

# HTTP Networking in iOS

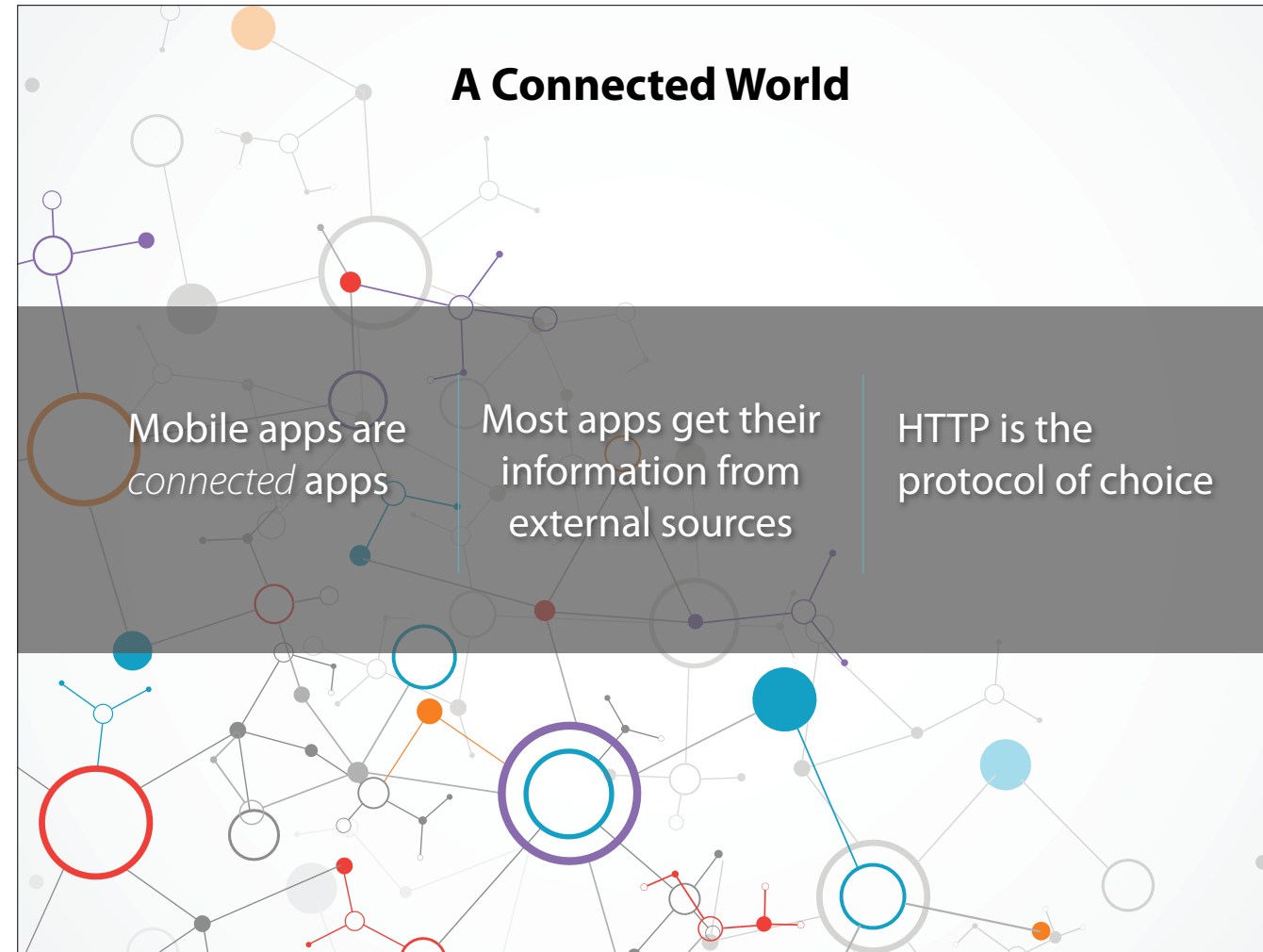
Connect your apps to the world

Alex Vollmer  
<http://alexvollmer.com>  
@alexvollmer



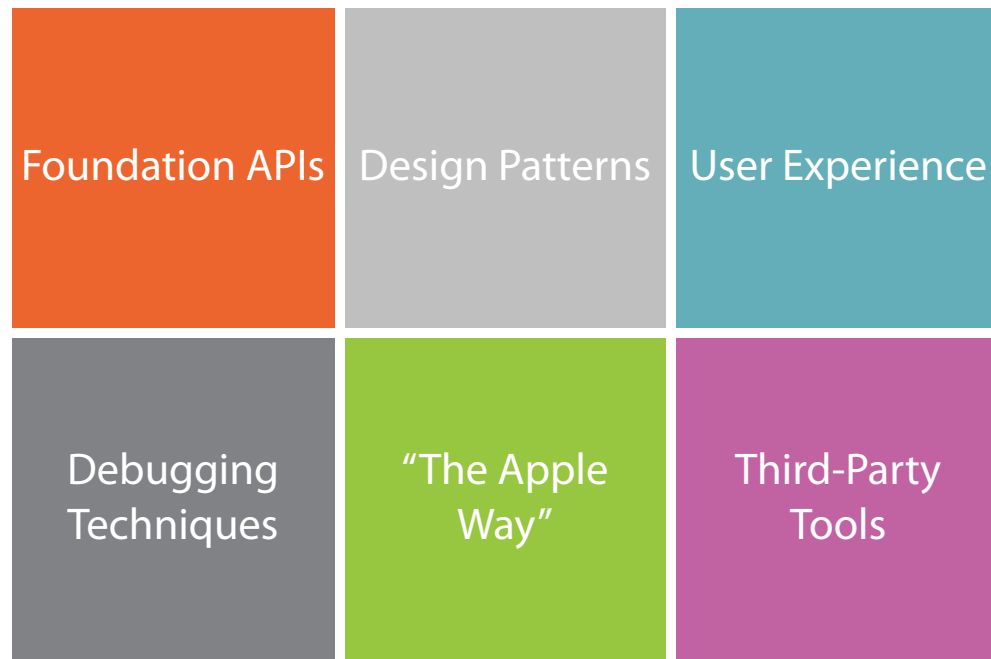
pluralsight  
hardcore dev and IT training

Welcome to “HTTP Networking in iOS”. My name is Alex Vollmer and in this course you’re going to learn how to integrate your iOS apps with remote HTTP services and build great user-experiences along the way.



We live in a connected world. Almost by definition, mobile apps are connected apps. Mobile apps typically act as portals to larger repositories of information. They can't store all of it locally, so they need to access these resources over the network. For these applications, HTTP is typically the protocol of choice.

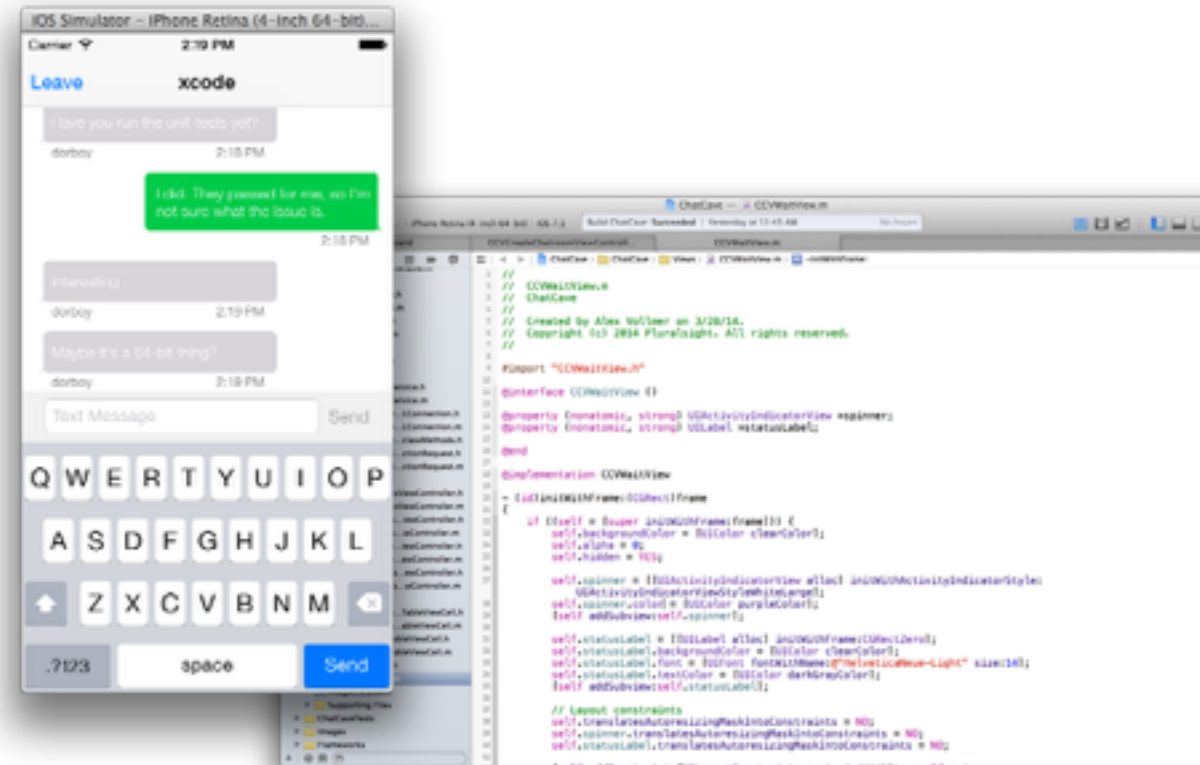
## Course Highlights



In this course, we we’re going to cover several topics:

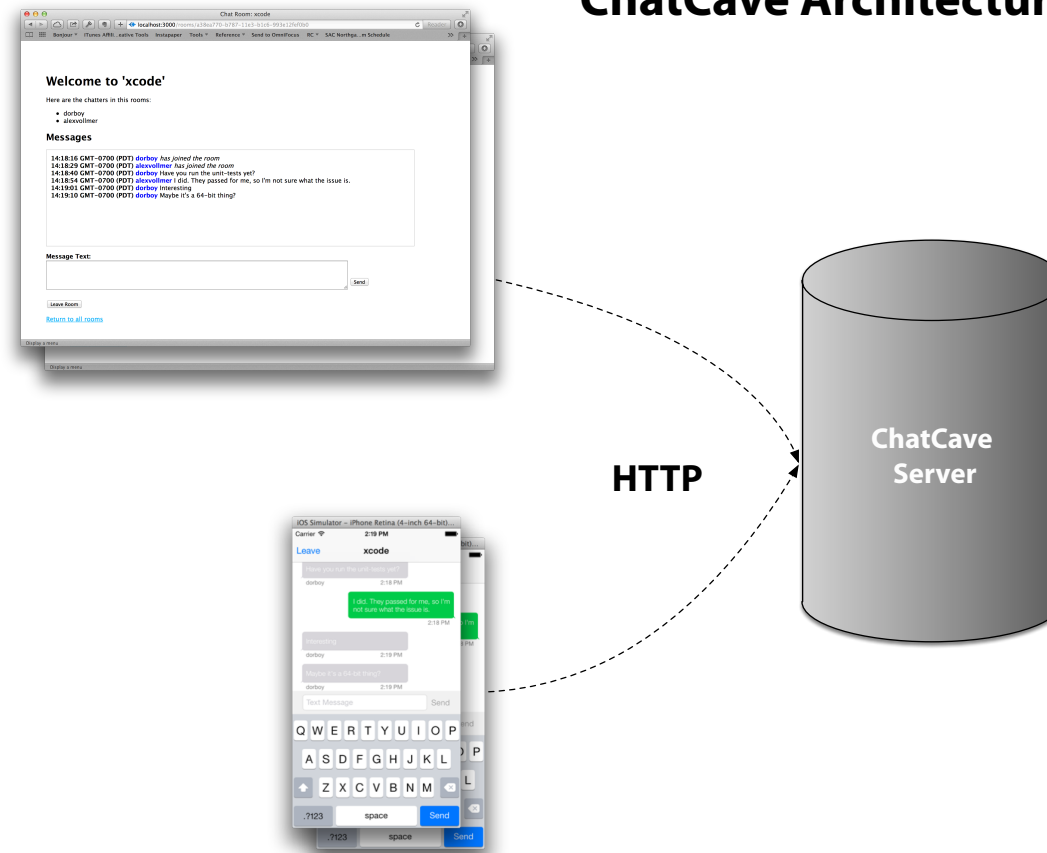
Networking is such a big topic that there are several things we won’t cover in here such as low-level socket connectivity, Mobile Safari and UIWebView integration, OAuth & third-party integration or Web Sockets.

# The Approach



To cover these topics, we'll be building a sample iOS app throughout the course. The application is a real-time chat messaging app we'll call "ChatCave"

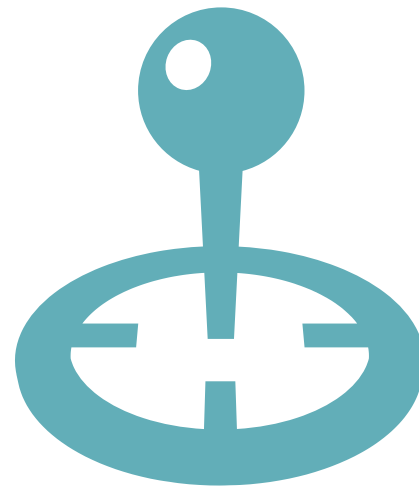
# ChatCave Architecture



The ChatCave architecture is pretty straight-forward. The ChatCave server is relatively simple HTTP server written in Node.js. Don't worry, you won't have to code in JavaScript or learn Node in order to use this. Later I'll show you the simple process of installing and running this server .

It supports browses via a built-in web app and native clients with a JSON-based REST API.

## The Roadmap



**Establish Goals & Principles**

**Review REST API**

**Review iOS App Design**

**Design Remote Access API**

**Implement Networking**

Before we dive into the app, let's look at the roadmap for this course:

## Golden Rules



**The main thread is for the user**

**UI can only be manipulated on main thread**

**Networking must be asynchronous**

**Do not directly create threads**

**Don't block the user**

There are several core design principles we want to build our app with. These are good rules are applicable when building any iOS app.

**presentational**  
**tate**  
**REST**  
**transfer**

The ChatcaveServer is architected as REST-ful server. It uses HTTP to express this architecture. But what, exactly does “REST” mean?



## **REST Architectural Elements**

**The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.**

...we could try this definition from Dr. Roy Fielding's dissertation in which he coined the term...

...but that's a little wordy and abstract

## Key Elements of REST

Unlimited set of nouns

Limited set of verbs

URLs describe hierarchy

Multiple representations

Orthogonal



Let's come up with a more concrete definition of REST by describing its components.

REST operates in a world of unlimited nouns that describe your problem domain. Within that world, there is a very limited set of nouns you can apply to those verbs. This is different from architectures like SOAP which are very RPC-like and focus on verbs.

Your domain of nouns can be organized into a hierarchy using URLs.

REST separates the underlying content from a specific representation. This is expressed in HTTP as different content-types for example HTML vs. XML vs. JSON

In REST we like to keep separate concerns separate. Orthogonal dimensions like security, content-type, transfer-encoding, chunking and the like are all treated independently of elements like URL structure and HTTP verbs.

## Chatroom Resources



**GET /rooms**

**POST /rooms**

**GET /rooms/:id**

**DELETE /rooms/:id**

Chatrooms are rooted at the `"/rooms"` resource. Authenticated users can retrieve a list of rooms, create a new one, fetch the details of a room and delete a room. Notice that when we create a room we POST to the `"/rooms"` collection. The server will generate a unique URL for the room which the client wouldn't be able to determine.

## Chatroom Members



**GET** /rooms/:id/chatters

**PUT** /rooms/:id/chatters/:id

**DELETE** /rooms/:id/chatters/:id

Authenticated users can join a chatroom with a PUT. We use the PUT verb here because the client has all the information to determine the URL—it knows the the unique room ID as well as the user ID. Users leave a chatroom by deleting the sub-resource. For debugging a /chatters sub-resource is provided.

## Message Resources



**GET /rooms/:id/messages**

**POST /rooms/:id/messages**

To send a message to a chatroom, we POST a JSON body to the “messages” sub-resource in a chatroom. We use POST here because the client is adding to a collection of resources. We could provide the ability to retrieve a specific message, but in practice it isn’t necessary.

We can retrieve the list of messages for a chatroom with a GET. As an optimization the server supports a “since” query parameter which allows a client to poll for messages after a particular ID.

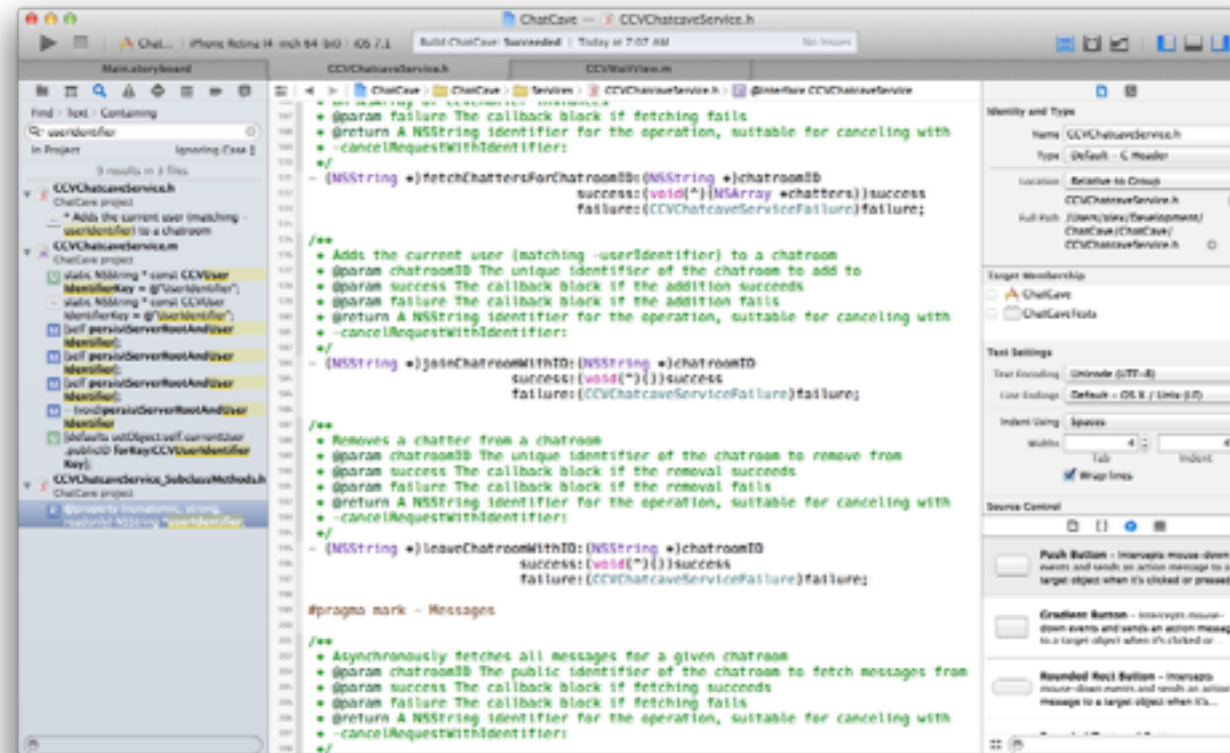
## Basic Authentication

```
GET /rooms HTTP/1.1
Host: localhost:3000
Connection: keep-alive
Accept: application/json
User-Agent: ChatCave/1.0 CFNetwork/672.1.13 Darwin/13.1.0
Accept-Language: en-us
Authorization: Basic YWxleHZvbGxtZXI6Zm9vYmFy
Accept-Encoding: gzip, deflate
```

The HTTP specification allows for a request to present authentication credentials to the server via the “Authorization” header. The value of this header can take a couple of forms, but the simplest is “Basic Authentication”.

Basic Authentication is not a secure way to pass credentials as it’s trivial for anyone to extract the username and password if they can intercept the message. However, if we deploy Basic Authentication over SSL, we can get rid of this security problem.

In ChatCave, we will use Basic Authentication. You can see that it’s just an HTTP header, but rather than manipulating headers directly, we will use a more elegant solution for managing credentials.



The ChatCave Xcode project is where we will be doing our work. Since our focus here is on the networking layer, I've stubbed-out a good portion of the application. Let's familiarize ourselves with the major components of the application.

# iOS App Design

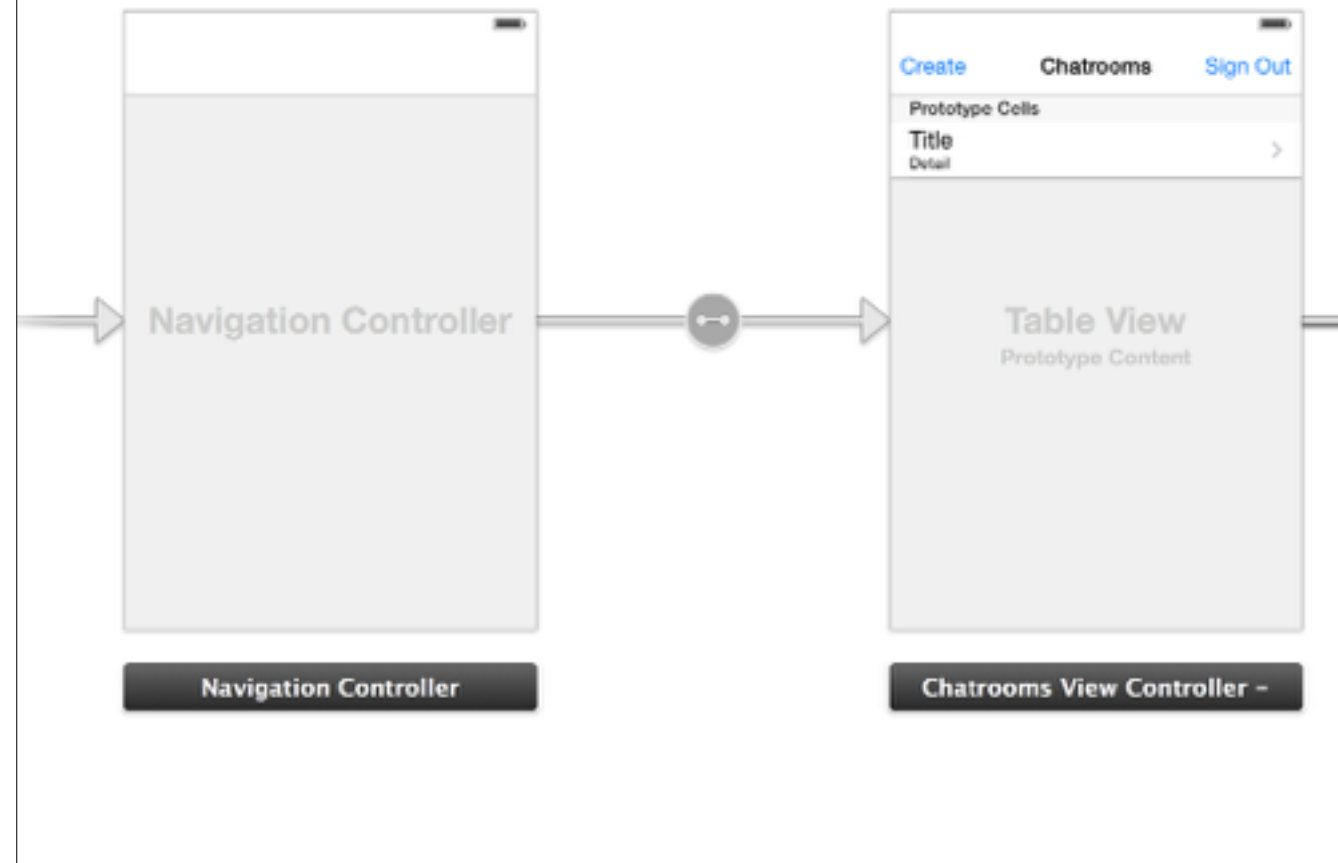


The design of the application is fairly simple. The application launches by displaying a list of available chatrooms. From here the user can join an existing room, or create a new one and join it. Once they are within a chatroom they can send and receive messages.

We'll treat authentication as an exceptional circumstance and design the user experience assuming the user is signed-in. If they aren't we will put them into a brief modal flow where they can sign-in or create a new account.



## CCVChatroomsViewController



Our root view-controller will display the list of chatrooms. Our class prefix is "CCV" so we'll call this class the CCVChatroomsViewController. Note that it is contained with a UINavigationController which will maintain the navigational stack for the user.

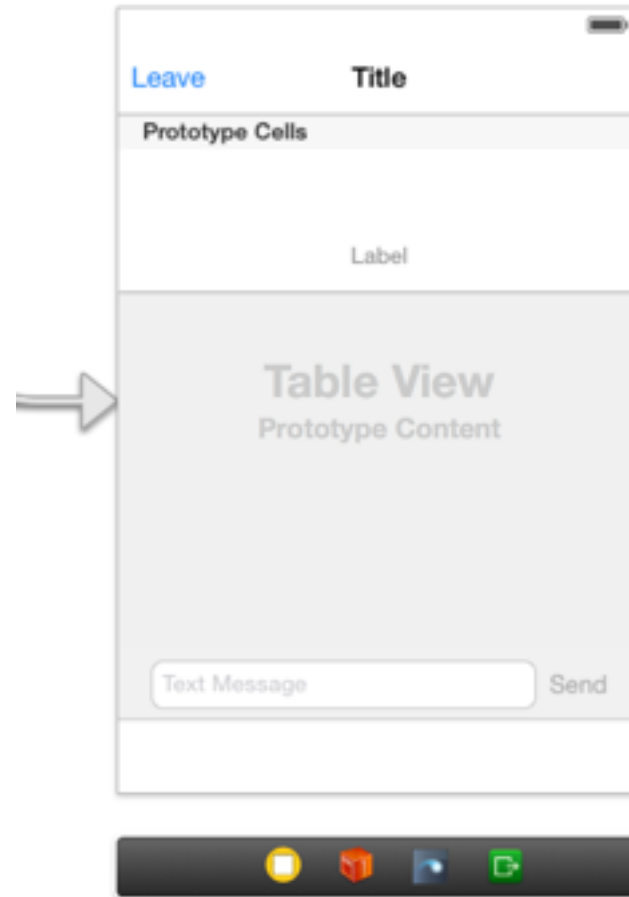
If the user taps the "Create" button...

## CCVCreateChatroomViewController



...we push a new view-controller onto the navigational stack. The user enters a room-name and taps the "Create Button" or cancels simply by navigating back on the stack. This is the CCVCreateChatroomViewController.

## CCVChatroomMessagesViewController



Once a user joins a chatroom we push another view controller onto the stack that displays the messages in the chatroom using a design similar to the built-in Messages app. This screen will be managed by an instance of the CCVChatroomMessagesViewController.

## CCVAuthenticationViewController

Join us!

User name

Password

Server URL

Sign In

Sign Up

Authentication View Controller

Once a user has authenticated, we want our app to remember those credentials and automatically authenticate the user to the system. However, we want to handle when the user first uses the app or credentials change on the server. So we will treat authentication a special “navigational loop”.

# Model Classes

CCVChatter
- (instancetype)initWithPublicID:(NSString *)publicID name:(NSString *)name; - (instancetype)initWithDictionary:(NSDictionary *)dict; - (NSDictionary *)dictionaryRepresentation;

CCVChatroom
- (instancetype)initWithPublicID:(NSString *)publicID name:(NSString *)name chatters:(NSArray *)chatters; - (instancetype)initWithDictionary:(NSDictionary *)dictionary; - (NSDictionary *)dictionaryRepresentation;

CCVMessage
- (instancetype)initWithPublicID:(NSString *)publicID type:(CCVMessageType)type text:(NSString *)text author:(NSString *)author timestamp:(NSDate *)date; - (instancetype)initWithDictionary:(NSDictionary *)dictionary; - (NSDictionary *)dictionaryRepresentation;

There are three classes in our model layer: CCVChatter, CCVChatroom and CCVMessage. By themselves, these classes don’t have a lot of behavior, but they do encapsulate the data passed between client and server.

# Object Serialization

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

@protocol CCVSerializable <NSObject>

/**
 * Initialize a new instance based on the properties and structure
 * of the given dictionary
 */
- (instancetype)initWithDictionary:(NSDictionary *)dict;

/**
 * Return a dictionary representing the data and structure of this
 * object. This is effectively the inverse of -initWithDictionary
 */
- (NSDictionary *)dictionaryRepresentation;

@end
```

Each of the model classes implements the CCVSerializable protocol which defines two method: -initWithDictionary: and -dictionaryRepresentation.

I find that it's helpful to have model classes be able to serialize to and from Foundation objects when dealing with remote network APIs. Later we'll see how we can use this protocol to handle JSON serialization and de-serialization.

## Design Goals

Block-based callbacks

Use domain-model objects

Abstract away details

Can be canceled



Now that you have an idea of how the iOS app is structured, let's start designing how we want to interface to the chat cave server. Before we dive into the interface design, let's establish a few goals...

## Encapsulating Remote Access

```
@interface CCVChatcaveService : NSObject
#pragma mark - Singleton access
+ (CCVChatcaveService *)sharedInstance;
// more to come...
@end
```

A strategy I find useful is to encapsulate access to the remote API in to one or more classes. In this application, we will abstract access to the REST API in the CCVChatcaveService. This class will make a single configured instance available to all classes in the application via single +sharedInstance method.



## Callbacks

```
typedef void (^CCVChatcaveServiceFailure)(NSError *error);
```



When a particular request for a resource succeeds, we want to return model objects when possible. Which means the arguments in our success-case callbacks will vary by request.

But in the cases of a failure, the callback block will look the same. Let's create a typedef for the callback block in the error case. This will keep our API consistent, help us avoid mistakes and cut down a bit on our typing.

Let's go ahead and map the various noun/verb combinations of our REST API to Objective-C methods in this class...

## Fetching Chatrooms

```
- (NSString *)fetchChatroomsSuccess:(void (^)(NSArray *chatrooms))success  
failure:(CCVChatcaveServiceFailure)failure;
```

## GET /rooms

The first method we'll create is -fetchChatroomsSuccess:failure:

Let's assume that the shared instance of the CCVChatcaveService is aware of user credentials and what server we're talking to. This first method issues a GET request on the /rooms resource. If the request succeeds, it invokes the give "success" block with an array of CCVChatroom instances. If the request fails it invokes the generic "error" block.

Notice how this API is very high-level and speaks in terms of our domain-model object. There's nothing here about network requests, HTTP details or dealing with JSON. The goal here is to design the ideal API that we want to work with in the rest of our code. By abstracting away all of the remote networking details into this class, we keep the rest of our code focused on the immediate domain problem.

## Creating A Chatroom

```
- (NSString *)createChatroomWithName:(NSString *)name
                                success:(void (^)(CCVChatroom *chatroom))success
                                failure:(CCVChatCaveServiceFailure)failure;
```

## POST /rooms

To create a room we POST a JSON body to the /rooms resource. We'll map this to our next method: -createChatroomWithName:success:failure:. The posted body only requires a name for the chatroom, so the first parameter to this method is a NSString instance for the new room name.

In the successful case, a block will be executed that is given a CCVChatroom instance. In the failure case the generic error block will be invoked.

## Joining A Chatroom

```
- (NSString *)joinChatroomWithID:(NSString *)chatroomId  
    success:(void (^)(void))success  
    failure:(CCVChatcaveServiceFailure)failure;
```

***PUT /rooms/:room\_id/chatters/:user\_id***

When a user joins a chatroom they issue a PUT request to a specific URL that requires the unique identifier of a chatroom as well as a user's unique identifier. Let's design the CCVChatcaveService so that it knows about the user identifier. When a user joins a chatroom we need to provide the unique ID of the given chatroom.

Our next method, -joinChatroomWithID:success:failure: only needs the unique identifier of the chatroom to join. In the successful case, the callback block is invoked without any arguments since there isn't any interesting data returned as a result of this call. In the failure case, the generic error callback block will be invoked.

## Leaving A Chatroom

```
- (NSString *)leaveChatroomWithID:(NSString *)chatroomId
    success:(void (^)(void))success
    failure:(CCVChatcaveServiceFailure)failure;
```

**DELETE /rooms/:room\_id/chatters/:user\_id**

Leaving a room is simply a matter of issuing a different HTTP verb to the same resource. Again, we'll assume that the CCVChatcaveService will keep track of the current user's identifier. The API then only needs the chatroom's unique identifier.

In the successful case a block with no arguments will be invoked and in the failure case the generic error callback will be invoked.

## Fetching Messages

```
- (NSString *)fetchMessagesForChatroom:(NSString *)chatroomId
    success:(void (^)(NSArray *messages))success
    failure:(CCVChatcaveServiceFailure)failure;
```

**GET /rooms/:room\_id/messages**

To fetch the messages for a room we need to issue a GET request against the rooms “messages” sub-resource. This resource will be polled by the iOS app to fetch new messages in the chat-room.

We’ll map this request to -fetchMessagesForChatroom:success:failure:. In the success case the callback block will receive an array of CCVMessage instances. In the failure case, the generic error callback block will be invoked.

To optimize this a bit, we need to distinguish between fetch all messages for a chatroom and fetching the newest messages in a chatroom....

## Fetching Newest Messages

```
- (NSString *)fetchMessagesForChatroom:(NSString *)chatroomId
    since:(NSString *)sentinelMessageID
    success:(void (^)(NSArray *messages))success
    failure:(CCVChatcaveServiceFailure)failure;
```

**GET /rooms/:room\_id/messages?since=xxx**

Once a user is in a room and we've retrieved all the messages at a point in time, we only need to fetch the latest messages. The "messages" sub-resource accepts a query-parameter named "since" which matches the unique identifier of the last message the client saw.

Once the iOS app has performed the initial fetch, it will use this method to retrieve just the latest updates.

## Posting a Message

```
- (NSString *)postMessageWithText:(NSString *)text
                        toChatroom:(NSString *)chatroomId
                        success:(void (^)(CCVMessage *message)) success
                        failure:(CCVChatcaveServiceFailure) failure;
```

### POST /rooms/:room\_id/messages

When a user sends a message to a room, we POST to a chatroom's "messages" child resource.


We'll map this to the -postMessageWithText:toChatroom:success:failure: method. This method takes a NSString instance for the text of the message, an enum type indicating the type of message and the unique identifier of the chatroom. The second argument is the unique identifier of the chatroom.

If the request succeeds, the callback block is given a CCVMessage instance encapsulating the the message sent to the server. In the failure case the standard error block is invoked.



## Canceling Requests

```
- (NSString *)fetchMessagesForChatroom:(NSString *)chatroomId  
    success:(void (^)(NSArray *messages))success  
    failure:(CCVChatcaveServiceFailure)failure;
```

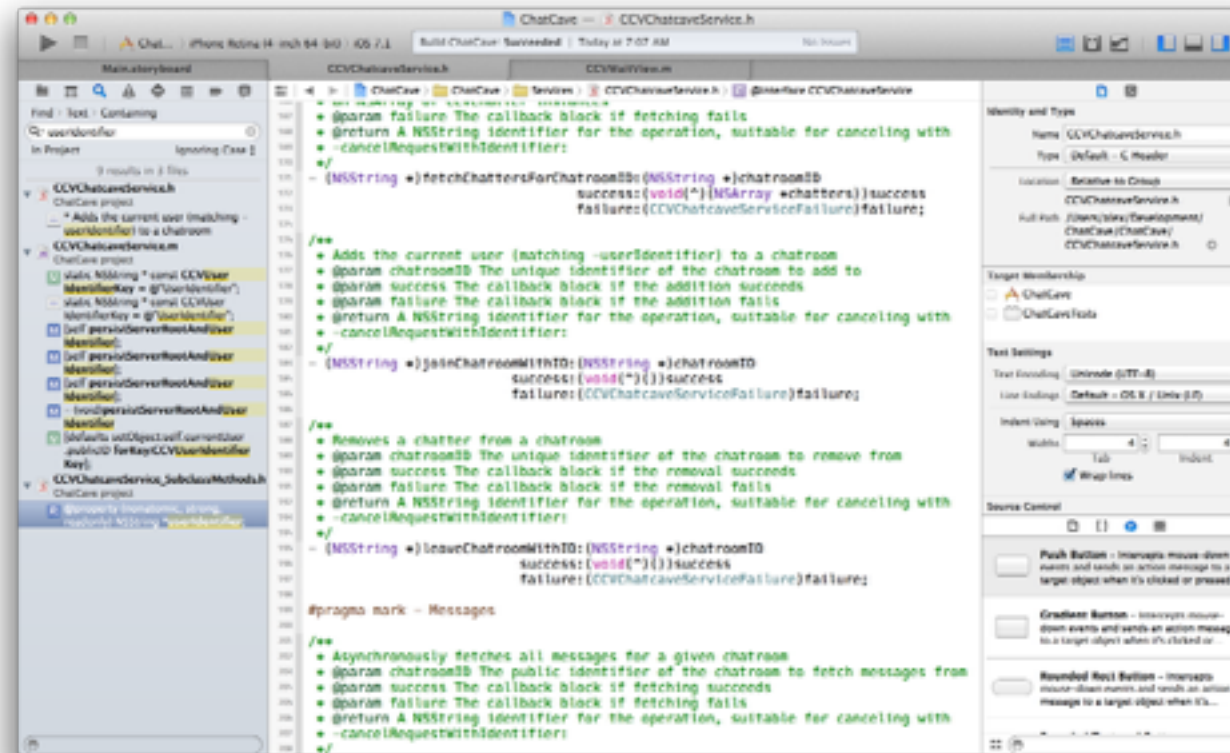


```
- (void)cancelRequestWithIdentifier:(NSString *)identifier;
```

Since our API requests will be asynchronous, our user may decide to abandon the screen that issued them before they return. We need a way to cancel inflight requests both to avoid unnecessary processing and callback blocks either retaining objects too long or crashing our app. We'll look at this in detail later, but for now let's declare this method to cancel requests.

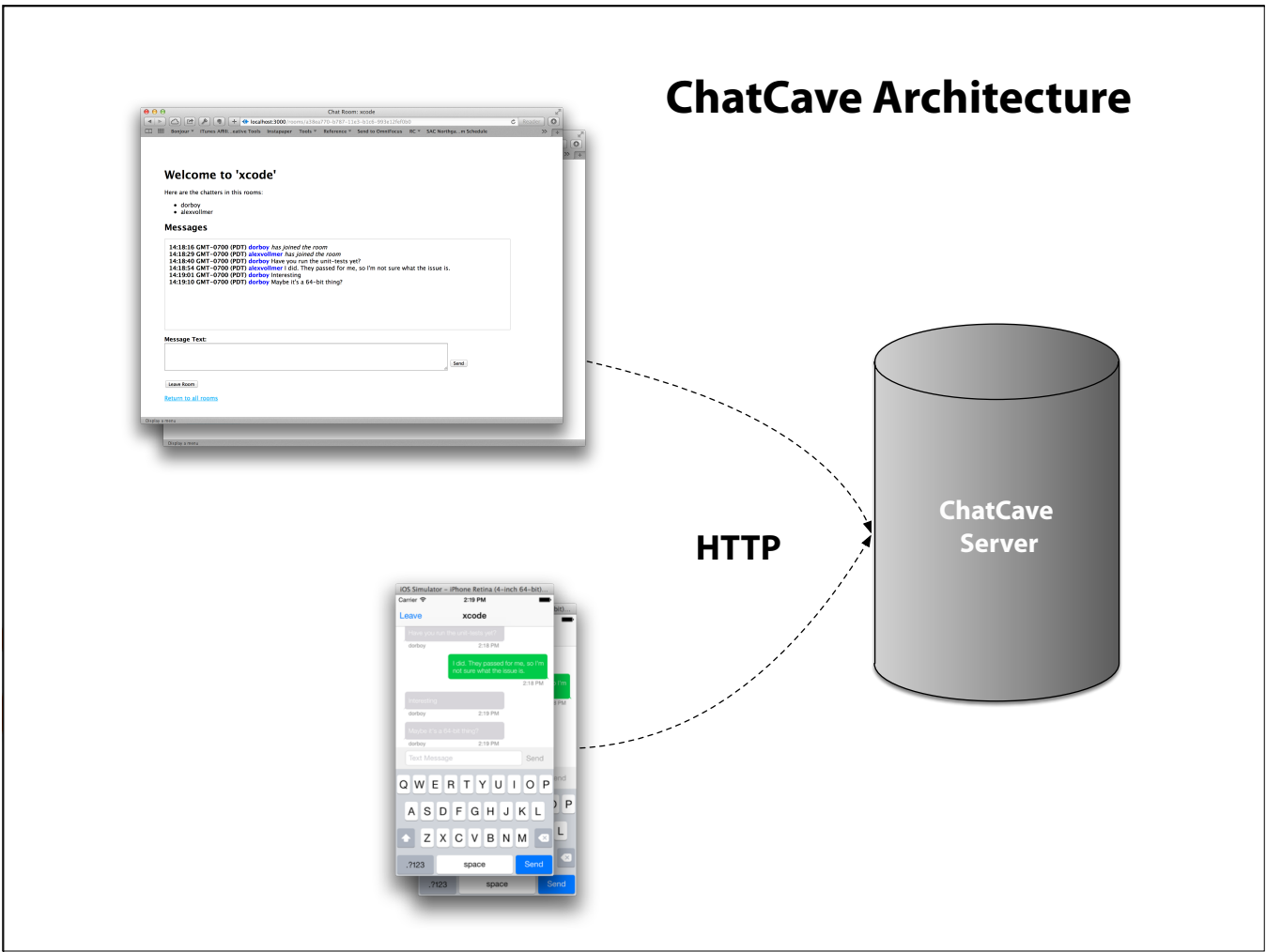
You may have noticed earlier that each request method in our API returns a NSString instance. That string is a unique identifier for the request and allows us to identify to the ChatcaveService which request we want to cancel.

Notice that we don't return an actual request object from these methods. This is because we want to avoid any unnecessary retain calls on the requests. We'll discuss this in more detail in a bit.



Now that we have an idea of how we want to proceed, let's get ready to create our first implementation of the `CCVChatcaveService`. In the next module we will look at using the `NSURLConnection` to power our `ChatcaveService` class.

# ChatCave Architecture



Let's review what we've looked at so far. First, I introduced the ChatCave sample application to you. We saw how its REST structure is able to serve a variety of clients.

## Golden Rules



**The main thread is for the user**

**UI can only be manipulated on main thread**

**Networking must be asynchronous**

**Do not directly create threads**

**Don't block the user**

We also established a few golden rules for the design of our application. These rules are true for any iOS application, not just the one we're building in this course.

## Key Elements of REST

Unlimited set of nouns

Limited set of verbs

URLs describe hierarchy

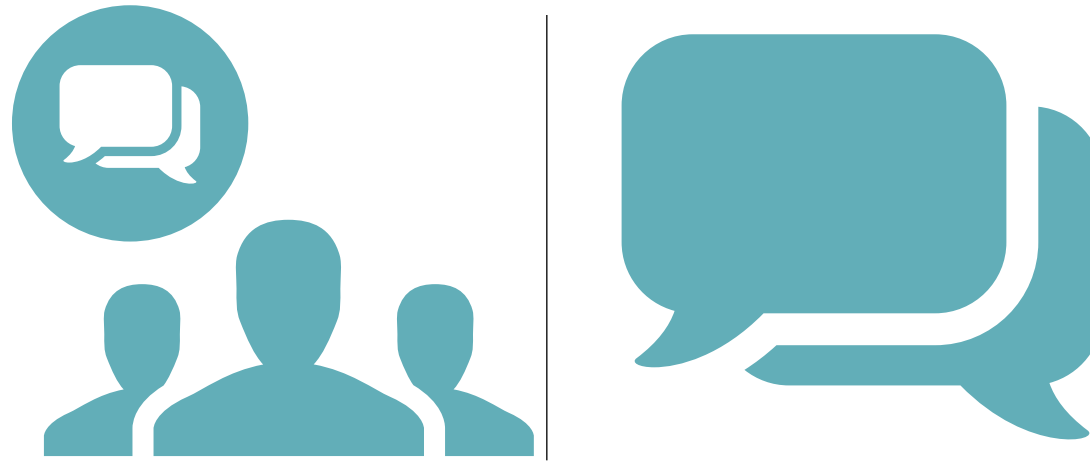
Multiple representations

Orthogonal



We quickly reviewed the key elements of a REST-ful architecture. With these design principles in mind, you can have a deeper appreciation for how the ChatCave server is built.

## Chatroom Resources



We reviewed the key resources in the ChatCave server application and verbs that we can apply to them.

# Model Classes

CCVChatter
- (instancetype)initWithPublicID:(NSString *)publicID name:(NSString *)name; - (instancetype)initWithDictionary:(NSDictionary *)dict; - (NSDictionary *)dictionaryRepresentation;

CCVChatroom
- (instancetype)initWithPublicID:(NSString *)publicID name:(NSString *)name chatters:(NSArray *)chatters; - (instancetype)initWithDictionary:(NSDictionary *)dictionary; - (NSDictionary *)dictionaryRepresentation;

CCVMessage
- (instancetype)initWithPublicID:(NSString *)publicID type:(CCVMessageType)type text:(NSString *)text author:(NSString *)author timestamp:(NSDate *)date; - (instancetype)initWithDictionary:(NSDictionary *)dictionary; - (NSDictionary *)dictionaryRepresentation;

We also looked briefly at how our Objective-C domain model objects reflect the structure of the server-side resources.

# iOS App Design



Since we're not building this application from scratch, we took a little time to review the existing iOS application architecture. A good portion of the application has already been written so that you can focus on integrating the networking layer.



## Design Goals

Block-based callbacks

Use domain-model objects

Abstract away details

Can be canceled



We also gave ourselves some design goals for implementing our networking layer. In the next module we will apply these guidelines as we dive into our first implementation of the networking layer.