago Santos de Souza
Roberto Rodrigues Junior

Revisão:

Tiago Santos de Souza

Design, edição e produção:

Tiago Santos de Souza

“As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais.”

Edição nº3
Janeiro 2018

Sumário



Apresentação.....3

Capítulo 1 - Webservices.....5

Seção 1-1 - Introdução.....7

Seção 1-3 - URLSession.....8

Capítulo 2 - Trabalhando com SQLite.....13

Seção 2-1 - Banco de dados.....15

Seção 2-2 - Preparando o SQLite.....16

Seção 2-3 - Comandos de SQLite.....18

Capítulo 3 - Core Data.....21

Seção 3-1 - Introdução.....23

Seção 3-2 - Componentes do Core Data.....24

Seção 3-3 - Core Data em um novo projeto.....26

Seção 3-4 - Criando um modelo de objetos.....27

Seção 3-5 - Avançando no projeto.....29

Seção 3-6 - NSManagedObjectContext.....31

Seção 3-7 - NSFetchRequest.....32

Seção 3-8 - NSManagedObject.....33

Seção 3-9 - NSPredicate.....34

Capítulo 4 - CloudKit.....35

Seção 4-1 - Introdução.....37

Seção 4-2 - Preparando o esquema de dados....38

Seção 4-3 - CKContainer.....40

Seção 4-4 - CKRecord.....41

Seção 4-5 - CKDataBase.....42

Seção 4-6 - CKQuery.....43

Apendice 1 - Caderno de Exercícios.....45



Depois de conhecermos a sintaxe do **Swift**, aprendermos a utilizar principais recursos relacionados ao desenvolvimento da **interface gráfica** e trabalharmos com frameworks que complementam nossos aplicativos, chegamos ao ponto onde vamos trabalhar com recursos mais avançados no desenvolvimento com iOS.

Nesse módulo serão abordados assuntos como consumo de **Web Services** e **banco de dados** utilizando **Core Data**.

Iremos aprender a criar aplicativos que possam se comunicar em rede com as principais classes Cocoa, além de adquirir conhecimentos para consumo de recursos de Web Services.

Outro ponto positivo a destacar nesse módulo, é a utilização do **CloudKit**.

CloudKit é o serviço de armazenamento de dados remoto da Apple baseado no **iCloud**. Com ele temos uma opção de baixo custo para armazenar e compartilhar dados de aplicativos usando as contas iCloud dos usuários como um serviço de armazenamento **back-end**.

Nesse módulo continuaremos a trabalhar tanto com recursos nativos dos framework Foundation, bem como a implementação de outras estruturas nos nossos projetos.

Nossa base de desenvolvimento continuará sendo o **Xcode**. Devemos nos atentar apenas para a **importação** correta dos frameworks a serem utilizados, quando forem necessários.

Bons estudos!



Consumo de Webservices



Dependendo do tipo de aplicativo que se desenvolva, acessar apenas dados locais não é suficiente. Nesse caso é necessário uma conexão com a internet para carregamento ou mesmo envio de dados a sistemas legados em servidores.

Podemos definir um WebService como uma aplicação para integrar sistemas. A premissa é servir de ponte para ligar dois ou mais sistemas que estruturalmente não estão conectados diretamente.

Um webservice tem duas características principais:

1. Ele deve servir de provedor de dados;

2. Ele deve ser acessado via protocolo HTTP.

Essas duas características básicas definem o escopo básico de uma aplicação chamada webservice. Ao longo do anos, alguns tipos de webservices foram criados, mas atualmente podemos simplificar em dois tipos mais usados: **SOAP** e **RESTful**.

RESTful

Um webservice **RESTful** (Representative State Transfer) é uma alternativa mais simplificada e enxuta do que o SOAP por permitir o trabalho com **JSON** (Javascript Object Notation).

Seu uso tem se consolidado nos últimos anos devido a sua estrutura definida inteiramente sobre o protocolo **HTTP**.

Desta forma, ele tende a ser mais alinhado com as características de webservices do que o SOAP, que depende de componentes e frameworks para sua implementação.

Classe nativa de Foundation para webservices

Podemos utilizar ferramentas nativas **Foundation** para realizarmos requisições em web services.

Desta forma daremos funcionalidades muito mais amplas para nossas aplicações, que poderão gerir dados de maneira dinâmica enviando e recebendo dados de um servidor. O Foundation possui uma classe para realizar estas operações, a **URLSession**.

Adiante temos um tópico abordando exclusivamente essa classe, bem como suas propriedades e métodos.



Herança: NSObject > URLSession

A classe **URLSession** foi construída como uma alternativa mais completa com comandos de acesso a web services.

Com a classe **URLSession**, podemos interagir com os protocolos **HTTP** e **HTTPS**, de forma altamente configurável, podendo proporcionar muitas possibilidades, como navegação privada por exemplo.

O conceito básico para entendimento da classe **URLSession** são as **tarefas (tasks)**, instâncias da classe **URLSessionTask**. Existem três tipos de tarefas: as tarefas de **dados**, tarefas de **upload** e tarefas de **download**:

- **Classe URLSessionDataTask**: Utilizada para realizar uma tarefa que retorna diretamente para a aplicação dados, como um JSON, por exemplo:

- **Classe URLSessionUploadTask**: Utilizada para realizar o envio de dados para o servidor. A classe **URLSessionUploadTask** é subclasse **URLSessionDataTask**;

- **Classe URLSessionDownloadTask**: Utilizada para realizar o download dos dados da requisição. A grande diferença em relação a **URLSessionDataTask** está no fato de que a **URLSessionDownloadTask** grava os dados no sistema de arquivos, e não na memória, possibilitando a pausa e a continuação de seu recebimento.

#Dica!

Em conjunto com a classe **URLSession**, podemos utilizar objetos da classe **URL** para indicarmos os caminhos dos nossos web services. Outra classe importante que também pode ser explorada durante o processo é a **JSONSerialization**. Com ela podemos serializar informações em formato **JSON** tanto para trazê-las para nossa aplicação, quanto para enviá-las para web services.

Principais propriedades

Propriedade	Tipo	Descrição
configuration	URLSession-Configuration	Retorna uma cópia da configuração da sessão
delegate	URLSession-Delegate?	Retorna o delegate
delegateQueue	OperationQueue	Retorna quando o objeto delegate é criado
session-Description	String?	Retorna a label descritiva para a sessão
shared	URLSession	Retorna um objeto singleton compartilhado

Principais métodos para requisições

```
//Método que inicia uma sessão com uma configuração  
especificada:  
init(configuration: URLSessionConfiguration)
```

```
//Método que faz uma requisição do tipo GET para uma URL  
especificada e retorna em um callback o resultado:  
func dataTask(with url: URL, completionHandler: @escaping  
(Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask
```

```
//Método que faz uma requisição com a URL de requisição  
fornecida e retorna o resultado através de um callback:  
func dataTask(with request: URLRequest, completionHandler:  
@escaping (Data?, URLResponse?, Error?) -> Void)  
-> URLSessionDataTask
```

Principais métodos para download

```
//Método que cria uma tarefa de download a partir de uma  
URL especificada, salva os resultados em um arquivo, e retorna  
os resultados em um callback:  
func downloadTask(with url: URL, completionHandler:  
@escaping (URL?, URLResponse?, Error?) -> Void)  
-> URLSessionDownloadTask
```

```
//Método que cria uma tarefa de download a partir de uma  
requisição URL especificada, salva os resultados em um  
arquivo, e retorna os resultados em um callback:  
func downloadTask(with request: URLRequest, completion-  
Handler: @escaping (URL?, URLResponse?, Error?) -> Void)  
-> URLSessionDownloadTask
```

```
//Método que cria uma tarefa de download a partir de uma  
para continuar um download cancelado ou que falhou e retorna  
os resultados em um callback:  
func downloadTask(withResumeData resumeData: Data,  
completionHandler: @escaping (URL?, URLResponse?, Error?)  
-> Void) -> URLSessionDownloadTask
```

Principais métodos para Upload

```
//Método que cria uma requisição HTTP para uma URL especificada,  
fazendo o upload do objeto fornecido e retornando o resultado  
em um callback:  
func uploadTask(with request: URLRequest, from bodyData:  
Data?, completionHandler: @escaping (Data?, URLResponse?,  
Error?) -> Void) -> URLSessionUploadTask
```

```
//Método que cria uma requisição HTTP para uma URL especificada,  
fazendo o upload de uma URL fornecida e retornando o resultado  
em um callback:  
func uploadTask(with request: URLRequest, fromFile fileURL:  
URL, completionHandler: @escaping (Data?, URLResponse?,  
Error?) -> Void) -> URLSessionUploadTask
```

Principais métodos para o gerenciamento da sessão

```
//Método que invalida o objeto de sessão, permitindo que  
as tarefas pendentes sejam terminadas:  
func finishTasksAndInvalidate()
```

```
//De forma assíncrona, chama um callback que retorna todos  
os dados, uploads e downloads, pendentes na sessão:  
func getTasksWithCompletionHandler(_ completionHandler:  
@escaping ([URLSessionDataTask], [URLSessionUploadTask],  
[URLSessionDownloadTask]) -> Void)
```

```
//Método que cancela todas as tarefas pendentes e invalida  
o objeto:  
func invalidateAndCancel()
```

```
//Método que reseta a sessão de forma assíncrona:  
func reset(completionHandler: @escaping () -> Void)
```

Protocolos Delegate

Existem protocolos para captar ações de session, task, data e download. Os protocolos devem ser adotados pelas classes conforme a necessidade de utilização.

```
extension ClasseViewController : ProtocoloDelegate{  
    // Declaração dos métodos  
}
```

Principais métodos URLSessionDelegate

```
//Método que comunica ao delegado que uma sessão se tornou  
inválida:  
optional func urlSession(_ session: URLSession,  
didBecomeInvalidWithError error: Error?)
```

```
//Solicita credenciais do delegado em resposta a uma  
solicitação de autenticação de nível de sessão do servidor  
remoto:  
optional func urlSession(_ session: URLSession, didReceive  
challenge: URLAuthenticationChallenge, completionHandler:  
@escaping (URLSession.AuthChallengeDisposition,  
URLCredential?) -> Void)
```

```
//Método que comunica ao delegado quando todas as mensagens  
enfileiradas foram entregues:  
optional func urlSessionDidFinishEvents  
(forBackgroundURLSession session: URLSession)
```

Principais métodos URLSessionTaskDelegate

```
//Método que retorna ao delegate quando uma tarefa terminou  
a transferência de dados:  
optional func urlSession(_ session: URLSession, task:  
URLSessionTask, didCompleteWithError error: Error?)
```

```
//Método que informa periodicamente ao delegate sobre o  
progresso do envio do conteúdo ao servidor:  
optional func urlSession(_ session: URLSession, task:  
URLSessionTask, didSendBodyData bytesSent: Int64,  
totalBytesSent: Int64, totalBytesExpectedToSend: Int64)
```

```
//Método que comunica ao delegate que o servidor remoto
solicitou um redirecionamento HTTP:
optional func urlSession(_ session: URLSession, task:
URLSessionTask, willPerformHTTPRedirection response:
HTTPURLResponse, newRequest request: URLRequest,
completionHandler: @escaping (URLRequest?) -> Void)
```

Principais métodos URLSessionDataDelegate

```
//Método que diz ao delegate que uma tarefa recebeu uma
resposta inicial (headers) do servidor:
optional func urlSession(_ session: URLSession, dataTask:
URLSessionDataTask, didReceive response: URLResponse,
completionHandler: @escaping (URLSession.
ResponseDisposition) -> Void)
```

```
//Método que diz ao delegate quando uma tarefa de dados
muda para uma tarefa de download:
optional func urlSession(_ session: URLSession,
dataTask: URLSessionDataTask, didBecome downloadTask:
URLSessionDownloadTask)
```

```
//Método que comunica ao delegate quando uma tarefa de dados
recebeu dados do servidor:
optional func urlSession(_ session: URLSession, dataTask:
URLSessionDataTask, didReceive data: Data)
```

```
//Método que pergunta ao delegate se uma tarefa de dados
(ou upload) deve armazenar a resposta em cache:
optional func urlSession(_ session: URLSession, dataTask:
URLSessionDataTask, willCacheResponse proposedResponse:
CachedURLResponse, completionHandler: @escaping
(CachedURLResponse?) -> Void)
```

Principais métodos URLSessionDownloadDelegate

```
//Método que comunica ao delegate que a tarefa de download
retomou o download:
optional func urlSession(_ session: URLSession, downloadTask:
URLSessionDownloadTask, didResumeAtOffset fileOffset: Int64,
expectedTotalBytes: Int64)
```

```
//Método que informa periodicamente ao delegate sobre o
progresso do download:
optional func urlSession(_ session: URLSession,
downloadTask: URLSessionDownloadTask, didWriteData
bytesWritten: Int64, totalBytesWritten: Int64,
totalBytesExpectedToWrite: Int64)
```




Trabalhando com SQLite

Seção 2-1

Banco de dados



Aplicativos que possuem um grande volume de informações requerem uma maneira otimizada para organizar esses dados. Uma abordagem que permite controle e performance sobre as manipulações dessa informações é o uso de **Bancos de Dados**.

Neste capítulo vamos aprender a trabalhar com o modelo de banco de dados local **SQLite**, que elimina a necessidade de uma conexão com um servidor para criar e manipular bases de dados.

Podemos definir um banco de dados como um sistema que armazena um conjunto de registros. Esses registros podem ser ordenados e manipulados de acordo com o tipo do banco de dados em questão.

Um ponto importante no trabalho com banco de dados é que sua estrutura é normalmente baseada no acesso centralizado de dados, ou seja, a arquitetura padrão para um banco de dados é **cliente-servidor**. Isso quer dizer que existe uma peça central que armazena os dados que podem ser acessados por diferentes clientes.

Essa arquitetura ganhou força com o crescimento da internet, onde existe um servidor central (hospedagem) que contém os arquivos que vários usuários podem carregar e acessar.

Porém, o fato de um banco de dados se limitar a conexão com um servidor pode ser um problema no caso de aplicações em dispositivos móveis, dado o fato de smartphones ou tablets não estarem constantemente conectados a Internet.

Nessa lacuna, ganhou força uma abordagem que permite tanto a criação, quanto a manipulação de dados diretamente no dispositivo. Essa forma de trabalho pode ser implementada pelo uso do **SQLite**.

#Dica!

É importante que um sistema de persistência de arquivos (**Sandbox**) seja desenvolvido para que possamos gravar e ou acessar as informações do nosso banco de dados. Esse arquivo pode ser gravado na pasta **Documents** da sua aplicação com uma extensão **.sqlite**.

Seção 2-2

Preparando o SQLite



Quando trabalhamos com **SQLite** temos que ter em mente que vamos utilizar um conjunto de comandos que foram desenvolvidos em uma linguagem diferente da Swift (linguagem C), ou seja, os comandos relacionados a manipulação do banco de dados SQLite.

Devemos fazer a importação da biblioteca de comandos, bem como a sua declaração nos nossos códigos para a partir daí acessarmos os recursos.

A partir da versão 4.0 da Swift, o SQLite está disponível como um framework, bastando fazer a sua importação no início dos arquivos que terão acesso aos comandos de banco de dados.

```
import SQLite3
```

A partir daí, já teremos acesso aos comandos de SQLite no nosso projeto.

Seção 2-3

Comandos de SQLite



SQLite é o nome de um tipo de banco de dados que pode ser criado e manipulado diretamente no dispositivo, sem a necessidade de um servidor remoto.

O recurso existe em algumas tecnologias Web e também foi incorporado ao iOS pela Apple para permitir o trabalho com dados locais.

Sua estrutura está baseada em **Ansi-C**, e o framework SQLite3 prove acesso a esses comandos após sua importação.

Tipos de dados suportados

Por ter como objetivo ser um banco leve e poderoso, o SQLite é implementado com poucos tipos de dados suportados.

A seguir você encontra uma tabela que lista esses tipos:

Tipos de dados suportados	
NULL	Valor Nulo
INTEGER	Números inteiros
REAL	Números decimais
TEXT	Textos(UTF-8, UTF-16)
BLOB	Dados binários (fotos, áudios, vídeos...)

#Dica!

Antes de criarmos os itens que serão inseridos no banco de dados, é importante que um sistema de sandbox esteja disponível no seu projeto para que aconteça a persistência dos dados, que utilize um arquivo de extensão **.sqlite**, ou algum outro banco de dados.

Principais métodos de SQLite

```
//Método que conecta um banco existente, ou executa a sua criação:  
func sqlite3_open(_ filename: UnsafePointer<Int8>!, _ ppDb: UnsafeMutablePointer<OpaquePointer?>!) -> Int32
```

```
//Método que fecha o banco de dados:  
func sqlite3_close(_: OpaquePointer!) -> Int32
```

```
//Método que executa uma instrução SQL sem retorno de dados  
func sqlite3_exec(banco : sqlite3, comando: String, callback, error)
```

```
//Método que executa uma instrução que retorna dados:  
func sqlite3_prepare_v2(_ db: OpaquePointer!, _ zSql: UnsafePointer<Int8>!, _ nByte: Int32, _ ppStmt: UnsafeMutablePointer<OpaquePointer?>!, _ pzTail: UnsafeMutablePointer<UnsafePointer<Int8>?>!) -> Int32
```

```
func sqlite3_errmsg(_: OpaquePointer!) ->
UnsafePointer<Int8>!
//Método é disparado para reportar um erro

func sqlite3_step(_: OpaquePointer!) -> Int32
//Método que executa uma instrução compilada, geralmente
utilizado dentro de um laço

func sqlite3_column_int(_: OpaquePointer!, _ iCol: Int32)
-> Int32
//Método que retorna uma determinada coluna a partir do seu
índice

func sqlite3_column_text(_: OpaquePointer!, _ iCol: Int32)
-> UnsafePointer<UInt8>!
//Método que retorna uma determinada coluna a partir do seu
nome

func sqlite3_finalize(_ pStmt: OpaquePointer!) -> Int32
//Método utilizado para excluir uma instrução preparada
```

Criando e adicionando itens à tabelas

Temos a disposição comandos para criar tabelas e inserir itens de dados em seu interior.

Como esses comandos são provenientes da linguagem C, eles devem ser declarados dentro de constantes (let) como se estivessem inscritos dentro de uma String, veja um exemplo:

```
let query = "insert into ALUNOS values (NULL, '\(nome)', \
(idade))"
"create table if not exists TABELA (id INTEGER NOT NULL
PRIMARY KEY AUTOINCREMENT, propriedade1 TEXT, propriedade2
INTEGER)"

//Cria uma nova tabela caso alguma com o mesmo nome não
exista, com a possibilidade de adição de propriedades
conforme as necessidades

"insert into TABELA values (propriedade1, propriedade2)"

//Insere dados na tabela de acordo com as propriedades
indicadas

"select * from TABELA"

//Lista itens da tabela

"update TABELA set propriedade1 = "Leonardo DaVinci" where
name = "Michelangelo";"

//Altera um registro em determinada propriedade

"delete from people where id = 2;"

//Deleta um registro em um determinado id
```




Core Data



Existem varias maneiras de trabalharmos com o armazenamento e exibição de informações em banco de dados. O **Core Data** é um framework que simplifica a forma de trabalharmos com dados e realizarmos a persistência dos mesmos.

Quando utilizamos nosso dispositivo, não nos damos conta de quantas informações estão armazenadas dentro dos aplicativos que utilizamos.

Essas informações não se limitam apenas a arquivos, como imagens e documentos, podendo ser também bancos de dados, arquivos JSON, XML, entre outras formas de guardarmos informações para podermos utilizá-las posteriormente, e são essas informações que permitem que você continue uma fase em um determinado jogo, ou que você possa fazer uma listagem de informações e salvá-las para continuar as edições depois.

O framework Core Data não significa tecnicamente uma base de dados, embora também seja utilizado para guardar suas informações em uma. Também não é um mapeador objeto relacional, apesar de possuir características semelhantes.

O Core Data gera um gráfico de objetos, permitindo que se crie, guarde, e resgate objetos que possuem seus atributos e seus relacionamentos com outros objetos. É uma maneira simples e poderosa de persistir desde informações básicas vindas de um simples formulário, até informações mais complexas, repletas de relacionamentos.

É importante salientar que o Core Data gera banco de dados No Cicle, orientado a objetos e que não depende de uma chave primária como um id por exemplo, para atuar sobre as tabelas e campos.

Por ser um framework, o Core Data não está incluído ao nosso projeto na sua criação, portanto devemos importar o framework logo no inicio do código.

```
//Importando o Core Data
import CoreData
```


Seção 3-2

Componentes do Core Data



Podemos dividir as classes do Core Data em dois grupos: As que precisamos configurar uma vez quando nossa aplicação é lançada, e as que utilizaremos no decorrer da aplicação para necessidades específicas.

Seguindo esta linha, vamos separar primeiramente as classes que definimos apenas uma vez, que são:

- **NSPersistentStore:** Representa um armazenamento persistente: Um local em seu dispositivo que armazena informações. Normalmente é utilizado o banco de dados SQLite, mas o Core Data também oferece outros tipos de persistências, como Atomic e In-memory;

- **NSPersistentStoreCoordinator:** Representa um **coordenador** de armazenamento persistente, responsável por coordenar entre os armazenamentos, modelos de objetos e contextos de objetos. Felizmente, isso é feito de forma fácil para criar e configurar. Com isso, a classe executa toda a tarefa complicada enquanto a aplicação roda;

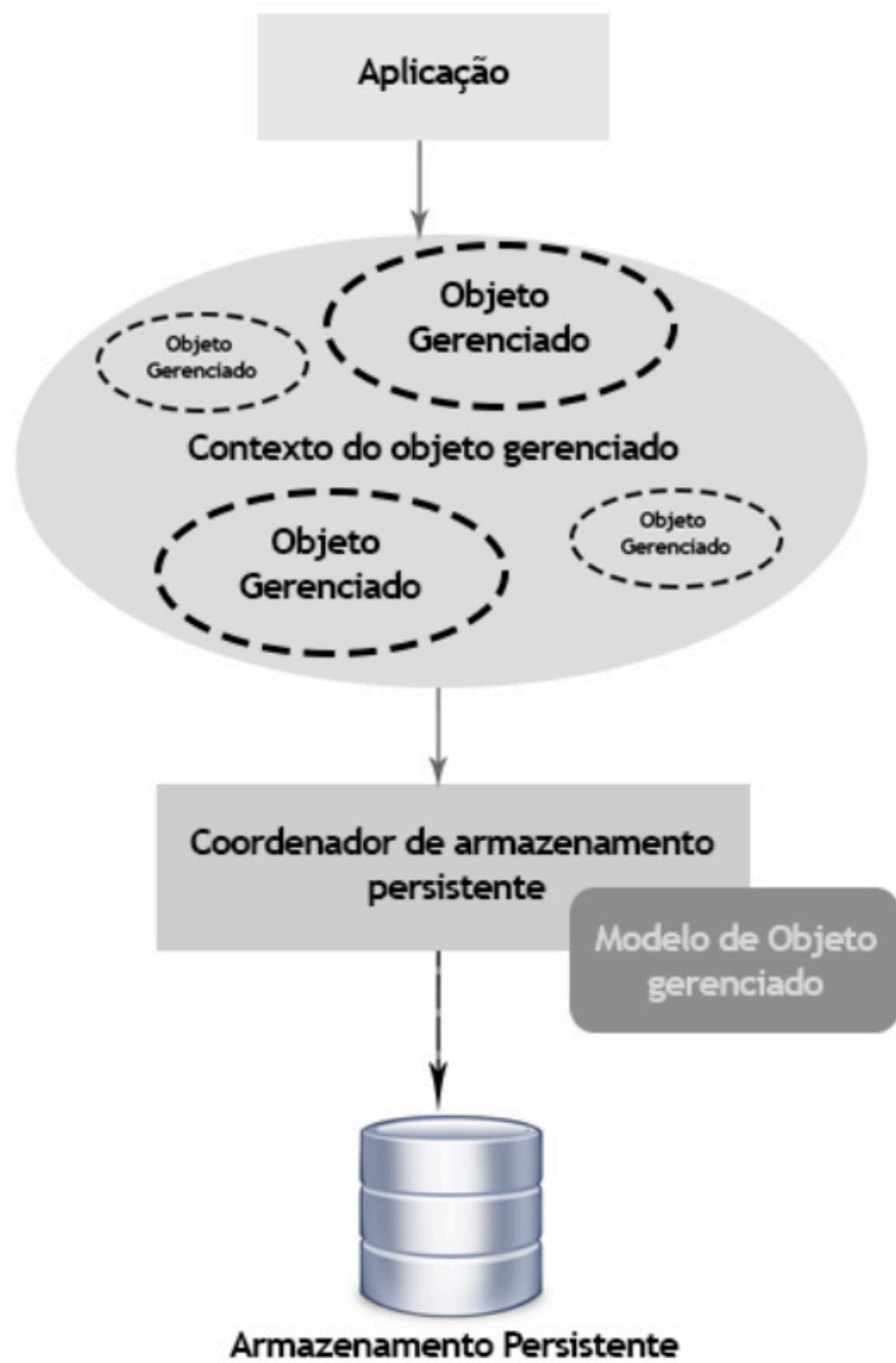
- **NSManagedObjectModel:** Representa um modelo de objeto gerenciado. É um objeto semelhante ao modelo de dados de um banco de relacional, onde é possível definir entidades, atributos e relacionamentos, semelhante ao modelo entidade relacionamento **(ER)**. É possível criar o modelo de objetos diretamente via código, porém o XCode fornece um editor de modelo de objetos que permite realizar a criação e gerenciamento de modelos de objetos que sua aplicação irá carregar e utilizar.

As próximas classes são utilizadas ao longo da aplicação, e são:

- **NSManagedObjectContext:** Representa o contexto do objeto gerenciado, ou apenas o contexto do objeto. O contexto do objeto fornece 'lugares' para os objetos 'viverem'. Criamos objetos no contexto do objeto, recuperamos objetos de dentro do contexto do objeto, e dizemos ao contexto do objeto para salvar. O contexto do objeto cuida de se comunicar com o coordenador do armazenamento persistente para guardar e resgatar objetos. Os contextos de objetos são extremamente utilizados em aplicativos que utilizam o Core Data;

- **NSManagedObject:** Representa um objeto gerenciado, ou um objeto que o Core Data gerencia. Podemos dizer que funcionam como linhas de uma base de dados. Objetos gerenciados possuem atributos e relacionamentos, que foram definidos por um modelo de objeto, e utilizam uma codificação chave-valor **(KVC)**, assim podemos facilmente definir ou resgatar as informações. Também podemos criar uma subclasse do **NSManagedObject** para criarmos propriedade que casam com nosso modelo de objetos e métodos que trabalham com os objetos gerenciados.

Na imagem a seguir podemos ver as classes do Core Data e como elas funcionam juntas:



Seção 3-3

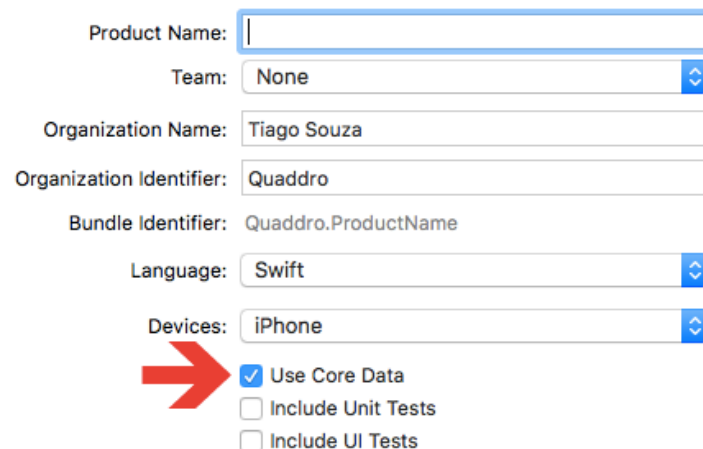
Core Data em um novo projeto



Para adicionar o Core Data em um novo projeto:

1 - Crie um novo projeto, do tipo **Single View App**;

2 - Na tela de seleção de nome da aplicação, selecione a caixa **Use Core Data**, como a imagem abaixo:

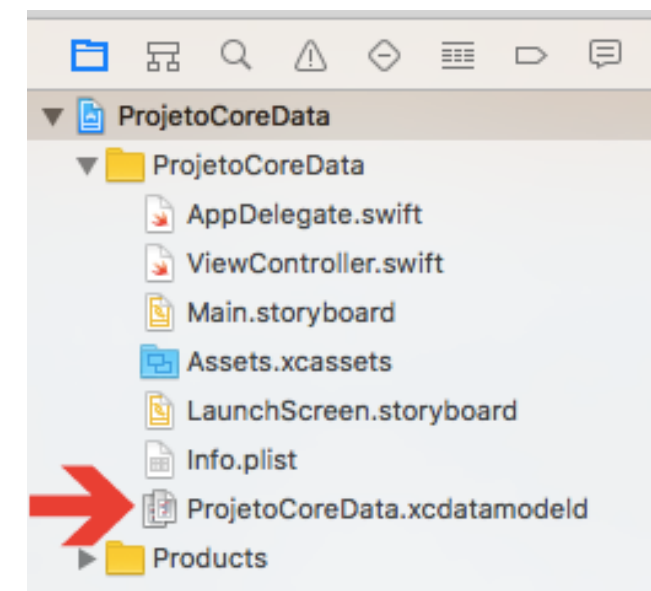


Selecione o item **Use Core Data** estamos dizendo ao XCode para adicionar o Core Data Framework em nosso projeto.

3 - Acesse o arquivo **AppDelegate.swift**. Veja que no topo do arquivo você irá encontrar o **import CoreData** feito automaticamente.

Além dessa importação, nesse arquivo também serão criados algumas propriedades e métodos que nos auxiliarão a persistir nossos dados. Os pontos do código **// MARK: - Core Data stack**, e **// MARK: - Core Data Saving support**, identificam esses elementos;

4- Note que o XCode também criou um modelo que será carregado em uma instancia da classe **NSManagedObjectContext** na sua aplicação. Para isso, o XCode criou automaticamente uma entrada com a extensão **.xcdatamodeld**, conforme a imagem abaixo:



Esta entrada é um diretório que contém artefatos para desenvolver o modelo de objetos. Quando compilarmos a aplicação, o XCode cria um diretório do tipo **.momd** dentro do pacote da aplicação.

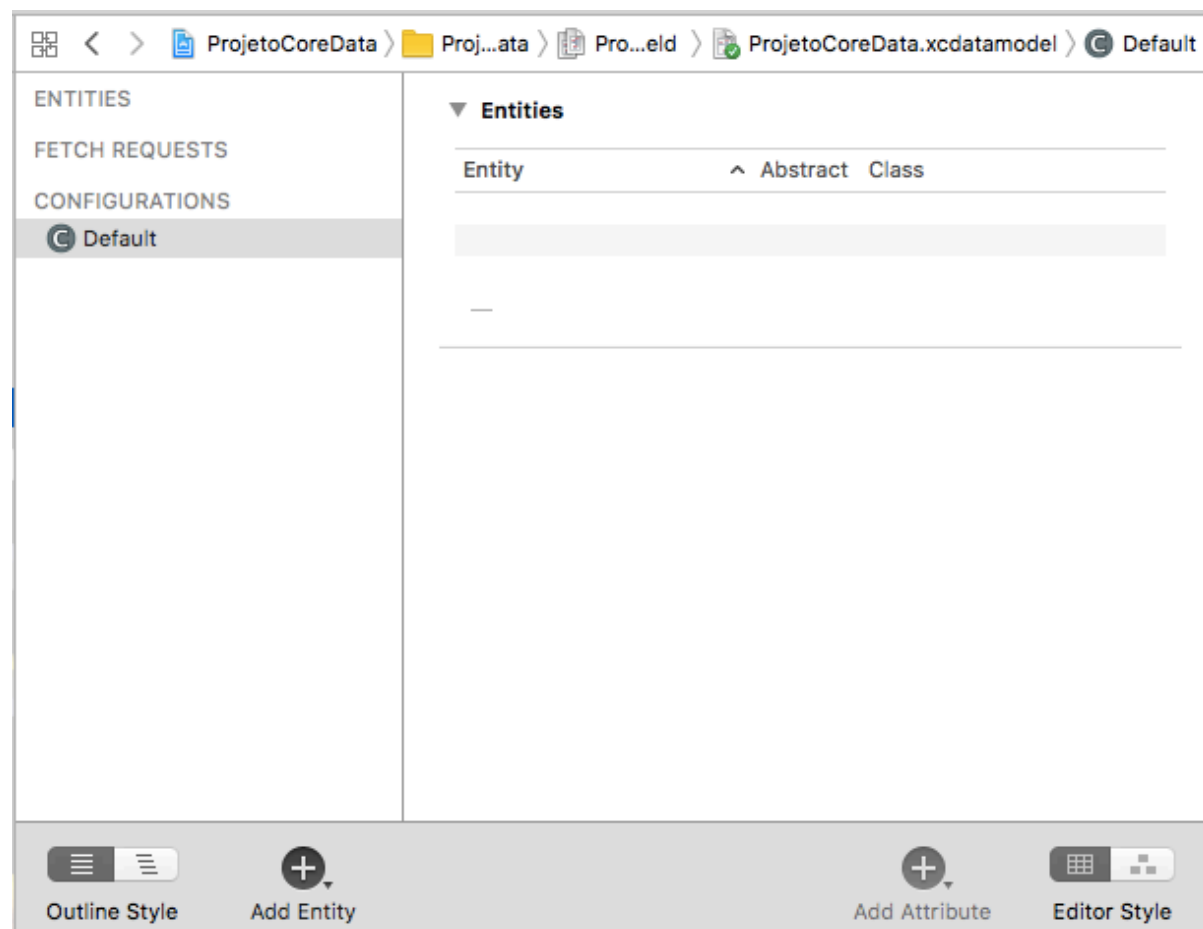
Seção 3-4

Criando um modelo de objetos



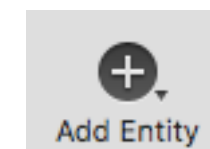
Ao inserirmos o Core Data em nosso projeto, podemos utilizar o arquivo com a extensão **.xcdatamodeld** para criarmos estruturas de dados para a nossa aplicação.

A interface disponível nesse arquivo se assemelha com as interfaces das ferramentas de modelagem de banco de dados, e de certa forma, possuem a mesma intenção, definir a estrutura de dados de forma simplificada e visual.



Assim como uma ferramenta de modelagem de dados de bancos de dados, quando trabalhamos com modelos de objetos, utilizamos a criação de entidades e suas propriedades. Também podemos definir os relacionamentos das entidades para facilitarmos a comunicação da nossa aplicação com estes modelos.

Para criarmos uma entidade, basta selecionarmos o arquivo com a extensão **.xcdatamodeld** em nosso **Project Navigator**, clicarmos no botão **Add Entity** que fica no canto inferior esquerdo.

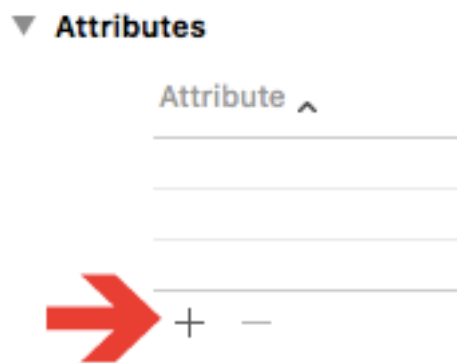


Ao realizar esta operação, o XCode irá criar uma nova entidade chamada **Entity**. Vamos mudar o nome desta entidade para **Pessoas**. Fazemos isso com um duplo clique sobre o objeto.



Com nossa entidade definida, podemos nos concentrar em seus **atributos**. Podemos chamar de atributos, informações que fazem parte de uma entidade. No caso da entidade Pessoa, podemos dizer que **Nome** e **Idade** são dois atributos válidos, já que uma pessoa provavelmente tem um Nome e uma idade.

Para adicionarmos atributos para nossa entidade, clicamos na sinal **+** na coluna **Attributes**.



Perceba que ao clicar no botão +, um novo atributo é adicionado a nossa entidade, permitindo que possamos editar seu nome e seu tipo.

Os principais tipos são:

- Integer 16, 32, 64
- Decimal
- Double
- Float
- String
- Boolean
- Date

No caso do nosso projeto, utilizaremos o tipos **String** para o **nome**, e o tipo **Integer 16** para **idade**.

▼ Attributes

Attribute ▼	Type
S nome	String
N idade	Integer 16
+ -	

Com a entidade e seus atributos definidos, já temos uma estrutura mínima para começarmos a trabalhar com o Core Data.

Seção 3-5

Avançando no projeto



Agora que já temos nossa base pronta precisamos, criar objetos para testarmos se nosso modelo básico é funcional.

Definindo acessos aos elementos

O primeiro passo é definir o acesso aos elementos já existentes no arquivo **AppDelegate**. O ideal é criarmos uma **extension** da classe que controla as view controllers, e a essa extensão acionarmos esse acesso:

```
extension UIViewController{

    //Acesso aos elementos que estão no AppDelegate
    var appDelegate : AppDelegate{

        return UIApplication.shared.delegate as! AppDelegate

    }

    //Objeto que dá acesso ao contexto do banco de dados
    var context : NSManagedObjectContext {

        return appDelegate.persistentContainer.viewContext

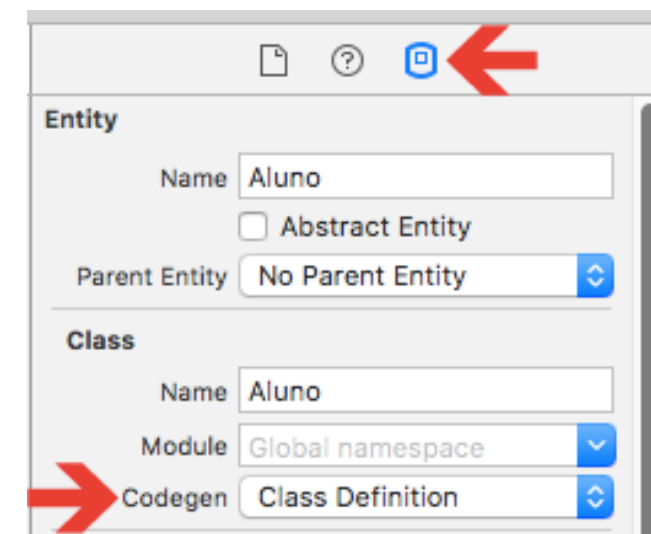
    }

}
```

Além do acesso ao AppDelegate, também definimos o objeto que irá gerenciar o contexto do nosso banco de dados.

Definindo a classe do banco de dados

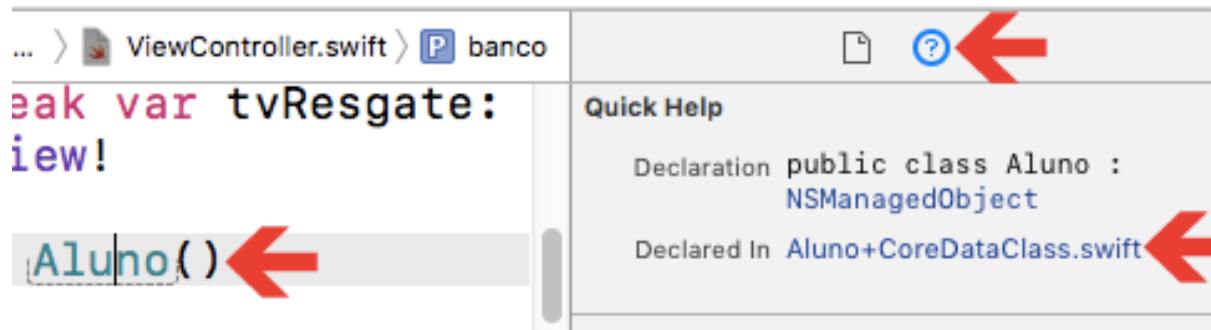
Com o nosso arquivo **.xcdatamodeld** selecionado, podemos indicar que o mesmo terá uma classe que cuidará das suas propriedades. Essa classe pode ser feita automaticamente ao selecionarmos nosso arquivo no pacote, e no painel **Data Model Inspector**, indicarmos a opção **Class Definition** para o campo **Codegen**:



#Dica!

É importante salientar que a **classe** criada terá o mesmo nome da **entidade (Entity)** que cuida de um determinado grupo de atributos.

Podemos acessar esse arquivo com a classe das entidades de uma forma simples. Basta criarmos uma instância dessa classe, e com o cursor de texto em cima dessa instância, utilizarmos o painel **Quick Help** para clicar sobre o arquivo:



Nesse arquivo podemos propor algumas formatações dos nossos dados, como e exemplo adiante:

```
import Foundation
import CoreData

@objc(Aluno)
public class Aluno: NSObject {

    override public var description: String{

        return "Nome: \(self.nome!) \nSobrenome:
               \(self.sobrenome!) \nIdade: \(self.idade)
               \n"
    }
}
```

Adiante teremos algumas classes que nos auxiliam quando trabalhamos com Core Data.

Seção 3-6

NSManagedObjectContext



Herança: NSObject > NSManagedObjectContext

Uma instância de **NSManagedObjectContext** representa um único uma coleção de objetos gerenciados de uma entidade.

Esses esses elementos formam um grupo de objetos de modelo relacionados que representam uma exibição internamente consistente de um ou mais armazenamentos persistentes.

Uma instância desse objeto gerenciado existe em um único contexto, mas várias cópias de um objeto podem existir em contextos diferentes. Assim, a exclusão de objeto é delimitada para um contexto específico.

Principais métodos

```
//Método que tenta efetuar alterações não salvas de objetos  
registrados em uma entidade:  
func save() throws
```




Herança: NSObject > NSPersistentRequest > NSFetchRequest

Podemos consultar os objetos inseridos em nossa base de dados e retorná-los de volta para nossa aplicação.

Utilizamos a Classe **NSFetchRequest** para criarmos um critério usado para consultar informações de um armazenamento. Podemos dizer que é estamos dizendo para a aplicação, qual tabela de um banco de dados ela estaria utilizando para consulta.

Uma instância desta classe coleta o critério necessário para selecionar, e, opcionalmente, ordenar um grupo de objetos gerenciados, ou informações sobre os registros inseridos em um armazenamento persistente.

Uma consulta deve conter uma descrição da entidade (uma instância da classe **NSEntityDescription**) que especifica qual entidade consultar. Também pode conter um atributo (Instancia da classe **NSPredicate**) que especifica que propriedade selecionar, e um array com descrições de organização (Intancia da classe **NSSortDescriptor**)

Principais propriedades

Propriedade	Tipo	Descrição
entityName	String?	Retorna o nome da entidade configurada
entity	NSEntityDescription?	Define/Retorna a entidade definida pelo receptor
predicate	NSPredicate?	O atributo do receptor
fetchLimit	Int	O limite da busca do receptor
sortDescriptors	[NSSortDescriptor]?	A descrição da ordenação do receptor

Principais métodos

```
//Método que inicializa uma solicitação de busca com um determinado nome de entidade:  
convenience init(entityName: String)
```

Seção 3-8

NSManagedObject



Herança: NSObject > NSManagedObject

NSManagedObject é uma classe genérica que implementa todo o comportamento básico necessário de um Objeto modelo de Core Data. É recomendado criar subclasses customizadas de NSManagedObject, embora isso não seja necessário.

Podemos dizer que um objeto NSManagedObject age como container que armazena de maneira eficiente as propriedades definidas com sua **NSEntityDescription** associada. Um Objeto NSManagedObject pode receber diversos tipos de atributos, como strings, datas e números.

Principais propriedades

Propriedade	Tipo	Descrição
objectID	NSManagedObjectID	Retorna o id do receptor
entity	NSEntityDescription	Retorna a entidade definida pelo receptor
managedObjectContext	NSManagedObjectContext?	Retorna o contexto do objeto gerenciado em que o receptor esta registrado
hasChanges	Bool	Retorna se o receptor foi inserido, deletado ou teve alterações não salvas
isInserted	Bool	Retorna se o receptor foi inserido em um contexto de objeto gerenciado

Principais métodos

```
//Método que retorna um valor para uma propriedade a partir de uma chave:  
func value(forKey key: String) -> Any?
```

```
//Método que define um valor para uma determinada propriedade para uma determinada chave:  
func setValue(_ value: Any?, forKey key: String)
```

```
//Método que valida uma propriedade para uma determinada chave:  
func validateValue(_ value: AutoreleasingUnsafeMutablePointer<AnyObject?>, forKey key: String) throws
```

Seção 3-9

NSPredicate



Herança: NSObject > NSPredicate

A classe **NSPredicate** é utilizada para criar uma condição lógica para definir uma busca. Podemos usar um NSPredicate para descrever objetos em um armazenamento e em objetos que estão em memória. Podemos fazer uma analogia de um objeto NSPredicate com uma expressão regular, por ser um padrão, mesmo sua sintaxe sendo diferente.

Principais métodos

```
//Cria e retorna um novo atributo substituindo os valores  
em um array em um formato de cadeia, e analisando os  
resultados:  
init(format predicateFormat: String, argumentArray  
arguments: [Any]?)
```

```
//Método inicializador que cria e retorna um atributo através  
da substituição dos valores em uma lista de argumentos em  
um formato de cadeia e analisando os resultados:  
init(format predicateFormat: String, arguments argList:  
CVaListPointer)
```

Capítulo 4



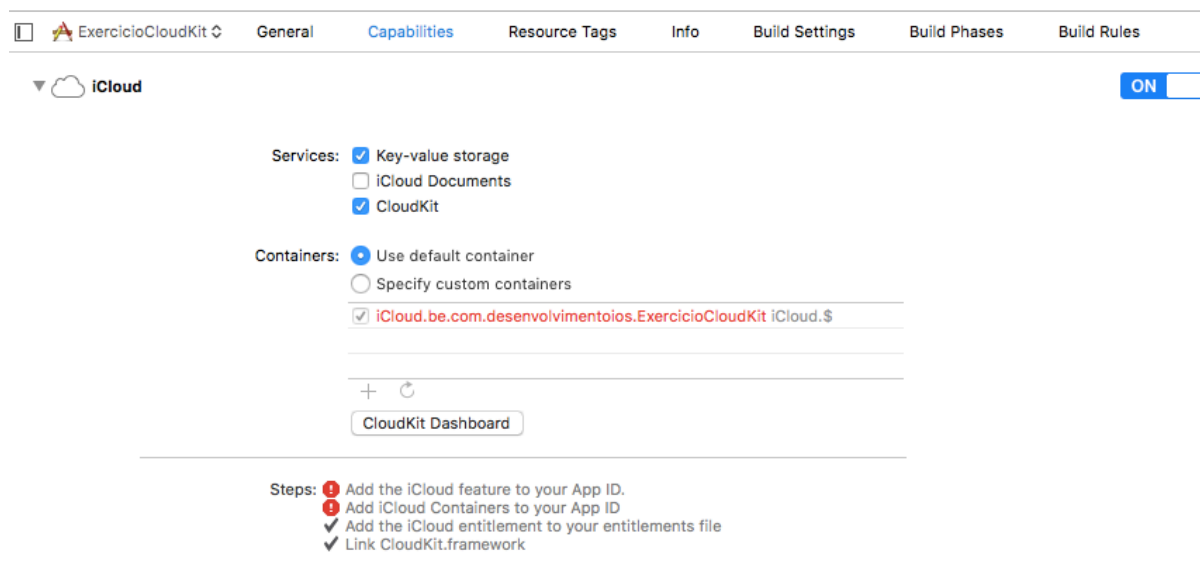
CloudKit



O **CloudKit** é um app service disponível somente para aplicativos que serão distribuídos pela **Apple Store**. O CloudKit requer configurações adicionais em seu projeto do Xcode, e também requer que o programador tenha acesso a uma conta de desenvolvedor paga.

Com o CloudKit podemos salvar e carregar dados relacionados ao nosso aplicativo diretamente na nuvem.

Devemos ativar o CloudKit usando o painel **Capabilities** no Xcode. Não será necessário editar direitos diretamente no Xcode ou na conta do desenvolvedor. O acesso aos recursos de CloudKit estão automaticamente habilitados na sua conta de desenvolvedor paga.



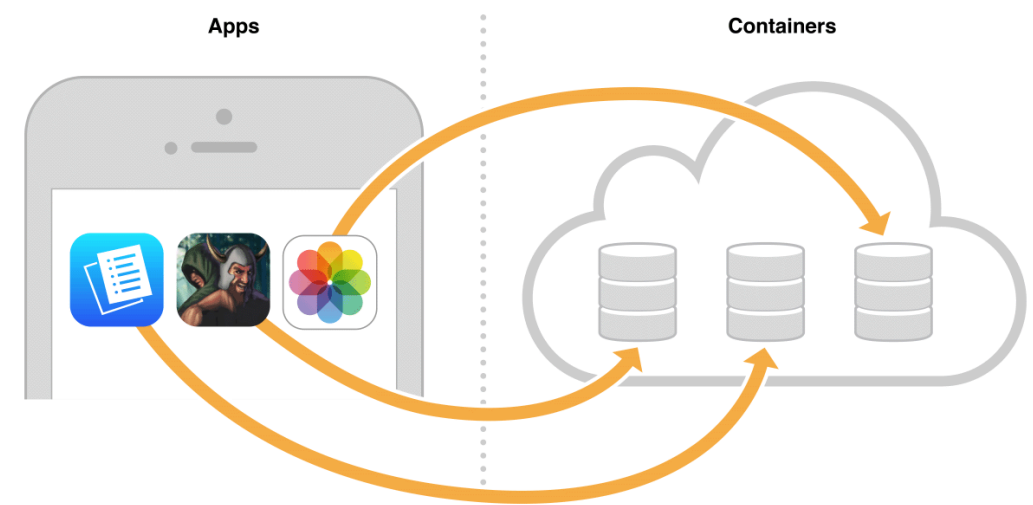
Sobre containers e bancos de dados

Vários aplicativos e usuários têm acesso ao iCloud. Os diferentes dados são separados e encapsulados em partições chamadas **containers**.

Os containers referentes à sua aplicação não podem ser acessados por aplicativos de outro desenvolvedor. No entanto, seus aplicativos podem compartilhar containers.

Vários aplicativos podem compartilhar o mesmo container, da mesma forma que um aplicativo pode usar vários containers. Há um container padrão por aplicativo, mas podemos criar containers personalizados adicionais.

O **identificador** do container padrão coincide com o **ID** do pacote do aplicativo. Os outros IDs de contêiner especificados precisam ser exclusivos em todas as contas de desenvolvedor.



Seção 4-2

Preparando o esquema de dados



Por ser um framework, o **CloudKit** não está incluído ao nosso projeto na sua criação, portanto devemos importar o framework para os arquivos ViewController que utilizarão a ferramenta como base.

```
//Início do Projeto
import UIKit

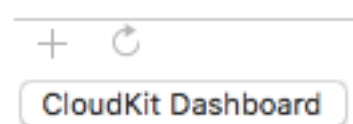
//Importando o QuickLook
import CloudKit
```

Acessando a CloudKit Dashboard

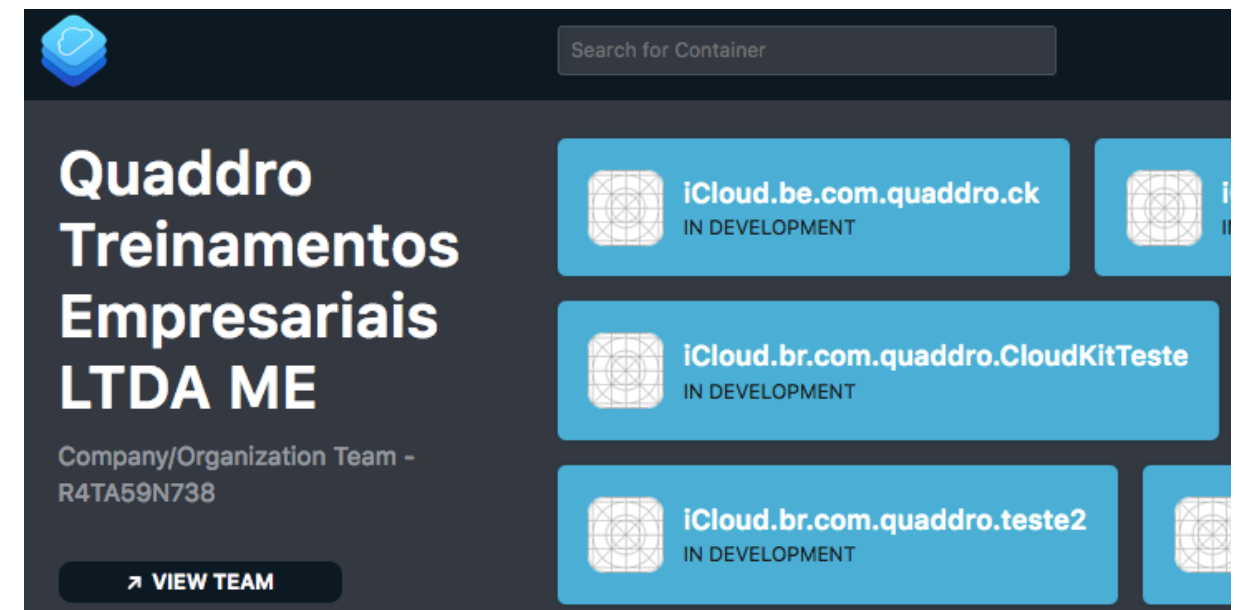
Utilizamos a **CloudKit Dashboard** para gerenciar os esquemas e registros de containers do CloudKit.

Os esquemas descrevem a organização de registros, campos e relacionamentos em um banco de dados. Em um banco de dados relacional, um tipo de registro corresponde a uma tabela e um campo corresponde a uma linha em uma tabela.

Podemos desenvolver toda essa estrutura através dessa Dashboard exclusiva para o trabalho com o CloudKit. Para ter acesso a Dashboard, podemos clicar no botão correspondente no painel **Capabilities:**



Já na Dashboard, devemos escolher o nosso projeto no menu localizado no canto superior esquerdo:



Quando acessamos um container, temos a disposição dois painéis de comandos:

O primeiro painel de cor azul, é responsável por comandos relacionados aos apps que estão ainda em fase de desenvolvimento. Os recursos adicionados nesse painel, terão efeitos apenas nos apps testados via Xcode ou TestFlight;

O segundo painel de cor verde, é responsável por comandos relacionados aos apps que já foram publicados. Os recursos adicionados nesse painel, terão efeito diretamente nos apps publicados.

Criando esquemas de dados para salvar registros




Durante o desenvolvimento, é fácil criar esquemas utilizando o CloudKit. Quando salvamos objetos de registro em um banco de dados, os tipos de registro associados e seus campos são criados automaticamente, e depois ficam disponíveis para serem acessados pela dashboard.

É importante salientar, que podemos criar todo o esquema de dados manualmente dentro da dashboard, mas tal processo é um pouco mais trabalhoso, e demanda muitos passos. Vamos optar por criar o esquema de dados diretamente via código.

Quando criamos tipos de dados, esses tipos devem respeitar a tabela adiante. Nela temos os tipos suportados pelo CloudKit:

Tipo de Dado	Classe	Descrição
Asset	CKAsset	Arquivo que está associado a um registro
Date/Time	NSDate	Data ou hora
Double	NSNumber	Números decimais
Int(64)	NSNumber	Números inteiros
Location	CLLocation	Coordenadas de geolocalização
Reference	CKReference	Relações entre objetos
String	NSString	Dados de texto
List	NSArray	Lista de dados em formato de array

No projeto adiante, vamos criar um simples cadastro com um campo, e em seguida utilizar esse dado resgatado para ser exibido:



Lista de Doces Preferidos

Cadastrar

Resgatar

Em seguida, vamos ao arquivo de classe ViewController.swift e implementar os seguintes códigos, Outlets e Actions, levando em consideração um projeto que tenha sido criado com os recursos de CloudKit habilitados, como vimos anteriormente.


```

import UIKit
import CloudKit

class ViewController : UIViewController {

    //MARK: ----- Outlets
    @IBOutlet weak var textFieldNome: UITextField!
    @IBOutlet weak var testViewResultado: UITextView!

    //MARK: ----- Propriedades
    //Nome da tabela a ser utilizada para gravação
    let meuRecordType = "ListaDoces"

    //Definindo qual base de dados será utilizada (Publico
    ou privada)
    let dataBase = CKContainer.default().publicCloudDatabase

    //MARK: ----- Actions
    @IBAction func cadastrar(_ sender: UIButton) {

        if !(textFieldNome.text!.isEmpty){

            let nome = textFieldNome.text!

            //Definindo o objeto a ser salvo
            let doce = CKRecord(recordType: meuRecordType)

            doce.setObject(nome as CKRecordValue?, forKey:
"nome")

            //Salvando os dados
            self.dataBase.save(doce, completionHandler: {
                (record, erro) in

                    if erro != nil{

                        print("Erro \(erro!)")

                    } else {

                        print("Registro gravado, com sucesso")

                    }

                })

        }

    }
}

```

```

@IBAction func resgatar(_ sender: UIButton) {

    var resultado = ""

    //Definindo o objeto de pesquisa
    let query = CKQuery(recordType: meuRecordType,
    predicate: NSPredicate(value: true))

    //Pesquisando
    self.dataBase.perform(query, inZoneWith: nil) {
        (retorno, erro) in

            if erro != nil{

                print("Erro: \(erro!)")

            }

            if let _ = retorno{

                for doce in retorno!{

                    resultado += "Doce: \(doce["nome"]!)\n\n"

                }

                DispatchQueue.main.async {

                    self.testViewResultado.text = resultado

                }

                print(resultado)

            }

        }

    }
}

```

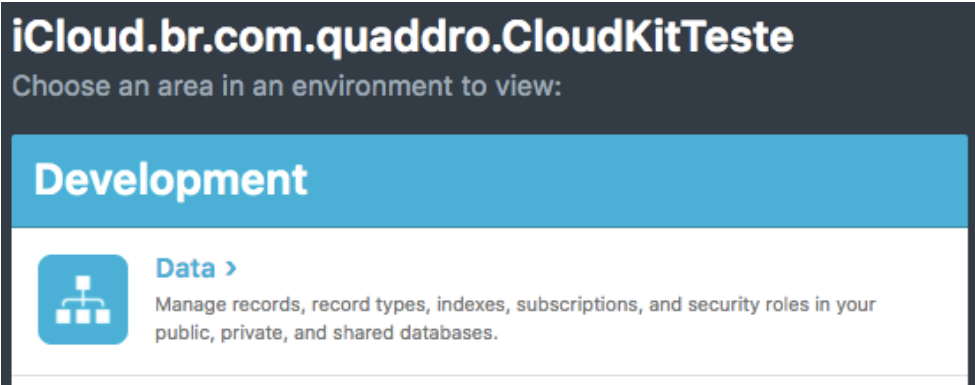
Podemos executar o projeto, e efetuar o primeiro cadastro. Porém vamos perceber que ao efetuar o resgate, a seguinte resposta será exibida em console:

```

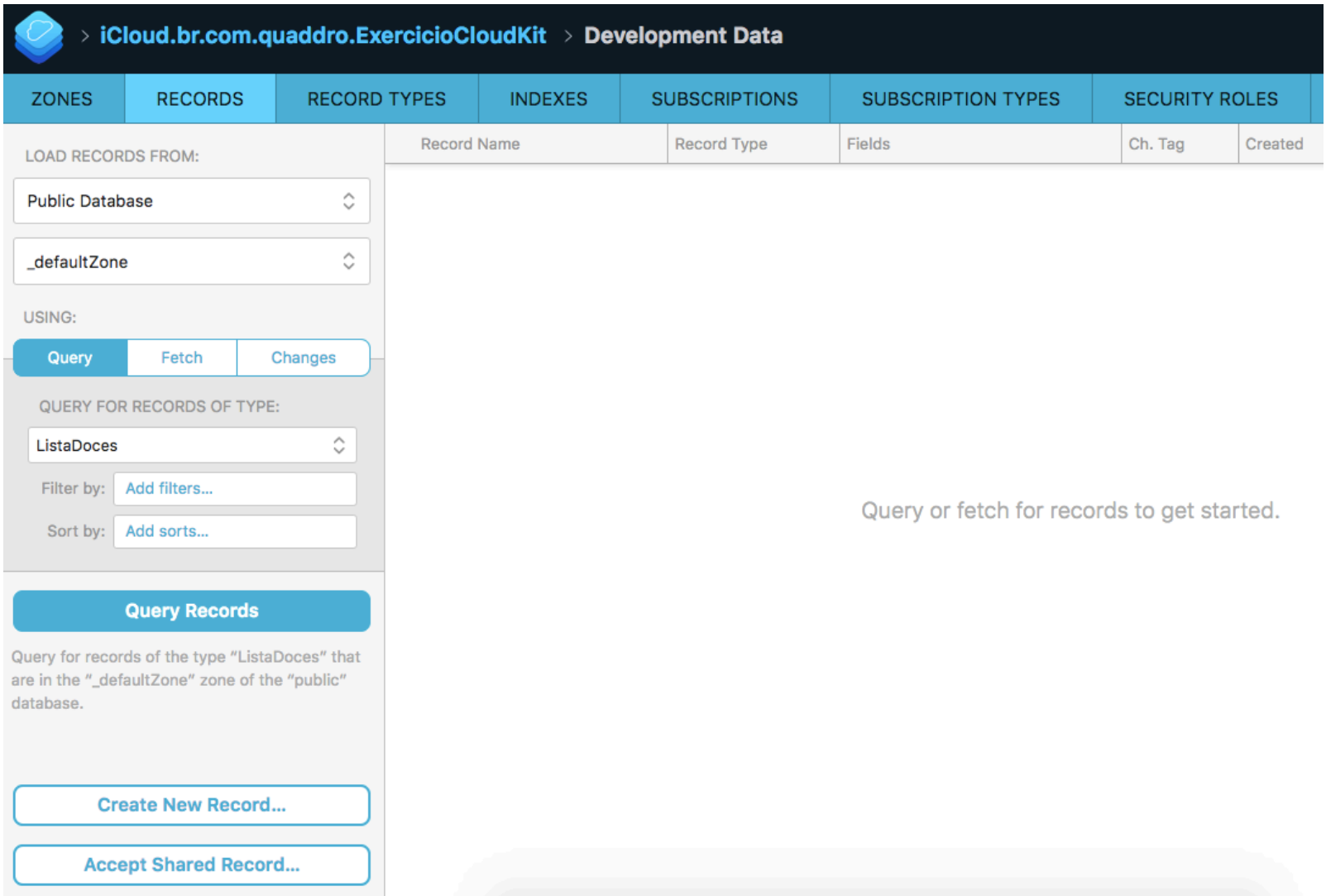
Erro: <CKError 0x604000449f00: "Invalid Arguments" (12/2015);
server message = "Field 'recordName' is not marked queryable";
uuid = 68A5331A-5C49-49C8-92D5-65AF2C3369F6; container ID =
"iCloud.br.com.quaddro.ExercicioCloudKit">

```

Isso acontece por conta de um índice de pesquisa que temos que incluir no CloudKit. Vamos até a dashboard, em seguida vamos navegar até o container que estamos utilizando, e acessar o botão **Data** para projetos em desenvolvimento:



A partir daí estamos na região da dashboard que nos auxilia a controlar nossas tabelas e registros.



Vamos até a aba Indexes, nele vamos encontrar três índices para o nosso campo nome:

ZONES	RECORDS	RECORD TYPES	INDEXES	SUBSCRIPTIONS	SUBSCRIPTION TYPE
SHOW INDEXES FOR TYPE:					
ListaDoces 3 indexes		Fields	Index Type		
		nome	Single Field	QUERYABLE	✗
		nome	Single Field	SEARCHABLE	✗
		nome	Single Field	SORTABLE	✗
		Add Index			
		Remove All Indexes < View Record Type Save Record Type			

Vamos adicionar mais um índice a lista, clicando no botão **Add Index**. Esse campo terá o **Field recordName**, e o **Index Type Queryable**. Com isso teremos a opção de pesquisar por toda a nossa tabela:

recordName

Single Field

QUERYABLE

✗

Add Index

Por ultimo, clicamos no botão **Save Record Type**, para finalizar e salvar o processo.

Podemos ir até a guia Records, e efetuar uma pesquisa clicando no botão Query Records. Com isso será exibido o registro q foi feito anteriormente:

ZONESRECORDSRECORD TYPESINDEXESSUBSCRIPTIONSSUBSCRIPTION TYPE

LOAD RECORDS FROM:

Public Database

_defaultZone

USING:

QueryFetchChanges

QUERY FOR RECORDS OF TYPE:

ListaDoces

Filter by: Add filters...

Sort by: Add sorts...

Query Records

Record Name	Record Type	Fields
8408F784-4CDB-42C3-8DCE-EB04E3DD3CBA	ListaDoces	nome Brigadeiro

Ao retornar a nossa aplicação, podemos proceder o resgate dos dados:

Carrier1:32 PM

Lista de Doces Preferidos

Nome do doce...

Cadastrar

Doce: Brigadeiro

Resgatar

É importante que seja observado que todo o processo de criação da tabela, campo e índice dos campos é criado automaticamente quando rodamos a nossa aplicação e solicitamos a criação do primeiro registro. Caso contrário, toda a estrutura de dados pode ser criada manualmente via dashboard, e referenciada da mesma forma no código.

Através da dashboard podemos fazer todo o gerenciamento dos dados, como pesquisa, exclusão de registros, ou até mesmo de uma tabela inteira.

Também é importante salientar que os dados gravados em zonas privadas, só poderão ser acessadas via app pelo próprio usuário que fez o registro, já os dados gravados em zonas públicas podem ser acessados por qualquer usuário da aplicação.



Herança: NSObject > CKContainer

Através da classe **CKContainer** podemos acessar e resgatar dados de um registro do CloudKit.

Esses dados podem ter diferentes tipos de **encapsulamento**, onde os dados possam ser acessados de forma **pública**, de forma com que todos os usuários do aplicativo tenham acesso as informações, ou de forma **privada**, onde cada usuário tem acesso apenas aos dados cadastrados por ele.

```
//Método que recupera o banco de dados com o escopo apropriado:  
func database (with databaseScope: CKDatabaseScope) ->  
CKDatabase
```

Principais propriedades

Propriedade	Tipo	Descrição
privateCloud-Database	CKDatabase	Retorna o banco de dados que contém os dados privados de um usuário
publicCloud-Database	CKDatabase	Define o Retorna o banco de dados que contém dados públicos compartilhado por todos os usuários

Principais métodos

```
//Método que retorna o objeto de container padrão para  
gerenciar o conteúdo do aplicativo atual:  
class func `default`() -> CKContainer
```



A classe **CKRecord** gera um dicionário baseado em pares key-value que podemos usar para adicionar os dados de um registro com o CloudKit.

Embora os registros agem como dicionários, ainda existem limitações para os tipos de valores que você pode atribuir às chaves. A seguir estão os tipos de objeto que a classe CKRecord oferece suporte:

- **NSString:** Armazena quantidades pequenas de texto. Sequências de caracteres podem ter qualquer comprimento. Caso queira armazenar grandes quantidades de texto, utiliza um **CKAsset**;
- **NSNumber:** Armazena quaisquer informações numéricas, incluindo inteiros e números decimais;
- **NSData:** Armazena bytes arbitrários de dados. Não é recomendado o uso objetos NSData para armazenar grandes arquivos de dados binários, nesse caso recomenda-se um **CKAsset**. Os campos de dados não são pesquisáveis;
- **NSDate:** Armazena informações de dia e hora de forma acessível;
- **NSArray:** Armazena um ou mais objetos de qualquer tipo. Podemos armazenar matrizes de strings, matrizes de números, matrizes de referências, e assim por diante;

- **CLLocation:** Armazena dados de coordenadas geográficas;
- **CKAsset:** Armazena um grande quantidade de dados em um registro;
- **CKReference:** Cria um link de referência para um registro relacionado. Uma referência armazena o ID do registro de destino.

#Dica!

A tentativa de especificar objetos de outro tipo gerará um erro e falhará. Campos de todos os tipos são pesquisáveis a menos que se indique outra forma.

Principais métodos

//Método que retorna o valor da chave fornecida que está armazenada no registro:

```
func object (forKey key: String) -> CKRecordValue?
```

//Método que define o valor para a chave especificada:

```
func setObject (_ object: CKRecordValue?, forKey key: String)
```



A classe **CKDatabase** estabelece um canal de acesso e execução de operações relacionados aos dados públicos e privados de um container de aplicativo.

A classe CKDatabase oferece métodos para acessar registros, zonas de registro (pública ou privada) e campos sem a necessidade da criação de um objeto para um registro em seu código.

Os métodos permitem buscar, salvar ou excluir um único item de forma assíncrona, e também permite processar os resultados em um segmento em background. Podemos também procurar por registros no banco de dados.

Principais propriedades

Propriedade	Tipo	Descrição
databaseScope	CKDatabase-Scope	Retorna tipo de dado, público ou privado

Principais métodos

```
//Método que procura a zona especificada de forma assíncrona para registros que correspondem aos parâmetros da consulta:  
func perform(_ query: CKQuery, inZoneWith zoneID: CKRecordZoneID?, completionHandler: @escaping ([CKRecord]?, Error?) -> Void)
```

```
//Método que executa a operação especificada de forma assíncrona em relação ao banco de dados atual:  
func add (_ operation: CKDatabaseOperation)
```

```
//Método que salva um registro de forma assíncrona, com uma prioridade baixa, para o banco de dados atual, se o registro nunca foi salvo ou se é mais recente do que a versão no servidor:  
func save (_ record: CKRecord, completionHandler: @escaping (CKRecord?, Error?) -> Void)
```

```
//Método que Exclui do banco de dados atual o registro especificado de forma assíncrona, com uma prioridade baixa:  
func delete(withRecordID recordID: CKRecordID, completionHandler: @escaping (CKRecordID?, Error?) -> Void)
```



Um objeto **CKQuery** gerencia os critérios a serem aplicados ao procurar registros em um banco de dados. Podemos criar um objeto de consulta como uma etapa no processo de pesquisa.

O objeto de consulta armazena os parâmetros de pesquisa, incluindo o tipo de registros a serem pesquisados, os critérios de correspondência (predicate) a serem aplicados e os parâmetros de classificação a serem aplicados aos resultados.

Principais Propriedades

Propriedade	Tipo	Descrição
recordType	String	Tipo de registro a ser pesquisado
predicate	NSPredicate	Critérios de pesquisa a serem usados quando os registros coincidem



Caderno de exercícios

Exercícios relacionados ao capítulo 1

Webservices



Exercício 1

Criar uma aplicação que dado determinado cep (pode ser inserido diretamente na URL) exiba em console as informações conforme abaixo:

Utilize o servidor “via cep” para resgatar as informações.

CEP: 01418-902

UNIDADE:

UF: SP

BAIRRO: Cerqueira César

LOCALIDADE: São Paulo

GIA: 1004

COMPLEMENTO: 1000

LOGRADOURO: Alameda Santos

IBGE: 3550308

Exercícios relacionados ao capítulo 3

Core Data



Exercício 1

Crie uma aplicação que consiga cadastrar usuários (com as propriedades **nome** e **idade**). Essa aplicação também deve fazer o resgate de todos os registros de uma única vez. A exibição do resgate pode ser feita em console.

***** Resgate de Dados *****

Nome: Leonardo

Idade: 23

Nome: Raphael

Idade: 21

Nome: Michelangelo

Idade: 20

Nome: Donatelo

Idade: 22

iPhone SE – iOS 10.0 (14A345)

Carrier 5:46 PM

Nome

Idade

Cadastrar Resgatar

Exercícios relacionados ao capítulo 4

CloudKit



Exercício 1

Crie uma aplicação que seja capaz de salvar registros de alunos com as propriedades, nome e sobrenome usando o **CloudKit** para este fim.

É necessário também fazer o resgate de todos os elementos já cadastrados. A exibição dos mesmos pode ser feita em console.

