MBaaS in iOS: Backendless - Data Services

Mobile Computing - iOS

Objectives

- Students will be able to:
  - explain the purpose of MBaaS
  - explain how MBaaS compares to other database technologies
  - create and configure a Backendless app and its iOS companion
  - utilize Backendless' Data to save/update/retrieve/delete objects

Introduction

- MBaaS — mobile back end as a service — provides a series of services that many mobile apps need, including:
    - cloud storage using a no-sql database
    - user management
    - social media integration
    - push notifications
    - server-side computing
- In this document, alongside the MBaaS demo and the documentation available at a popular MBaaS provider, you will learn how to use MBaaS in your projects. You will still need to do some study in order to master MBaaS. The documentation is quite extensive, and well done. They also have a Slack channel, and you can expect replies there almost immediately.

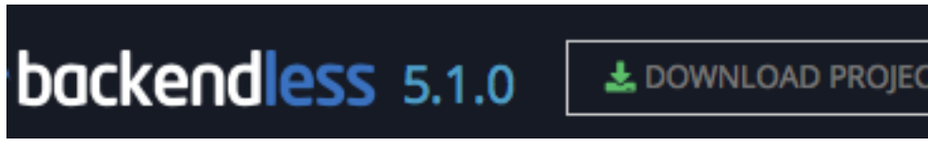
Terminology

- There are two apps at play here - the iOS app that you create in Xcode, and the Backendless app, on their site, that interacts with your iOS app

- In situations where "app" might be ambiguous, we will write **iOS app** or **Be app**

Getting Started With a New Xcode Project

1. Get an account on backendless.com
2. Create a Be app on backendless
3. Download the project template
    1. Choose iOS and Swift



4. Install frameworks using Cocoapods
    1. Edit the Podfile so the platform is 11.3 (instead of 8.0)
    2. In Terminal, cd to the root of the project folder
    3. Issue the command **pod install** (this assumes that you already have CocoaPods installed).
5. Open the .**xcworkspace** file (**not** the .xcodeproj file that we normally use: that will lead to heartatche)
6. Run the project (you may need to clean it first, and verify that the version is 11.3)
7. Verify on the console that an object was created.

Techy Aside*: Getting Started with a Pre-existing Xcode Project

Already have an app? No problem, you can install the Backendless pod at any time

To create a new project with Backendless Pod, follow the steps below:

1. Create a new project in Xcode as you normally would, then close the project.
2. Open a Terminal window, and change the current directory to be the project's directory.
3. Run the following command in the Terminal window, which will create a file with the name `Podfile`.

```
pod init
```

4. Open the created Podfile using a text editor and add the following text inside of the `target` block:

```
pod 'Backendless'
```

5. Save Podfile, return to the Terminal window and run the following command:

```
$ pod install
```

6. Once the pod is downloaded, Xcode project workspace file will be created. This should be the file you use to open the project in Xcode.
7. If you develop with Swift, you will need to add a Swift bridging header. To do that, click the root node in the Project Structure and select the **Build Settings** section. Locate the **Swift Compiler - General** section. Enter the following value into the **Objective-C Bridging Header** field:

```
Pods/Backendless/SDK/ios/backendless/include/Backendless-Bridging-Header.h
```

8. Open `.xcworkspace` file to launch your project, and build it.

*this is a techy aside because we will be downloading the pre-made app from Backendless: but this might be useful in the future, so bear it in mind

Techy Aside*: Getting Started with a Pre-existing Xcode Project

The App ID and API key are available in the backendless app

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

 let APP_ID = "YOUR-APPLICATION-ID" // these are available in your backendless app
 let API_KEY = "YOUR-APPLICATION-IOS-API-KEY"

 var backendless = Backendless.sharedInstance()

 var window: UIWindow?

 func application(application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [NSObject: AnyObject]?) -> Bool {

   backendless.initApp(APP_ID, apiKey:API_KEY)

   return true
 }
}
```

Class Requirements for Backendless

- Instances of a class in an iOS app are stored in Backendless in a **data table** with the same class name. Each row in a data table corresponds to one instance, each column to a property

- When an instance is saved **for the first time**, the data table will be created

- Classes must

  1. be annotated with @objcMembers*

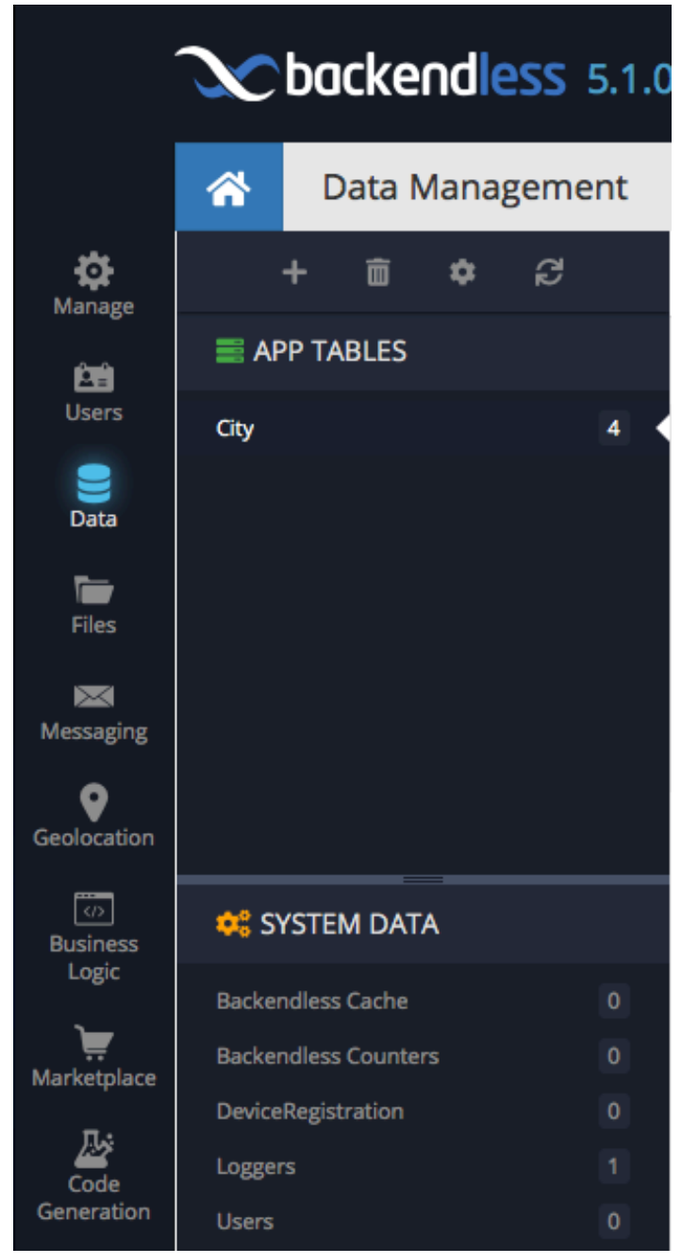
  2. subclass NSObject

  3. be implemented outside of ViewController and AppDelegate classes - putting them in their own .swift file is best

  4. contain a default, publicly accessible, no-argument initializer

  5. contain at least one property

  6. use non-optional properties for Bool, Int, Double, Float



** @objcMembers causes @objc to be prepended to any Swift members, making them available in Objective-C, required since Backendless is written in Objective-C.*

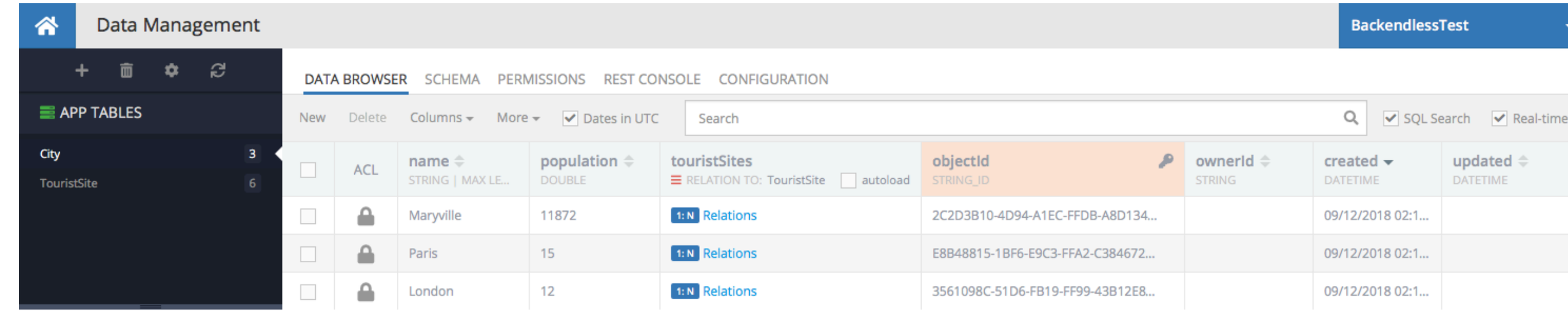
A Class That Follows the Rules

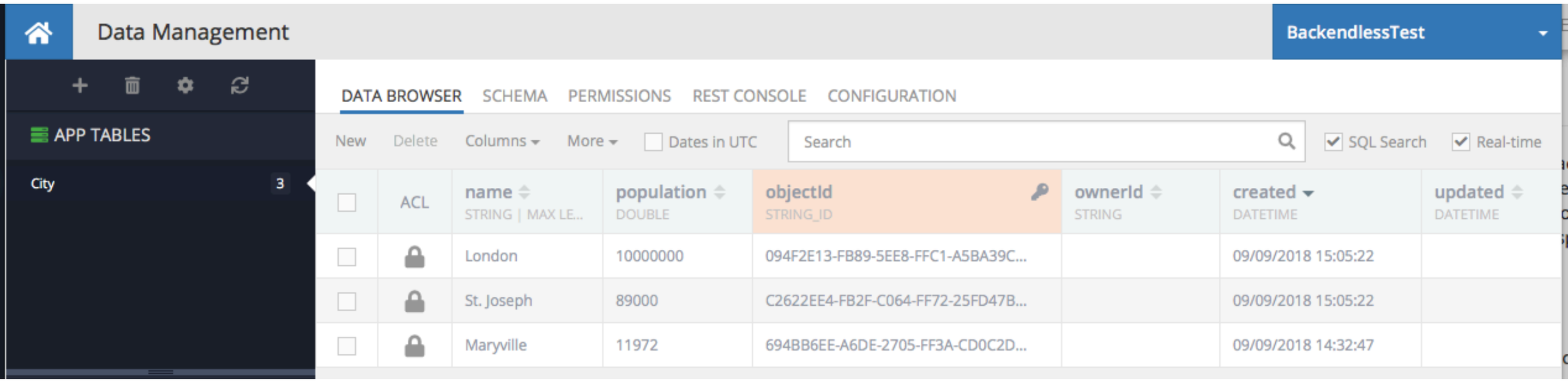
Q: Why are we overriding description?

```
@objcMembers

class City : NSObject {

    var name:String?
    var population:Int
    var touristSites:[TouristSite] = []
    override var description: String { // NSObject adheres to CustomStringConvertible
        return "Name: \(name ?? ""), Population: \(population), ObjectId: \(objectId ?? "N/A")"
    }
    var objectId:String?

    static let impossiblePopulation = -1

    init(name:String?, population:Int?, touristSites:[TouristSite]){
        self.name = name
        self.population = population ?? City.impossiblePopulation
        self.touristSites = touristSites
        //self.objectId = ""
    }

    convenience override init(){
        self.init(name:"", population:City.impossiblePopulation, touristSites:[])
    }

}
```

Another Class that Follows the Rules

```
@objcMembers
class TouristSite : NSObject {

    var name:String?
    var admissionFee:Double
    override var description: String { // NSObject adheres to CustomStringConvertible
        return "Name: \(name ?? ""), Admission Fee: \(admissionFee)"
    }
    var objectId:String?

    static let impossiblePopulation = -1
    static let impossibleAdmissionFee = -1.00

    init(name:String?, admissionFee:Double?){
        self.name = name
        self.admissionFee = admissionFee ?? TouristSite.impossibleAdmissionFee
    }

    convenience override init(){
        self.init(name:"", admissionFee:TouristSite.impossibleAdmissionFee)
    }

}
```





Apart from City properties **name**, **population** and **touristSites**, Be automatically creates 4 others. To access them in our code, define these properties in the class:

```
var objectId:String?
var created:NSDate?
var updated:NSDate?
```

Do not assign values to these in your code, Be does this. If you forget, you'll have to figure out what FAULT = '1000' [Entity with the specified ID cannot be found] means 😱

Fun fact! Data tables are completely editable! You can add/delete rows and edit values within the Be app. Use this to get some starter data in an app, or just for testing.

The Data Service API

- The Data Service API is used for **saving**, **updating**, **retrieving**, and **deleting** objects. All Backendless APIs use a singleton object, so we must first retrieve that, and from it get a datastore.

- A **datastore**, in Backendless' parlance, is an in-iOS-app representation of the Be app's data table

- It adheres to the **IDataStore** protocol.

Getting the Backendless singleton and DataStore

```
let backendless = Backendless.sharedInstance()! // singleton (of type Backendless)
let cityDataStore = backendless.data.of(City.self) // connection to City data table
```

- Recommendation: define these in a model class, and write methods in the model class that handle creating, saving, updating and deleting

- In these notes, for clarity, we may define these where we need them, but our sample app will follow best practices

Saving Objects Synchronously (Blocking)

```
let city = City(name: "Toronto", population: 1352822, touristSites: [])
let savedCity = cityDataStore.save(city) as! City        // save city & get a copy with
                                                         // objectId filled in
```

- save() saves the city synchronously: execution will not resume until it has finished
- If the network is slow, the app may appear unresponsive.
- save() returns a City object that is exactly the same as the one passed in except its objectId property is filled in.
- This becomes relevant when creating 1:1 and 1:n relationships

Saving Objects Asynchronously (Non-Blocking)

```
dataStore.save(city,
               response: { (result:Any?) -> Void in
                        let citySaved = result as! City
                        print("We saved \(citySaved)")
                         },
              error: { (fault:Fault?) -> Void in
                         print("Problem saving City: \(fault!)")
                        })
```

- This version of save() is asynchronous: execution will continue, and after the app finishes, either the response or error closure will be called (in a different thread)
- If successful, result will be a City object with the same fields as the one saved, but with the objectId filled in

Error Handling

- Be uses its own Types class, with the type method **tryblock(_:, catchblock:)**.

- It takes 2 closures containing the code you wish to execute, the other the error handling code.

An Error Handling Example

```
Types.tryblock({() -> Void in
        for city in self.cities {
            let savedCity = self.cityDataStore.save(city) as! City
        }
    },
        catchblock:{ (exception) -> Void in
            print(exception.debugDescription)
        }
)
```

Try this

Catch errors

*in a parameterless, Void closure, **`() -> Void in`** can be eliminated

Updating Objects

- Updating an object is simple:

1. **Retrieve it** (with find(), or one of its variants)

2. **Change it** (just don't mess with the objectId!)

3. **Save it** (using the same Save APIs as discussed previously)

Deleting Objects Synchronously

```
let dataStore = backendless.data.of(YOUR-CLASS.self)

// Removes an existing object from Backendless database
let num = dataStore.remove(entity: Any!)

// Removes an object identified by its objectId
let num = dataStore.removeById(objectId: String!)
```

Deleting Objects Asynchronously

```
let dataStore = backendless.data.of(YOUR-CLASS.self)

// Removes an object in the Backendless database
// Server's response (result or error) is delivered through
// closure-based callbacks
dataStore.remove(entity: Any!,
                 response: ((NSNumber?) -> Void)!,
                 error: ((Fault?) -> Void)!)

// Removes an object identified by its objectId
dataStore.remove(byId: String!,
                 response: ((NSNumber?) -> Void)!,
                 error: ((Fault?) -> Void)!)
```

Finding Objects

- Backendless provides methods to:

  1.retrieve all objects from a datastore - **find()**

  2.retrieve an object by id - **findById(objectId:)**

  3.retrieve first & last objects - **findFirst()**, **findLast()**

  4.retrieve objects that meet a criteria - **find(queryBuilder:)**

- 1 and 4 return an array; 2 and 3 return one object. Use as! to downcast to the correct class

Example: Finding Objects Synchronously

```
var allCities:[City] = []
var firstCity:City!
var lastCity:City!
var bigCities:[City] = []
```

```
Types.tryblock({() -> Void in

      allCities = self.cityDataStore.find() as! [City]
      firstCity = self.cityDataStore.findFirst() as! City
      lastCity = self.cityDataStore.findLast() as! City
      let qb = DataQueryBuilder()
      qb!.setWhereClause("population > 50000")
      bigCities = self.cityDataStore.find(qb) as! [City]

}, catchblock: {(exception) -> Void in

    print(exception.debugDescription)

})
```

Example: Finding Objects Asynchronously

```
self.cityDataStore.find({(allCities) -> Void in
    print("All Cities: ", allCities as! [City])
}, error: {(exception) -> Void in
    print(exception.debugDescription)
})
```

2 closures, the first unlabeled, the second labeled as error:

```
self.cityDataStore.findFirst({(firstCity) -> Void in
    print("First City", firstCity as! City)
}, error: {(exception) -> Void in
    print(exception.debugDescription)
})

let qb = DataQueryBuilder()
qb?.setWhereClause("name LIKE '%ville'") // names ending in ville
qb?.setSortBy(["name", "population"]).   // sorted by name & pop'n

self.cityDataStore.find(qb, response: {(cities) -> Void in
    print("Cities ending in ville", cities as! [City])
}, error: {(exception) -> Void in
    print(exception.debugDescription)
})
```

Relations

- Backendless supports **relations**, references from an object in one table to an object (or objects) in another
- Relations may be **1:1** or **1:n**
- Relations may be created using the console or via API
- A **parent** is an object that contains a reference to another object, its **child**.
- Backendless distinguishes between
  - **setting** a relation, in which the object to be related to the parent **replaces** any previous child(ren)
  - **adding** a relation, in which the object to be related to the parent is **added** to the collection of existing children. This generates an error if the parent-child relation is 1:1

Relations

In the **City** datastore, when establishing the City to TouristSite relation, the City is the parent, a TouristSite is the child. We use **addRelation()**, as this is a to-many relation (1 City, many TouristSites)

In the **TouristSite** datastore, when making the TouristSite to City relation, the TouristSite is the parent, the City is the child. We could use **setRelation()**, as this is an n:1 relation (1 TouristSite, 1 City)

- Each City can have many TouristSites
- Each TouristSite belongs to exactly 1 City

The City & TouristSite Tables

APP TABLES

City 1

TouristSite 6

DATA BROWSER | SCHEMA | PERMISSIONS | REST CONSOLE | CONFIGURATION

New | Delete | Columns | More | Dates in UTC | name = 'Paris'

ACL	name STRING | MAX LENG...	population DOUBLE	touristSites RELATION TO: TouristSite autoload	objectId STRING_ID
🔒	Paris	2206488	1:N Relations	E8B48815-1BF6-E9C3-FFA2-C384672...

APP TABLES

City 4

TouristSite 6

DATA BROWSER | SCHEMA | PERMISSIONS | REST CONSOLE | CONFIGURATION

New | Delete | Columns | More | Dates in UTC | Search | SQL Search | Real-time

ACL	admissionFee DOUBLE	name STRING | MAX LENGTH...	objectId STRING_ID	ownerId STRING	RELATION FROM: City[touristSites]
🔒	25	Title Town Fitness	87AE30AB-DC88-D...		Click to go to the parent object(s)
🔒	15000	Northwest	99232F4A-067C-0...		Click to go to the parent object(s)
🔒	18	Loeuvre	016DE09B-B56F-0...		Click to go to the parent object(s)
🔒	25	Notre Dame	9F6B7CBA-089E-1...		Click to go to the parent object(s)
🔒	25	Big Eye	A7BB79BA-749B-8...		Click to go to the parent object(s)
🔒	15	Big Ben	E7776894-3B56-A...		Click to go to the parent object(s)

Adding Relations

- **addRelation()** is used to build a 1:n relation, so child objects will be *added* to the parent's collection*
- addRelation() builds *both* sides of the relation, so, for instance, if have a City and add a TouristSite to it, 2 entries will be created — in the City, referencing the TouristSite, and in the TouristSite, referencing its City

*__setRelation()__ will replace the parent's collection, but addRelation() will suffice for our purposes

Adding Relations Synchronously

```
let dataStore = backendless.data.of(YOUR-CLASS.self)
let result = dataStore.addRelation(columnName: String!,
                                   parentObjectId: String!,
                                   childObjects: [String]!) as! NSNumber
```

columnName
- the name of the column in the datastore specifying the relation. If the column does not exist in the table when this is invoked, it must also include the name of the child table, separated by a colon, and the cardinality (e.g., "touristSites:TouristSite:n")

parentObjectId
- the objectId of the parent object

childObjects
- a collection of the children's objectIds (not the entire object, just the objectId (a String): *your instructor learned this the hard way* 🆘)

Return Value
- the number of child objects set into the relation

Adding Relations Asynchronously

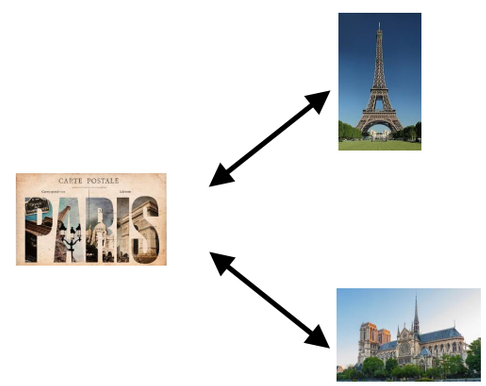
```
let dataStore = backendless.data.of(YOUR-CLASS.ofClass())
dataStore?.addRelation(columnName: String!,
                       parentObjectId: String!,
                       childObjects: [String]!,
                       response: ((NSNumber?) -> Void)!,
                       error: ((Fault?) -> Void)!)
```

```
columnName
- the name of the column specifying the relation. If the column does not
exist in the table when this is invoked, it must also include the name of
the child table, separated by a colon, and the cardinality (e.g.,
"movie:Movies:n")
parentObjectId
- the objectId of the parent object
childObjects
- a collection of the children's objectIds (not the entire object, just the
  objectId (a String)
responseBlock
- a closure that is passed the number of children set into the relation
errorBlock
- print out the Fault to see what's gone wrong (it will be nil if successful)
```

Building 1:n Relations

addRelation() will
1. create a field in the City datastore, **touristSites**, pointing to the TouristSite datastore (if that field doesn't already exist)
2. make a reference from Paris to both eiffel and notreDame TouristSites
3. make a connection from each TouristSite back to Paris.

That's a lot of work in one little statement! 😍



the **parent** datastore

a field to create in the parent datastore

name of the child datastore (TouristSite)

Assume **paris** has 2 TouristSites, **eiffel, notreDame**. In the City table, we need a field, touristSites.

# of children per parent (1 or n)

```
cityDataStore.addRelation("touristSites:TouristSite:n",
                          parentObject:paris,
                          childObjectIds:[eiffel.objectId, notreDame.objectId)
```

Retrieving Related Objects

- When you retrieve an object, its related objects do not come with it automatically

- For instance, if you fetch a City, you do not get its TouristSites automatically

- Use **setRelated()** to specify the related objects to retrieve

Retrieving Related Objects: An Example

```
let queryBuilder = DataQueryBuilder()
queryBuilder!.setRelated(["touristSites"]) // TouristSites referenced in City's touristSites
                                           // field will be retrieved for each City
queryBuilder!.setPageSize(100) // up to 100 TouristSites can be retrieved for each City
cityDataStore.find(queryBuilder, response: {(results) -> Void in    //
    let cities = results as! [City]
    for city:City in cities {
        print("\(city.name!) has \(city.touristSites.count) tourist sites")
        for touristSite in city.touristSites {
            print(touristSite)
        }
    }

}, error: {(exception) -> Void in
    print(exception.debugDescription)
})
```

- setRelated(["touristSites") means that when we fetch Cities, their TouristSites (objects related via the touristSites field) will be retrieved as well

- We can use any variant of find(), or use a where clause to limit the Cities retrieved: but whatever cities are retrieved, their TouristSites will be as well

Retrieving Related Objects

- The previous example showed, in a 1:n relation, how we could find all the TouristSites related to each City.

- What if we wanted to do the opposite, i.e., given a TouristSite, find its City? This example shows how.

```
// first, find a TouristSite (e.g., Royal Ontario Museum)
var royalOntarioMuseum:TouristSite!
let queryBuilder = DataQueryBuilder()
queryBuilder?.setWhereClause("name = 'Royal Ontario Museum'")
royalOntarioMuseum = touristSitesDataStore.find(queryBuilder)[0] as! TouristSite

// Now, find the City associated with it, i.e., the one who's touristSites.objectId matches the Royal
Ontario Museum. This iterates through all the touristSites of all the retrieved Cities (in this case all
of them) until it's found the right City -- Toronto

queryBuilder!.setPageSize(100) // up to 100 TouristSites can be retrieved for each City
queryBuilder!.setWhereClause("touristSites.objectId = '\(royalOntarioMuseum.objectId!)'")
cityDataStore.find(queryBuilder, response: {(cities) -> Void in    //
    let city = cities![0] as! City
    print("The city corresponding to this tourist site, the Royal Ontario Museum, is \(city.name!)")
}, error: {(exception) -> Void in
    print(exception.debugDescription)
})
```

The Full City Tourist Site App

- See the full City Tourist Site app for a demonstration of how all this works in practice.

- Have it open while reading the next few slides ...

A Matter of Timing

- If a table view controller needs data and is willing to wait, it can do so synchronously ...

```
   override func viewWillAppear(_ animated: Bool) {
         touristBureau.reloadAllCities()
         tableView.reloadData()
      }


// fetches all cities and their tourist sites, storing results in cities
// this method is in the TouristBureau class (see the app)
   func reloadAllCities() {
       let queryBuilder = DataQueryBuilder()
       queryBuilder!.setRelated(["touristSites"])
       queryBuilder!.setPageSize(100)
      Types.tryblock({() -> Void in self.cities = self.cityDataStore.find(queryBuilder) as! [City]},
       catchblock: {(fault) -> Void in print(fault ?? "Something has gone wrong  reloadingAllCities()")})
  }
```

touristBureau represents the model

A Matter of Timing

- ... or asynchronously. But in the latter case there is a problem: reloadData() will be called *before* the data has actually been retrieved

```
override func viewWillAppear(_ animated: Bool) {
      touristBureau.reloadAllCitiesAsynchronously()
      tableView.reloadData() // tableView(_:cellForRowAt indexPath:) uses self.cities
   }


// fetch all cities and their tourist sites asynchronously, storing results in cities
 func reloadAllCitiesAsynchronously() {
     let queryBuilder = DataQueryBuilder()
     queryBuilder!.setRelated(["touristSites"])

     queryBuilder!.setPageSize(100)
     cityDataStore.find(queryBuilder, response: {(results) -> Void in
         self.cities = results as! [City]
    }, error: {(exception) -> Void in
         print(exception.debugDescription)
     })

 }
```

A Matter of Timing

- The solution is for the model to post a notification when the cities have been retrieved, and for the TVC to then react to it.

In the view controller

```
override func viewDidLoad() {
      super.viewDidLoad()
      touristBureau = TouristBureau.sharedInstance
       NotificationCenter.default.addObserver(self, selector: #selector(citiesReloaded), name: .CitiesReloaded, object: nil)
}

override func viewWillAppear(_ animated: Bool) {
      touristBureau.reloadAllCitiesAsynchronously()
      tableView.reloadData()
   }

@objc func citiesReloaded(){
     tableView.reloadData()
 }
```

In the model

```
// fetch all cities and their tourist sites asychronously, storing results in cities
 func reloadAllCitiesAsynchronously() {
     let queryBuilder = DataQueryBuilder()
     queryBuilder!.setRelated(["touristSites"])

     queryBuilder!.setPageSize(100)
     cityDataStore.find(queryBuilder, response: {(results) -> Void in
         self.cities = results as! [City]
         NotificationCenter.default.post(name: .CitiesReloaded, object: nil)
     }, error: {(exception) -> Void in
         print(exception.debugDescription)
     })
 }
```

Techy Aside: Saving Objects Asynchronously - a Better Way with GCD 😍

- We have seen how we can perform asynchronous operations using the asynchronous version of Backendless's methods

- However, these involve two closures, and the code is somewhat difficult to read

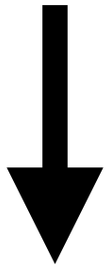
- An alternative is to use Apple's Grand Central Dispatch (GCD) API to run methods in different threads

- GCD can be used for any purpose, not just Backendless

Techy Aside: Saving Objects Asynchronously - a Better Way with GCD 😍

Synchronous

```
Types.tryblock({

        let city = City(name: "Toronto", population: 1352822, touristSites: []) // make a City
        let savedCity = self.cityDataStore.save(city) as! City // save city, and get a copy with objectId filled in
        let touristSite:TouristSite = TouristSite(name: "Royal Ontario Museum", admissionFee: 25.00)
        let savedTouristSite = self.touristSiteDataStore.save(touristSite) as! TouristSite
        self.cityDataStore.addRelation("touristSites:TouristSite:n", parentObjectId: savedCity.objectId, childObjects: [savedTouristSite.objectId!])

   }, catchblock: {(exception)->Void in
            print("😱 Problem when saving: \(exception.debugDescription)")
      })
```



Asynchronous

```
DispatchQueue(label: "hello").async {
     Types.tryblock({

          let city = City(name: "Toronto", population: 1352822, touristSites: []) // make a City
          let savedCity = self.cityDataStore.save(city) as! City // save city, and get a copy with objectId filled in
          let touristSite:TouristSite = TouristSite(name: "Royal Ontario Museum", admissionFee: 25.00)
          let savedTouristSite = self.touristSiteDataStore.save(touristSite) as! TouristSite
          self.cityDataStore.addRelation("touristSites:TouristSite:n", parentObjectId: savedCity.objectId, childObjects: [savedTouristSite.objectId!])

        }, catchblock: {(exception)->Void in
              print("😱 Problem when saving: \(exception.debugDescription)")
           })

}
```

Techy Aside: Dispatch Queues

1. The Dispatch framework allows concurrent execution of code
2. It uses FIFO queues, **DispatchQueues**, to which you can submit closures, known as work items
3. Work items can execute
    1. serially -- one after another -- or
    2. concurrently -- items are dequeued in order, but then can execute at the same time (and the completion order depends on how long each work item takes
4. DispatchQueues have a label and a Quality of Service quantifier, which indicates how quickly the work gets done
5. iOS comes with several global DispatchQueues already to go
6. iOS apps always run on the **main** DispatchQueue

Resources

- backendless.com