BarterBay System Manual

Group 093
Peter Crampton - Project Manager
Chingiz Mardanov - Developer
Collin Barlage - Developer
Pratik Shekhar - Developer

Table Of Contents

Introduction

What is BarterBay?

BarterBay is a mobile application, developed using the **Ionic[1]** platform with the purpose of allowing users to sign up for account within the BarterBay domain and use their account to interact with other users also within the BarterBay domain and trade items with them. The application has two major components:

- ➔ A mobile application which serves as a front-end for end users to use in order to interact with BarterBay
- ➔ Back-end components that include a database, **XMPP[2]** server, and **NodeJS[3]** server application.

We believe that BarterBay serves the purpose of filling a need for an application which allows users to trade with each other as opposed to simply using currency and purchasing items from each other. BarterBay represents a different paradigm of shopping and we believe this is what makes BarterBay unique.

BarterBay General Structure

The general structure of BarterBay is shown in the diagram below. Each component will be discussed in detail.

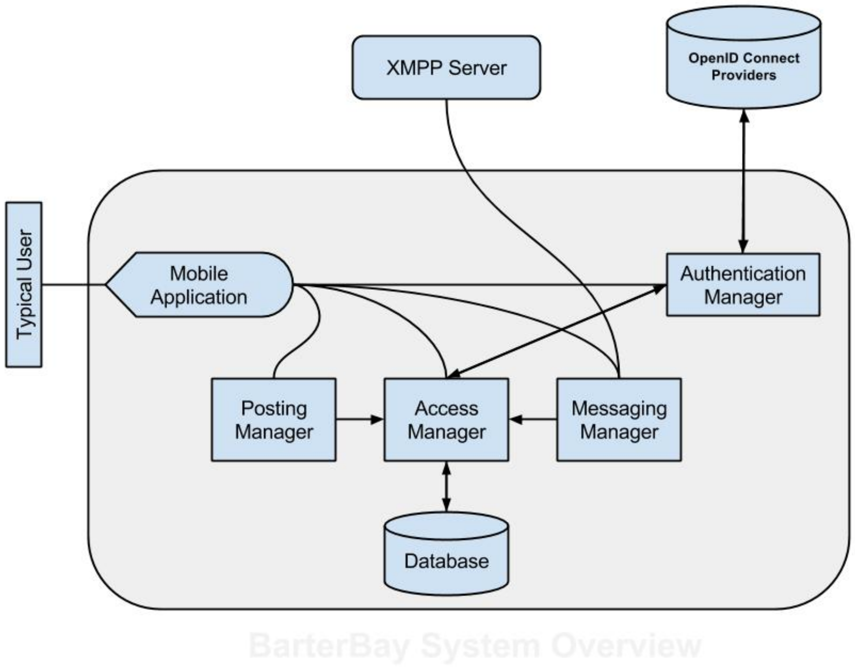


Figure 1 - BarterBay System Overview

Mobile Application

The mobile application of BarterBay in many respects represents the most important piece of the entire system. This single integral component provides end users with a friendly way to interact with the more complex foundations upon which the BarterBay system is built.

When planning out the design of the BarterBay mobile application, one of the major considerations taken into account was cross-platform availability. Our team recognized that in today's digital landscape there are numerous different devices which end users may like to use a system like BarterBay for and it was for this reason that our team chose to make portability our number one priority.

Our team acknowledged that it would be beneficial to only have one singular code base and instead of having to re-write every portion of a particular code base to compatible across devices our team decided upon a novel approach which has been gaining more momentum in the past several years: Hybrid Application Development.

The major premise behind Hybrid Application Development is exactly what our team was focusing on: portability and lack of re-writing code for different platforms. In the scheme that our team chose, we decided to make use of a Hybrid Application Framework called the Ionic platform. The Ionic platform allows for developers to write in a common language which in this case was HTML, JavaScript, and CSS. The Ionic platform then allows developers the ability to take that code base and compile native applications which will run on numerous different devices. The major ones that our team was focused on were:

- ➔ iOS
- ➔ Android
- ➔ Windows Phone
- ➔ Blackberry

Ionic and the Hybrid application model are not without its faults however and our team had to work to resolve an issue involving a chatting system which will be discussed later. In general however Ionic provided our team with a solid foundation by which we were able to develop code which could then be used on many different platforms.

The technical details for how Ionic works are far beyond the scope of this document, but for the developer looking to modify the BarterBay mobile application, here are some general design choices that our team made and some code snippets from the codebase to demonstrate what different aspects of the application is doing.

Routing

First, the Ionic system is wrapped within a WebBrowser view which is native to the device that it is running on, this is what gives the platform its ability to make cross-platform applications run without modification, essentially the application is nothing more than a webpage cloaked to look like a native application.

When the application is first loading, one of the things we must do is setup a routing system. A routing system is essential to ensure that the application is able to navigate between pages without the application appearing to simply be a webpage and having the re-load on each navigation.

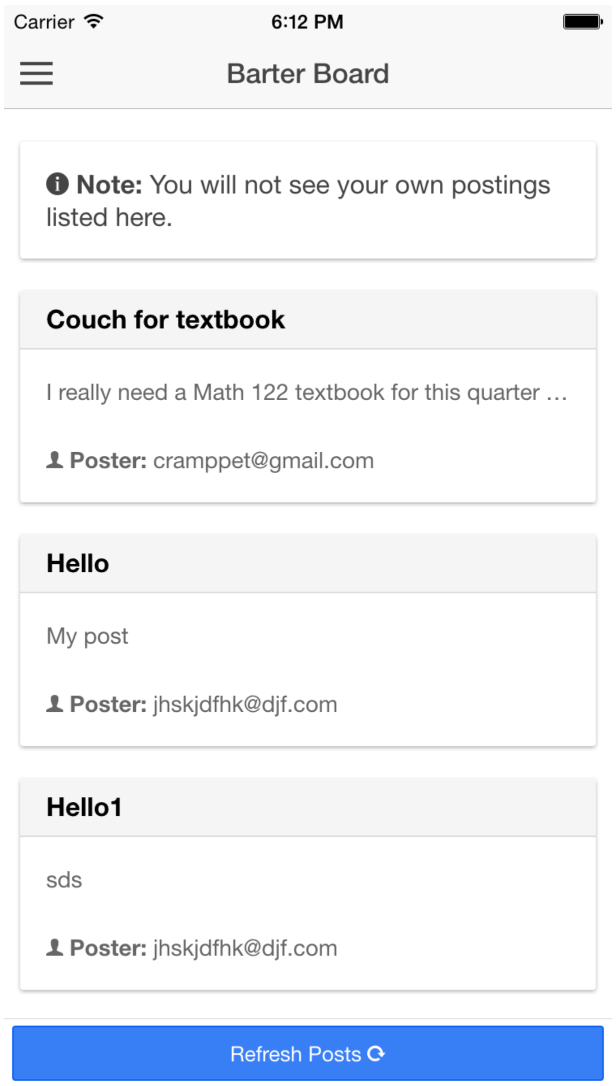
Writing a routing system by hand is tedious and also time expensive and there were many other considerations which made it highly impractical to solve by hand. The general way to solve this issue is to make use of a framework that supports a paradigm called Model-View-Controller or MVC. In this paradigm views or the static webpages are manipulated by controllers which also interact with data models that show dynamic content updates and also allows for pages to be accessed without having to re-load any particular part of the application that is not based on dynamic content.

When using the Ionic platform, the creators of the platform had already chosen a popular MVC framework to use called **AngularJS[4]**. The AngularJS framework gives JavaScript developers a very powerful way of interacting with the **Document Object Model or DOM[5]**. It also contains routing capabilities and the support of an increasingly large community which has developed numerous plugins and modules which can be used to extend AngularJS's functionality.

When making use of AngularJS along with Ionic, we were able to setup routing for our application based on states (a feature provided the angular-route plugin) and by making use of states we were able to pass parameters between states allowing the application to be able to drill down into the details of different data models and give users the ability to get more details about BarterBoard posts. The following is a snippet which shows how routing was accomplished within the application:

```
.config(function($stateProvider, $urlRouterProvider, $httpProvider, $authProvider) {
  $stateProvider

  .state('app', {
    url: "/app",
    abstract: true,
    templateUrl: "templates/menu.html",
    controller: 'AppCtrl'
  })

  .state('app.myposts', {
    url: "/myposts",
    views: {
      'menuContent': {
        templateUrl: "templates/myposts.html",
        controller: 'MyPostsCtrl'
      }
    }
  })

 … // Additional routes were present.

File: www/js/app.js
```

As can be seen from the code snippet above, setting up routing became a trivial task once our team understood how Ionic and AngularJS's routing schemes were supposed to be laid out. The snippet above defines different states which were going to be shown to the user via view templates. A controller is also bound to the state which allows for the manipulation of data models within the view. This is what a sample view would look like when rendered within the mobile application:



Carrier 6:12 PM

Barter Board

Note: You will not see your own postings listed here.

Couch for textbook

I really need a Math 122 textbook for this quarter …

Poster: cramppet@gmail.com

Hello

My post

Poster: jhskjdfhk@djf.com

Hello1

sds

Poster: jhskjdfhk@djf.com

Refresh Posts

Figure 2 - A screenshot taken from the iOS Simulator

The screenshot above shows what the "BarterBoard" looks like. It is simply a collection of user-submitted posts which can be viewed by all other users of the BarterBay system.

From an architectural point of view, the screen views for the mobile application can be summarized by looking at the following diagram:

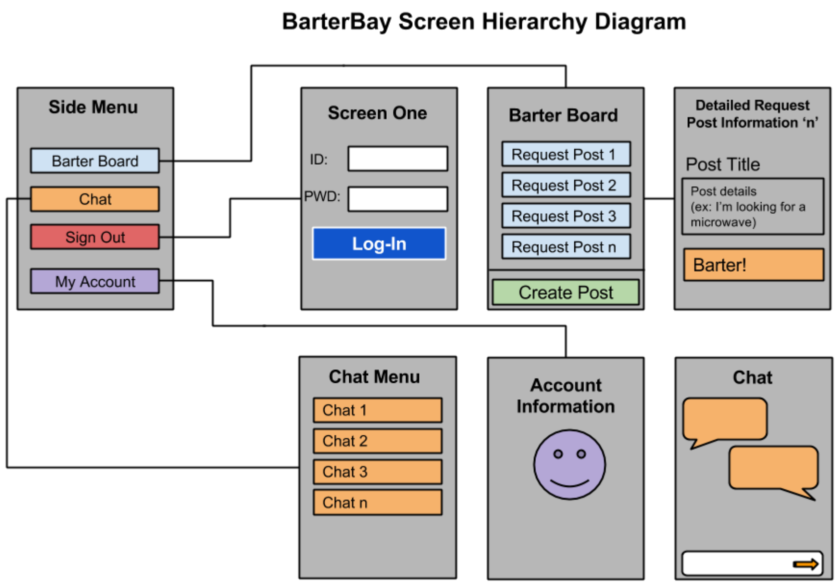


Figure 3 - The Mobile Application High Level Overview

Backend Interaction

After our team overcame how we were going to develop the mobile application, our next consideration became populating the views with data models that would come from a centralized stored location. In our case, we chose to aggregate chats among users along with account information and postings which were made to the BarterBoard in a central database. In order for the mobile application to have access to this information and make changes to this information two items had to be acknowledged:

- ➔ Users should not be allowed to do anything they wish on the system
- ➔ Users will have to interact with the backend system by through the mobile application which is based on client side web technologies

These tasks seem somewhat daunting however our team was able to overcome both issues by making use of AngularJS's **HyperText Transfer Protocol (HTTP)[6]** module which allowed our mobile application to send arbitrary HTTP requests. However, this alone was not enough. JavaScript and client side web technologies like it are not allowed to request items which are outside of their domain this is called the same origin policy. Essentially this prevents our mobile application from interacting with anything not directly hosted on the same physical device that the application is. Clearly this was not desirable. Luckily an open standard called **Cross-Origin Resource Sharing or CORS[7]** mitigated this from becoming an issue in our application because when implemented on a backend server our mobile application would then be allowed to request resources (ie. server endpoints part of an API) on the server and be able to get data back from the server and thus show dynamic content being displayed within the mobile application's views.

In terms of preventing users from performing certain actions based on who they are logged in as, this was also a challenging task. Providing secure authentication to a service is rarely easy and we had to take special considerations given the fact that again we were working as a Hybrid mobile application and we have special restrictions placed on us.

For authentication on the front-end our team chose a simple scheme whereby users would send their login credentials to the centralized server and would receive back a JSON Web Token or JWT. The generation of this token will be discussed more in the section on the backend however suffice to say the token gives the users their ability to interact with the system and without it the user cannot perform any actions to the backend database.

Our team also made the special effort to provide users the ability to log in using existing social media account providers. Our team made authentication available via Facebook and Google and provided users with an easy way to authenticate to our system without ever having to disclose their credentials to our system. This was a challenging process and involved backend server communication with Google and Facebook and also required our application to be verified by both providers before our application could be given permissions to request user account details based on the credentials the users provided when logging in.

Chatting

The final major hurdle with the front-end mobile application development was how to go about performing chatting between users. While simple websockets may have been able to perform this, our team realized that we would also like to have the extensibility to allow users to perform more complex chatting operations if they choose to. While not implemented in this release, our choice of protocol, **The Extensible Messaging and Presence Protocol (XMPP)[2]**, has many extensions which allows for file transfer, friend rosters, pub-sub relationships between users and multiple user chat room support. Implementing XMPP in JavaScript was at one time a great challenge however through traditional JavaScript, a library called Strophe.js emerged to provide JavaScript with the ability to communicate via XMPP. In practice however, Strophe.js was not built for running on Hybrid mobile applications and this produced some challenges which our team had to overcome, the major challenges were:

- ➔ Allowing propagation of Strophe.js events within the context of Ionic and AngularJS
- ➔ Integrating our own personal authentication system with an XMPP based authentication system
- ➔ Keeping stored conversations on the backend and having them be refreshed at the appropriate time without using local storage.

Chatting was the last piece of the mobile application put in place and this was due to the fact that it depended upon many other aspects of the application to be finished before being able to implemented successfully. To gain an understanding of how the chatting works within the application, take a look at the following code-snippet. In the example, we are showing how our group had to wrap the Strophe.js library into a custom

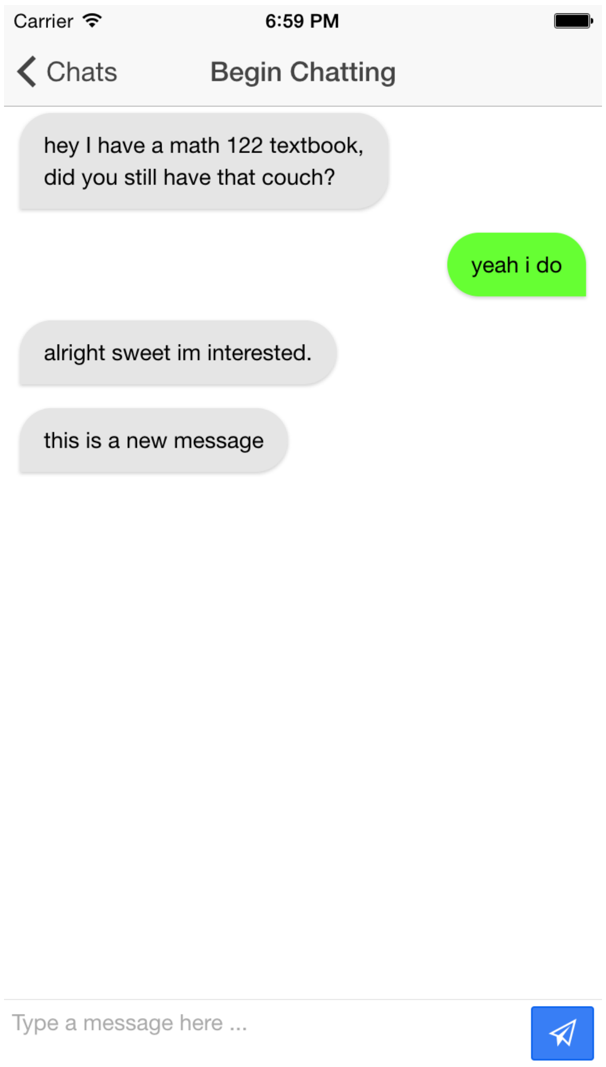
AngularJS service and then deal with the event callbacks from the XMPP communication happening in the background via either stateful websockets or via XHR polling and then take the data collected from the event callback and push the data to the correct AngularJS controller via a custom AngularJS event trigger:

```
.service('strophe', function strophe($rootScope, $http, $window) {
    var XMPPConnection = null;

    $rootScope.STROPHE_onMessage = function(stanza) {
      if (stanza.getAttribute("from") !== 'undefined') {
        var senderuser = stanza.getAttribute("from").split('@')[0];
        $rootScope.$broadcast('strophe-message-recieved', { from: senderuser });
      }
      return true;
    };

    $rootScope.STROPHE_sendMessage = function(message, toUser, fromUser, chatId) {
      $http.post('[SERVER ENDPOINT OMITTED]' + chatId, {
          senderuser:     fromUser,
          messagebody:    message
        })
        .success(function(res) {
          var uniqueId = XMPPConnection.getUniqueId();

          var messageStanza = $msg({
            "id": uniqueId,
            "to": toUser
          }).c("body").t(message);

          XMPPConnection.send(messageStanza.tree());
          $rootScope.$broadcast('strophe-message-sent', { to: toUser });
        });
    };

    $rootScope.STROPHE_onConnect = function(status) {
      if (status === Strophe.Status.CONNECTED) {
        XMPPConnection.addHandler($rootScope.STROPHE_onMessage, null, 'message', null, null,
null);
        XMPPConnection.send($pres().tree());
      }
      return true;
    };

    var stropheService = {
      createConnection: function(username, password) {
        XMPPConnection = new Strophe.Connection('[SERVER ENDPOINT OMITTED]');
        XMPPConnection.connect(username, password, $rootScope.STROPHE_onConnect);
      },
      destroyConnection: function() {
        XMPPConnection.options.sync = true;
        XMPPConnection.flush();
        XMPPConnection.disconnect();
      },
      sendMessage: function(message, toUser, fromUser, chatId) {
        $rootScope.STROPHE_sendMessage(message, toUser, fromUser, chatId);
      }
    };

    return stropheService;
  })
```

This code snippet is the AngularJS Strophe.js service which our group had to implement in order to have Strophe.js and AngularJS communicate effectively. The next code snippet shows the chatting controller which handled receiving messages from the Strophe.js service:

```
.controller('ChatsCtrl', function($scope, $window, $http, alert) {
  $scope.username = $window.localStorage['loggedInUser'];

  $scope.refresh = function() {
    $http.get('[SERVER ENDPOINT OMITTED]' + $scope.username)
      .success(function(chats) {
        $scope.chats = chats;
      })
      .error(function(err) {
        alert('assertive', 'Error fetching chats', err.message);
      });
  };

  $scope.$on('$stateChangeSuccess', function(event, toState, toParams, fromState, fromParams) {
    if (toState.url === '/chats')
      $scope.refresh();
  });

  $scope.$on('logout-purge', function() {
    $scope.chats = {};
  });

  $scope.$on('strophe-message-recieved', function() {
    $scope.refresh();
  });
})

.controller('ChatCtrl', function($scope, $stateParams, $window, $http, strophe, alert) {
  $scope.chatId = $stateParams.chatId;
  $scope.username = $window.localStorage['loggedInUser'];

  $scope.refresh = function() {
    $http.get('[SERVER ENDPOINT OMITTED]' + $scope.chatId)
      .success(function(messages) {
        if (messages.length > 0) {
          $scope.messages = messages;

          if (!($scope.toUser)) {
            if (messages[0].sender === $scope.username)
              $scope.toUser = messages[0].reciever;
            else
              $scope.toUser = messages[0].sender;
          }
        }
      })
      .error(function(err) {
        alert('assertive', 'Error fetching messages', err.message);
      });
  };

  $scope.sendMessage = function() {
    strophe.sendMessage($scope.message, $scope.toUser, $scope.username, $scope.chatId);
    $scope.message = '';
  };

  $scope.$on('strophe-message-recieved', function() {
    $scope.refresh();
  });
```

In both of these code snippets we demonstrated how Strophe.js and AngularJS could be made to work together in order to provide a chatting experience for the user which resembled that of a native chatting application. The following is a screenshot of the chatting running within the application:



hey I have a math 122 textbook, did you still have that couch?

yeah i do

alright sweet im interested.

this is a new message

Figure 7 - Two users chatting

As you can see, the mobile application was clearly one of the most important aspects of the BarterBay system and there were many considerations which had to be taken into account in order to make the system work correctly and easily. In the next section, we discuss the backend service which supports the front-end mobile application.

Backend Service

While the mobile application is what the users are seeing, that is not what they are actually interacting with. In reality, the users are communicating with the data stored in our database and are interacting with an API that our team built out to give the mobile application the functionality that it has. In this section we are going to explore what happens behind the scenes with BarterBay.

Unlike the mobile application, portability was not a concern when deciding on the best tools for implementing our backend service, instead the considerations that our team had were the following:

- ➔ Working with a familiar language for our developers
- ➔ Using event based systems that allow for asynchronous operations (like communicating with Google and Facebook)
- ➔ Use a language that is fast enough to be responsive for several requests happening at once

Based on all of these considerations, our team chose to make use of the **NodeJS[3]** implementation of JavaScript for backend services. It fit all of the considerations previously enumerated and had the added benefit of popularity which allowed our team to leverage many existing node modules to simplify the code which we had to write to support BarterBay.

In terms of a database, again our team wanted to simplify our development experience by not having to design tables and structures typically found in traditional databases. Instead, we wanted our database to be dynamic, flexible and allow for communication with the NodeJS server application which we would be writing. Based on these considerations, we chose to make use of the **MongoDB[8]** database. MongoDB and NodeJS go hand-in-hand very well. There are excellent NodeJS modules built out for interacting with a MongoDB database and MongoDB is also a NoSQL database meaning that it does not permit the use of SQL as a way of performing queries of data entries. Instead, MongoDB allows users to write JavaScript to perform queries which once again reduced the number of different languages our team needed to be fluent in in order to develop this project.

The backend service can be broken down into three separate sub services which are as follows:

- ➔ Authentication Management
- ➔ Posting Management
- ➔ Chatting Management

Each of these components will now be discussed in detail.

Authentication Management

By far the most critical portion of the application was the management of authentication. Authentication is often a challenge and in this section we will discuss our implementation of authentication for the BarterBay system and provide code snippets to demonstrate how the system is working behind the scenes.

To begin, we must first understand how we wanted the authentication to work, for our system, we wanted authentication to be temporary, but also long lived enough to allow users to perform whatever they needed to do even when they would minimize the application and come back to it later. We did not want to embrace practices which had been shown numerous times to be insecure and overall a poor implementation of authentication. Examples of poor authentication strategies would have been: cookie authentication, storing authentication credentials within the running JavaScript code or keeping the information in local storage. In general, we wanted to embrace an encrypted token based authentication service which would require that in order for a user to authenticate to the system they must first log in and obtain a token which will then be passed along with every request they make and then validated by the server before they are given access to API resource endpoints.

In order to implement this scheme, we originally designed our own JSON Web Token service within NodeJS which allowed our clients to be granted an access token to our service based on the credentials they provided. The token was both encoded and encrypted and was validated upon each request to the server. Eventually, after we felt we understood the idea of JWT's well enough, our team decided to make use of an existing JWT library called "jwt-simple".

The authentication process works something like this:

1. A user sends a request to the /login endpoint on the server
2. The server checks the database to see if a user exists based on the credentials supplied
3. If a user is found the server builds a JWT for the client and sends it back as a response to the clients request.
4. The client then attaches their JWT to each request they make
5. The server decrypts and validates that the signature on the JWT is valid and approves the request
6. The server permits the user's action if and only if their JWT was validated.

The following is a code snippet taken from the backend which demonstrates what happens when the user hits the /login API endpoint:

```
app.post('/login', passport.authenticate('local-login'), function(req, res) {
      createSendToken(req.user, res);
});
```

In this snippet, we can see that we are choosing to make use of the Passport NodeJS library which we used to perform local authentication along with Google and Facebook authentication. Although the negotiation process with Google and Facebook is not nearly as simple as this local authentication example, this snippet shows how our service responds to login request from clients.

The other snippet which is relevant is the Passport local strategy which we were required to created and supply Passport with before it was able to handle the authentication process. This is the local strategy which we told Passport to use:

```
exports.loginStrategy = new LocalStrategy(config.strategyOptions, function(email, password, done)
{
        var searchUser = {
                email: email
        };

        User.findOne(searchUser, function(err, user) {
                if (err) {
                        console.log(JSON.stringify(err));
                        return done(err);
                }

                if (!user)
                        return done(null, false, { message: 'Wrong email/password' });

                user.comparePasswords(password, function(err, isMatch) {
                        if (err)
                                return done(err);

                        if (!isMatch)
                                return done(null, false, { message: 'Wrong email/password' });

                        return done(null, user);
                });
});
```

From this snippet, you can see how the BarterBay system is searching for user accounts based on the email provided and then is comparing the hashed and salted passwords stored in the database to the hashed and salted version of whatever the user passed in.

Searching based on email is important because this allows for multiple account synchronization. In general, this makes it possible for users to link their Facebook, Google and local BarterBay accounts together allowing them to login with any of the services and still be logged into the correct account.

As for a general request occurring, each time a request comes in the token sent with the request must be validated. Part of this validation process is shown here:

```
if (!req.headers.authorization) {
	res.status(401).send({
		message: 'You are not authorized'
	});
	return false;
}

var token   = req.headers.authorization.split(' ')[1];
var payload = jwt.decode(token, config.BARTERBAY_SECRET);

if (!payload.sub) {
	res.status(401).send({
		message: 'You are not authorized'
	});
	return false;
}

return true;
```

This validation process demonstrates how tokens, even though they may exist on the client, may be rejected by the server providing greater levels of security.

Registration works much the same way; with the client sending a request to a defined API endpoint. This endpoint handles the registration process by checking to make sure that another user with the email that the sending user is attempting to register to with does not already exist. It performs several other validation checks and returns errors at each step in the process.

Authentication via **OAuth[9]** providers like Google and Facebook is not standardized and the implementations for each provider are slightly yet subtly different. In particular the ways in which they expect the token negotiation process to happen is different. Our team had to become familiar with the authentication flow that both Google's OAuth system along with Facebook's goes through and be able to create implementations that would eventually integrate with the Passport NodeJS module which we were using for local authentication and local registration.

Our group had to look into the documentation for each provider and determine how to properly create requests to the necessary services and receive back long-lived authentication tokens while also requesting data from the OAuth provider in order to store information about the user who had signed up using the 3rd party provider.

Below is a code snippet which demonstrates the backend handling authentication flow with Google's OAuth system:

```
request.post(config.GOOGLE_URL, { json: true, form: params }, function(err, response, token) {
	var accessToken = token.access_token;

	var headers = {
		Authorization: 'Bearer ' + accessToken
	};

	request.get({ url: config.GOOGLE_API_URL, headers: headers, json: true }, function(err, 
response, profile) {
		User.findOne({ googleId: profile.sub }, function(err, foundUser) {
			if (foundUser)
				return createSendToken(foundUser, res);

			var newUser = new User();
			newUser.googleId = profile.sub;
			newUser.email = profile.email;
			newUser.posts = [];

			newUser.save(function(err) {
				if (err)
					return next(err);

				createSendToken(newUser, res);

			});
		});
	});
});
```

In the snippet above, we are working with Google's OAuth flow and are exchanging an access code which was sent from the client (and also obtained there) for a long-lived access token which we were able to use to get information about the user's Google Profile and then create them an account within our system.

Overall, authentication was one of the most important aspects of our application and a lot of time was spent ensuring that we were using industry best practices and are using as standard and compliant processes as possible. The rest of the backend hinged upon having a reliable account and authentication system so it was imperative that this system function correctly.

Posting Management

The second component of the backend service that our group focused on was postings. Postings are simply bits of data that users submit to BarterBay in the hopes of being able to find someone who will respond to their posting through chatting and will negotiate to barter with them for whatever it was that they wanted. Based on this description it is obvious to see that Posting and Chatting are both linked very closely with authentication being the glue that holds all of the backend service components together.

In general, posting is not very complex, by making use of a standard **REST[10]** API that allows for users to perform **CRUD[11]** operations on posting objects stored in our MongoDB instance it becomes a trivial task of simply setting up the proper API endpoints and handling the requests from the users to make sure they are authorized to perform whatever action they were planning on doing.

For all intents and purposes with regard to postings, our group was able to get away with simply using very basic routes which handled the incoming requests and perform the necessary database operations or otherwise returned any errors that the backend service encountered while trying to service the request that the user attempted to make. The following code snippet shows what some of the posting routes looked like:

```
app.get('/posts', function(req, res) {
	if (validateToken(req, res)) {
		Post.find(function(err, foundPost) {
			if (err)
				res.send(err);
			res.json(foundPost);
		});
	}
});

app.get('/posts/:postId', function(req, res) {
	if (validateToken(req, res)) {
		Post.findOne({ _id: req.params.postId }, function(err, foundPost) {
			if (err)
				res.send(err);
			res.json(foundPost);
		});
	}
});


app.delete('/posts/:postId', function(req, res) {
	if (validateToken(req, res)) {
		Post.remove({ _id: req.params.postId }, function(err, foundPost) {
			if (err)
				res.send(err);
			res.json({ message: 'Post successfully deleted.' });
		});
	}
});
```

These routes were interacted with by the client sending requests and the requests had to be authorized and validated by using the JWT sent back to the client when they logged in. They would receive responses that were either what they expected or errors based on lack of validation or some other error.

What postings do for BarterBay is quite simple, they allow for user submitted data to propagate throughout the system. They are the user submitted content that is the main focus of the application, however they are not complicated to implement, it is the extra elements like authentication and chatting which proved to be more challenging. Indeed, even on the client it was simple to implement posting request so long as CORS had been enabled on the server, which we did in fact enable by creating middleware that added the necessary HTTP headers to response which indicated that the server was willing to allow CORS.

Validation within the MongoDB instance was not necessary because MongoDB serializes the posting objects based on unique ID's that it creates so the data model that we were able to use within MongoDB to store posting objects was this:

```
var PostSchema = new mongoose.Schema({
       title:        String,
       description:  String,
       poster:              String,
       timestamp:    Date
});
```

This data model was enough to store all the information required in the postings and by making use of **Express[12]** as a NodeJS module, the REST capabilities within Express were more than enough to make the task of managing postings quite simple.

Chatting Management

Aside from authentication, chatting is one of the most critical parts of the application because it allows for the interaction between two users and facilitates conversations within the BarterBay domain thus allowing BarterBay's major goal of being able to help people barter with each other be accomplished. When our group was considering how to go about allowing chatting to take place we had a few things that we had to consider, first we had to decide on how we wanted to actually have the chatting be accomplished in terms of a protocol. Initially our group thought that perhaps a technology like **WebRTC[13]** could have been used to demonstrate how video and voice chatting could be done using only JavaScript however this became incredibly complex as we attempted to find a way to implement this within the BarterBay system.

We then broke down our considerations for choosing a chatting protocol into the following categories:

- ➔ Ease of implementation and integration with existing code
- ➔ Ability to handle offline messaging
- ➔ Allow for future extensions if desired

Based on the criteria outlined above, our group came to the conclusion that the Extensible Messaging and Presence Protocol (XMPP) was the best choice for our application as it filled all of the criteria above through a JavaScript library called **Strophe.js[14].** Strophe being a client which exists for front-end operations was not usable on the backend and we therefore had to work out a solution to handling chats on our own. We knew that there were several things that needed to happen in order for conversations and chatting to work within the BarterBay system. These were:

- ➔ Allowing for new conversations to be started
- ➔ Allowing for new messages to be added to existing conversations

➔ Allowing for conversations to synced offline

Fortunately, XMPP as a protocol handles the third consideration so we did not have to be concerned with this. However, the first two were features we would have to provide manually. Our group noted several interesting similarities between postings and chats and for this reason chose to implement the required chatting capabilities by using the same scheme used for creating postings--by exposing several API endpoints which when data was sent to would result in the system processing and performing the desired actions.

We had to set up several routes which were similar to those found in the section on Posting Management they looked similar to the following:

```
app.get('/chat/:chatId', function(req, res) {
        if (validateToken(req, res)) {
                Chat.findOne({ _id: req.params.chatId }, function(err, foundChat) {
                        if (err)
                                res.send(err);

                        if (foundChat)
                                res.json(foundChat.messages);

                        else
                                res.json({ message: 'Could not find conversation.' });
                });
        }
});
```

In general, our team noted that the routes for handling chatting were more complex than those for simple postings this was because we did not want duplicate messages to be put into the database and for this reason the way in which we perform storing messages was when each client would receive a message then would sent a request to have the message stored in the database. The reason why this is beneficial is because XMPP guarantees the delivery of messages even if offline, so when the user would log back in they would connect to the XMPP server and then immediately be sent any messages that they did not receive while offline. This also ensures that if there was a delivery error that it would not propagate into having multiple of the same messages existing within the database.

Our group chose to have our own instance of an XMPP server running on our local server and integrate accounts with that server by writing new accounts to its database (which is just a flat file database). This made it easy to link posting accounts to chatting accounts and use one single addressable scheme to keep messages and postings linked to a single account.

Conclusion

Overall, as you can see the BarterBay system is not very complicated, while there were several challenges that had to be overcome based on the criteria that we wanted our application to fulfill it was not overly laborious or terribly difficult and in general there were not many hurdles which took more than a couple of days to resolve completely.

BarterBay's two component nature and its breakdown into modules and services allows for it to be modified easily and demonstrates its ability to invite future improvement and extensions that developers would desire to make.

Glossary

Here are the list of technical terms used throughout the documentation along with respective definitions:

1. **Ionic** - A hybrid mobile application platform sponsored by Google which allows for developers to create mobile applications that run on various platforms by using a single codebase written in client side web technologies.
2. **Extensible Messaging and Presence Protocol (XMPP)** - A popular messaging protocol used to exchange messages between two clients via a dedicated XMPP server. Server's can either be turned into federated distributed nodes like SMTP or isolated to only work within a specific domain. XMPP has many extensions that are community supported.
3. **NodeJS** - A server side implementation of the JavaScript language used to create code which runs on servers. In most cases, NodeJS is used to create Web servers which handle incoming request and IO operations on the server.
4. **AngularJS** - A popular model-view-controller framework for JavaScript which is developed by Google. Allows JavaScript code to segmented in modules, gives the concrete concept of scope to separate controllers, and is supported by many community plugins.
5. **Document Object Model (DOM)** - A model which represents all of the objects present within a webpage. These are typically represented as nodes of an HTML document.
6. **HyperText Transfer Protocol (HTTP)** - The standardized web protocol used for transferring web pages and various content to HTTP clients like web browsers.
7. **Cross-Origin Resource Sharing (CORS)** - A standard used to overcome the challenges of JavaScript's same origin policy. In general only a couple of headers must to added to a server's HTTP responses in order for this to work correctly.
8. **MongoDB** - A NoSQL database which stores data records as collections and documents. Allows for querying to be done via JavaScript.
9. **OAuth** - An open standard used to allow 3rd parties to authenticate on behalf of a user. This technology is a part of the Single Sign On paradigm.

10. **Representational State Transfer (REST)** - A series of guidelines for designing application. In general REST does not support "states" and works from request to request meaning validation must be performed per request.
11. **Create, Read, Update, Delete (CRUD)** - A series of common operations which are typically done on database objects.
12. **Express** - A very popular NodeJS module which gives many capabilities to web server applications like routing and numerous middleware components based on the Connect NodeJS module.
13. **WebRTC** - A new standard being developed for peer-to-peer communication between internet connected devices.
14. **Strophe.js** - A JavaScript library which allows for XMPP clients to be written in native JavaScript.