CS 47

Beginning iPhone Application Development

Week 5: Data Retrieval and Storage

Office Hours

- Cancelled this weekend
- Please email or send questions to the discussion forum

Quick MVC Review

- Data retrieval and storage is focused mainly in the model component of your application
- Remember: The model should be as reusable as possible
- Think layers of abstraction

Connectivity

- The iPhone provides an IP stack over the 3G or Wifi connection.
- This connection should be considered unreliable. If you have a choice, focus on a fast request-response communication model

Sockets

- Yes, you can use all of the standard socket programming calls: socket(), bind(), accept (), connect(), read(), write(), etc.
- Or one level higher: CFSocket API

Sockets

- Persistent sockets have their uses
- Good for games, highly interactive apps that constantly receive information from a server (streaming audio, video, etc)
- But unlike the desktop environment, you will need more code to gracefully handle socket disconnects

Request-Response

- Most iPhone apps will only need request-response style data retrieval
- Think: web browser
  - 99.99%* of web apps use the request-response model

*Not scientifically determined

Request-Response

- So put yourself in the mindset of making a web application that happens to render natively on the phone
- We already have a widely-used protocol for request-response communication: HTTP

Request-Response

- Any time we want data, or want to notify the server of something, we are going to fire off an HTTP request
- This can be wrapped in an Objective-C handler class, and integrates nicely into your MVC architecture
- Happy Birthday: I'm giving you mine
- But we should still have a basic understanding of the internals. Let's start with the basics:

NSURL

- NSURL : NSObject
- Represents URLs as defined by RFC 1808, 1738 and 2732
- Generally you will just initialize them with a string, but several initialization methods are available

```
NSURL *myURL = [NSURL URLWithString:@"http://www.fieldman.org"];
```

NSURL

- What about GET URLs with weird characters?

```
http://www.fieldman.org?user_id=2&comment=Hello World
```

- Need to percent escape them

```
NSString *getURLString = ...;
NSURL *myURL = [getURLString
          stringByAddingPercentEscapesUsingEncoding:NSASCIIStringEncoding];
```

NSURLRequest

- NSURLRequest : NSObject
- (Generally you want a NSMutableURLRequest)

```
NSURL        *url     = [NSURL URLWithString:@"..."];
NSURLRequest *request = [NSURLRequest requestWithURL:myURL];
```

- Encapsulates the parameters of a request to a URL

NSURLRequest

- Things you can set in the NSMutableURLRequest class

`setCachePolicy:`	Should we use internal cache?
`setTimeoutInterval:`	How long before we give up
`setHTTPMethod:`	POST, GET or others
`setHTTPBody:`	Fill out the POST data
`setValue:forHTTPHeaderField:`	Set random HTTP headers

NSURLConnection

- So we have a request object representation, how do we execute it?
- NSURLConnection : NSObject
  - Handles the very basic, low-level HTTP socket connection
  - Reports back to its delegate when protocol events occur

Sync vs. Async

- Synchronous vs. Asynchronous relates to the blocking nature of the request
- A synchronous call blocks execution of the current thread until the request is complete

```
+ sendSynchronousRequest:returningResponse:error:
```

- That seems pretty simple

Synchronous

- Problem:
- Most of your code will be in the main thread. What happens if you make a synchronous URL request from the main thread?
- All UI handling is blocked until the request returns

Synchronous

- Solution:
- Call sendSynchronousRequest from another thread

```
- (void) someMainThreadFunc {
  NSMutableURLRequest *request = ...;
  [self performSelectorInBackground:@selector(sendReq:) withObject:request];
}

- (void) sendReq:(NSURLRequest*)req {
  NSData *retData = [NSURLConnection sendSynchronousRequest:req returningResponse:nil error:nil];
  [self performSelectorOnMainThread:@selector(receivedData:) withObject:retData waitUntilDone:NO];
}

- (void) receivedData:(NSData*)data {
  /* Do something with data */
}
```

Synchronous

- This is fairly simple to implement and is good for quick prototyping, but it's not very reusable
- Also, you don't have any visibility into the request while it's processing (e.g. no %-complete indicators)
- Can't be cancelled
- What if we're getting a massive file?

Asynchronous

- So let's use asynchronous instead - a bit more complicated, but more flexible and in-line with the protocol-delegate pattern
- Asynchronous calls are nonblocking and all interact in the main thread

Asynchronous

- Create the connection object; it needs to exist for the lifetime of the connection

```
NSURLRequest    *request    = ...; /* Fully configured request object */
NSURLConnection *connection = [[NSURLConnection alloc] initWithRequest:request
                                                              delegate:self];
```

- The connection begins immediately when the object is instantiated

Asynchronous

- The asynchronous model does not hand us back a neatly packaged NSData object
- Rather, the delegate is notified when a chunk of data is read from the stream
- It is the job of the delegate to stream the incoming data chunk to the proper location (memory, disk, audio, etc)

Asynchronous

- Let's take the example of storing to disk in our asynchronous delegate methods

```
- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse *)response {
  /* We can assume this is an NSHTTPURLResponse subclass */
  if ( [((NSHTTPURLResponse*)response) statusCode] == 200 ) {
    fileHandle         = [[NSFileHandle fileHandleForUpdatingAtPath:downloadToFilePath] retain];
    currentlyReceived = 0;
    expectedLength     = [response expectedContentLength];
  }
}

- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {
  [fileHandle writeData:data];
  currentlyReceived += [data length];
  [someDelegate connectionHasReceived:currentlyReceived of:expectedLength];
}
```

Asynchronous

- When the connection is complete, our delegate will either get a success or failure

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection;
- (void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error;
```

- We can use these methods to notify a higher-level delegate of completion

Asynchronous

- Regardless of using synchronous or asynchronous, we are getting data
- That data can be anything: Images, audio, JSON, XML, text, etc - it's up to you to decide what to do with that data

Data Storage

- So what do we do with that data?
- Display it (text, images, etc)
- Modify it and upload it
- Save to disk

Data Storage

- Your application's storage quota is only limited by the size of the devices drive
- However you are sand-boxed: You only have permission to read/write to certain directories
- You need to programmatically determine which paths these are

Data Storage

- NSSearchPathForDirectoriesInDomains

```
/* Path enumerations
NSDocumentDirectory   (backed up)
NSCachesDirectory     (not backed up) */

NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                     NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];

/* Creating a subdirectory */
NSFileManager *fileManager = [NSFileManager defaultManager];
NSString *subdir = [NSString stringWithFormat:@"%@/mysubdir", documentsDirectory];

[fileManager createDirectoryAtPath:subdir
       withIntermediateDirectories:YES
                        attributes:nil
                             error:nil];
```

NSFileHandle

- We've already seen one method: NSFileHandle

```
NSFileHandle *fileHandle = [NSFileHandle fileHandleForUpdatingAtPath:path];
```

- Allows standard operations like read/write in an Objective-C framework

```
- availableData
- readDataToEndOfFile
- readDataOfLength:
- writeData:
```

- Just a wrapper for C file descriptors (open/read/write)

NSFileHandle

- Has interesting methods to read and write data in the background
  - `readInBackgroundAndNotify`
  - `readToEndOfFileInBackgroundAndNotify`
- Uses the notification API which we will discuss in a future class

Other Alternatives

- Worth mentioning:
  - SQLite (compact SQL database API)
  - Core Data
- You may need these for random access into very large data sets, but it's usually overkill for most iPhone applications

Archiving

- A simple, robust, data-oriented approach is to serialize/archive generic data structures

```
NSString
NSNumber
NSData
NSDate
NSNull
NSArray
NSDictionary
```

- Or anything else that adheres to the NSCoding protocol

Archiving

- When archiving a container (NSArray, NSDictionary), the coder will automatically archive objects they contain
- You can create a multi-tiered archive (arrays inside dictionaries inside arrays, etc)

Archiving

- The top-level container classes (NSArray and NSDictionary) provide the methods

```
- writeToFile:atomically:
- writeToURL:atomically:
```

- Simply pass in a file path, and the entire container is archived and saved as a property list into that file path

Archiving

- Just as easy to extract that info back

```
xxxWithContentsOfFile:(NSString *)aPath
xxx = init, array, dictionary

NSArray *array = [NSArray arrayWithContentsOfFile:x];
NSDictionary *mDic = [[NSDictionary alloc] initWithContentsOfFile:y];
```

- Builds a container object with the property list at that file path

Archiving

- Very simple to code, very powerful as a means to store general data hierarchies
- Inefficient because it stores in a pseudo-XMLish format. We can do better with the binary property list generator

Archiving

```
/* Archiving */
NSData *propListData = [NSPropertyListSerialization
                        dataFromPropertyList:dictionary
                                      format:NSPropertyListBinaryFormat_v1_0
                            errorDescription:nil];
[propListData writeToFile:filePath atomically:YES];

/* Unarchiving */
NSData *propListData = [NSData dataWithContentsOfFile:filePath];
NSPropertyListFormat format;
NSMutableDictionary *dictionary = [NSPropertyListSerialization
                    propertyListFromData:propListData
                        mutabilityOption:kCFPropertyListMutableContainersAndLeaves
                                  format:&format
                        errorDescription:nil];
```

JSON

- Tying general communication and data storage together: JSON data transport (JavaScript Object Notation)

- JSON was designed to emulate the same general types and containers that most languages use (NSArray, NSDictionary, NSNumber, etc)

JSON

- Sample

```
{
  “key”  : “value”,
  “key2” : 10,
  “key3” : false,
  “key4” : null,
  “key5” : {
             “subkey1” : “value”,
             “subkey2” : “value2”
           },
  “key6” : [ 4, 5, 6, { “key” : “value” }, true ]
}
```

JSON

- A JSON library converts a JSON string into a native data container (array/dictionary)
- So you can now exchange generic types with a server

JSON

- So make your native model classes support transformation to and from generic data containers

- You can then use highly reusable code to either exchange those data containers with a server or disk

```
Server <-(JSON)->     Generic Container [ <-(Your code)-> Your Native Model ]
Disk   <-(archiver)-> Generic Container [ <-(Your code)-> Your Native Model ]
```