Build a real-time polls application with Node.js, Express, AngularJS, and MongoDB

Joe Lennon

June 27, 2014
(First published December 03, 2013)

Build a polling application that updates in real-time as votes roll in. The app I created has a simple architecture that uses JavaScript for everything. Node.js and Express power the back-end, and MongoDB stores the app's data. At the front-end, AngularJS and Bootstrap power the user interface, with Web Sockets enabling the voting to update clients in real-time.

To view the introductory video **Build a real-time polls application with Node.js, AngularJS, and MongoDB** , please access the online version of the article. If this article is in the developerWorks archives, the video is no longer accessible.

Recently while lecturing on HTML5 to a large group of students, I wanted to poll them and display their voting results, updating in real-time. I decided to quickly build a polling app for this purpose. I wanted a simple architecture and not too many different languages and frameworks. So I decided to use JavaScript for everything — Node.js and Express for the server-side, MongoDB for the database, and AngularJS for the front-end user interface.

> *“ This MEAN stack (Mongo, Express, Angular, Node) may one day surpass the simplicity of the LAMP stack (Linux, Apache, MySQL, PHP) for web application development and deployment. ”*

Run the app
Get the code

I chose to use DevOps Services (formerly JazzHub) to manage the source code for my project. Not only does it give me a full version control system for my code, but it also has an online IDE for editing my code in the cloud, and abundant agile features for project management. DevOps Services also integrates easily with Eclipse, which has plug-ins to enable one-click deployment to platforms like Bluemix or Cloud Foundry.

What you'll need to build your app

- A basic familiarity with Node.js, and a Node.js development environment
- These Node.js modules: Express framework, Jade, Mongoose, and socket.io
- AngularJS JavaScript framework
- MongoDB NoSQL database
- The Eclipse IDE, with the Nodeclipse plug-in installed

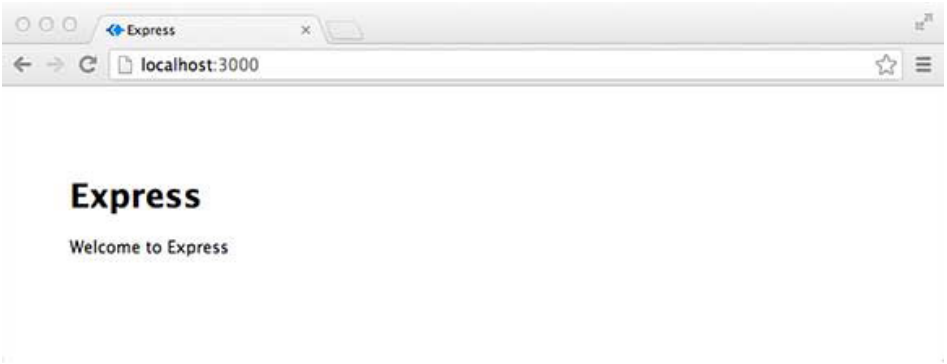
Step 1. Build a basic Express back-end

In Eclipse, switch to the Node perspective, and create a new Node Express project. If you create a DevOps Services project, as I did, name your Node Express project with the same name. Leave Jade selected as the template engine. Eclipse will automatically download the required npm modules to create a simple Express app.

Run the Express app

In Project Explorer, find app.js in the root of your project, right-click and choose **Run As** > **Node Application**. This will start a Web server and deploy the app to it. Next, open your browser and navigate to http://localhost:3000.

Starter Express app



Configure the basic front-end

The polls app uses the **Bootstrap framework** for the general user interface and layout. Let's make some changes to the Express app now to reflect this. First, open routes/index.js, and change the title property to `Polls`:

routes/index.js

```
        exports.index = function(req, res){
             res.render('index', { title: 'Polls' });
         };
```

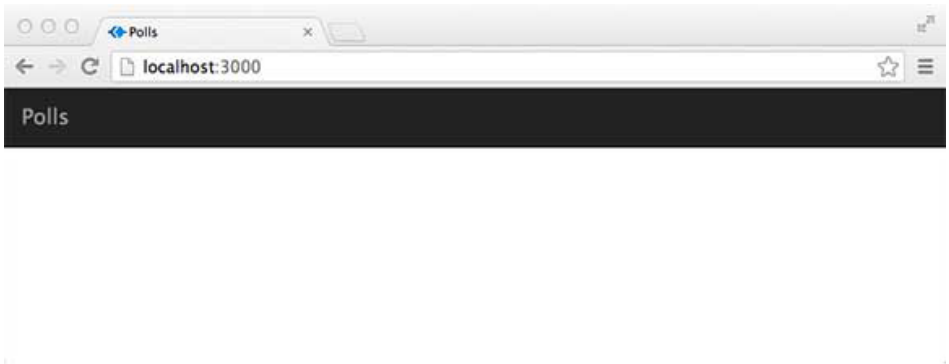
Next, change the views/index.jade template to include Bootstrap. Jade is a short-hand template language that compiles to HTML. It uses indentation to remove the need for closing tags, significantly reducing the size of your templates. You'll use Jade for the main page layout only. You'll use Angular partial templates to add functionality to this page in the next step.

views/index.jade

```
        doctype 5
        html(lang='en')
          head
            meta(charset='utf-8')
            meta(name='viewport', content='width=device-width,
 initial-scale=1, user-scalable=no')
            title= title
            link(rel='stylesheet', href='//netdna.bootstrapcdn.com/bootstrap/3.0.1/
 css/bootstrap.min.css')
            link(rel='stylesheet', href='/stylesheets/style.css')
          body
            nav.navbar.navbar-inverse.navbar-fixed-top(role='navigation')
              div.navbar-header
                a.navbar-brand(href='#/polls')= title
              div.container
                div
```

To see the changes to your app, kill the web server process in Eclipse, and run the app.js file again:

Polls app boilerplate



Note: When using Jade templates, take care to indent your code properly, or you will run into trouble. Also, avoid mixing indentation styles, as Jade will give you errors if you try.

Step 2. Craft the front-end user experience with AngularJS

To start using Angular, you first need to include it and add some directives within your HTML page. In the views/index.jade template, change the `html` element to the following: `html(lang='en', ng-app='polls')`.

Within the head block in this file, add the following script elements:

Script elements to load Angular and the Angular Resource module

```
        script(src='//ajax.googleapis.com/ajax/libs/angularjs/1.0.8/angular.min.js')
        script(src='//ajax.googleapis.com/ajax/libs/angularjs/1.0.8
/angular-resource.min.js')
```

Next, change the `body` element in the template, adding an `ng-controller` attribute (to later bind the user interface to the controller logic code): `body(ng-controller='PollListCtrl')`.

Finally, change the last `div` element in the template to include an `ng-view` attribute: `div(ng-view)`.

Build the Angular module

> An impressive feature in Angular is data binding, which automatically updates your views when the back-end models change. This drastically reduces the amount of JavaScript you need to write, as it abstracts away the messy task of manipulating the DOM.

By default, Express publishes static resources like JavaScript source files, **CSS stylesheets**, and images from the public directory in your project. In the public directory, create a new subdirectory named javascripts. In this subdirectory, create a file named app.js. This file will contain the Angular module for the app, defining the routes and templates to use for the user interface:

public/javascripts/app.js

```
        angular.module('polls', [])
          .config(['$routeProvider', function($routeProvider) {
            $routeProvider.
              when('/polls', { templateUrl: 'partials/list.html', controller:
PollListCtrl }).
              when('/poll/:pollId', { templateUrl: 'partials/item.html', controller:
PollItemCtrl }).
              when('/new', { templateUrl: 'partials/new.html', controller:
PollNewCtrl }).
              otherwise({ redirectTo: '/polls' });
          }]);
```

Angular controllers define the scope of your app, providing data and methods for the views to bind to.

public/javascript/controllers.js

```
        // Managing the poll list
        function PollListCtrl($scope) {
          $scope.polls = [];
        }
        // Voting / viewing poll results
        function PollItemCtrl($scope, $routeParams) {
          $scope.poll = {};
          $scope.vote = function() {};
        }
        // Creating a new poll
        function PollNewCtrl($scope) {
          $scope.poll = {
            question: '',
            choices: [{ text: '' }, { text: '' }, { text: '' }]
          };
          $scope.addChoice = function() {
            $scope.poll.choices.push({ text: '' });
          };
          $scope.createPoll = function() {};
        }
```

Create the partial HTML templates

To render data from the controllers, Angular uses partial HTML templates that allow you to use placeholders and expressions to include data and perform operations such as conditionals and iterators. In the public directory, create a new subdirectory named partials. We will create three

partials for our app. The first partial will display the list of polls available, and we will use Angular to easily filter this list by a search field.

public/partials/list.html

```
        <div class="page-header">
          <h1>Poll List</h1>
        </div>
        <div class="row">
          <div class="col-xs-5">
            <a href="#/new" class="btn btn-default"><span class="glyphicon
glyphicon-plus"></span> New Poll</a>
          </div>
          <div class="col-xs-7">
            <input type="text" class="form-control" ng-model="query"
placeholder="Search for a poll">
          </div>
        </div>
        <div class="row"><div class="col-xs-12">
<hr></div></div>
        <div class="row" ng-switch on="polls.length">
          <ul ng-switch-when="0">
            <li><em>No polls in database. Would you like to
 <a href="#/new">create one</a>?</li>
          </ul>
          <ul ng-switch-default>
            <li ng-repeat="poll in polls | filter:query">
              <a href="#/poll/{{poll._id}}">{{poll.question}}</a>
            </li>
          </ul>
        </div>
        <p> </p>
```

The second partial allows the user to view the poll. It uses the Angular switch directive to determine whether the user has voted. Based on that determination, it will display either a form to vote on the poll, or a chart with the poll results.

public/partials/item.html

```
        <div class="page-header">
          <h1>View Poll</h1>
        </div>
        <div class="well well-lg">
          <strong>Question</strong><br>{{poll.question}}
        </div>
        <div ng-hide="poll.userVoted">
          <p class="lead">Please select one of the following options.</p>
          <form role="form" ng-submit="vote()">
            <div ng-repeat="choice in poll.choices" class="radio">
              <label>
                <input type="radio" name="choice" ng-model="poll.userVote"
 value="{{choice._id}}">
                {{choice.text}}
              </label>
            </div>
            <p><hr></p>
            <div class="row">
              <div class="col-xs-6">
                <a href="#/polls" class="btn btn-default" role="button"><span
class="glyphicon glyphicon-arrow-left"></span> Back to Poll
              </div>
              <div class="col-xs-6">
```

```
              <button class="btn btn-primary pull-right" type="submit">
 Vote &raquo;</button>
            </div>
          </div>
        </form>
      </div>
      <div ng-show="poll.userVoted">
        <table class="result-table">
          <tbody>
            <tr ng-repeat="choice in poll.choices">
              <td>{{choice.text}}</td>
              <td>
                <table style="width: {{choice.votes.length
 /poll.totalVotes*100}}%;">
                  <tr><td>{{choice.votes.length}}</td></tr>
                </table>
              </td>
            </tr>
          </tbody>
        </table>
        <p><em>{{poll.totalVotes}} votes counted so far. <span
ng-show="poll.userChoice">You voted for <strong>{{poll.userChoice.text}}
</strong>.</span></em></p>
        <p><hr></p>
        <p><a href="#/polls" class="btn btn-default" role="button">
<span class="glyphicon glyphicon-arrow-left"></span> Back to
Poll List</a></p>
      </div>
      <p> </p>
```

The third and final partial defines the form that allows the user to create new polls. It asks the user to enter a question and three choices. A button is provided to allow additional choices to be added. Later, we will validate that the user has entered at least two choices — as it wouldn't be much of a poll without some choices.

public/partials/new.html

```
      <div class="page-header">
        <h1>Create New Poll</h1>
      </div>
      <form role="form" ng-submit="createPoll()">
        <div class="form-group">
          <label for="pollQuestion">Question</label>
          <input type="text" ng-model="poll.question" class="form-control"
id="pollQuestion" placeholder="Enter poll question">
        </div>
        <div class="form-group">
          <label>Choices</label>
          <div ng-repeat="choice in poll.choices">
            <input type="text" ng-model="choice.text" class="form-control"
placeholder="Enter choice {{$index+1}} text"><br>
          </div>
        </div>
        <div class="row">
          <div class="col-xs-12">
            <button type="button" class="btn btn-default" ng-click=
"addChoice()"><span class="glyphicon glyphicon-plus">
</span> Add another</button>
          </div>
        </div>
        <p><hr></p>
        <div class="row">
          <div class="col-xs-6">
            <a href="#/polls" class="btn btn-default" role="button">
```

```
<span class="glyphicon glyphicon-arrow-left"></span>
Back to Poll List</a>
            </div>
            <div class="col-xs-6">
              <button class="btn btn-primary pull-right" type="submit">
 Create Poll &raquo;</button>
            </div>
          </div>
          <p> </p>
        </form>
```

Finally, to display the results, we need to add a few CSS declarations to the style.css file. Replace the contents of this file with this:

public/stylesheets/style.css

```
        body { padding-top: 50px; }
        .result-table {
          margin: 20px 0;
          width: 100%;
          border-collapse: collapse;
        }
        .result-table td { padding: 8px; }
        .result-table > tbody > tr > td:first-child {
          width: 25%;
          max-width: 300px;
          text-align: right;
        }
        .result-table td table {
          background-color: lightblue;
          text-align: right;
        }
```

At this point, if you run the app, you will see an empty poll list. If you try to create a new poll, you will be able to see the form and add more choices, but you won't be able to save the poll. We'll tie up all of this in the next step.

Step 3. Store data in MongoDB using Mongoose

To store data, the app uses the MongoDB driver and Mongoose npm modules. These allow the app to communicate with a MongoDB database. To get these modules, open the package.json file in the app's root directory, and in the dependencies section, add these lines:.

Adding lines to the dependencies section

```
        "mongodb": ">= 1.3.19",
        "mongoose": ">= 3.8.0",
```

Save the file, right-click it in Project Explorer, and choose **Run As** > **npm install**. This will install the npm modules and any additional dependencies.

Create a Mongoose model

Create a new subdirectory in your app's root named models, and in that subdirectory, create a new file named Poll.js. This is where we'll define our Mongoose model, which will be used to query and save data to MongoDB in a structured manner.

models/Poll.js

```
var mongoose = require('mongoose');
var voteSchema = new mongoose.Schema({ ip: 'String' });
var choiceSchema = new mongoose.Schema({
  text: String,
  votes: [voteSchema]
});
exports.PollSchema = new mongoose.Schema({
  question: { type: String, required: true },
  choices: [choiceSchema]
});
```

Define API routes for data storage

Next, set up some routes in the app.js file at the root of your app to create JSON endpoints that can be used to query and update MongoDB from the Angular client-side code. Find the line `app.get('/', routes.index)` and add the following code after it:

Create JSON endpoints

```
app.get('/polls/polls', routes.list);
app.get('/polls/:id', routes.poll);
app.post('/polls', routes.create);
```

Now you need to implement these functions. Replace the contents of the routes/index.js file with this code:

routes/index.js

```
var mongoose = require('mongoose');
var db = mongoose.createConnection('localhost', 'pollsapp');
var PollSchema = require('../models/Poll.js').PollSchema;
var Poll = db.model('polls', PollSchema);
exports.index = function(req, res) {
  res.render('index', {title: 'Polls'});
};
// JSON API for list of polls
exports.list = function(req, res) {
  Poll.find({}, 'question', function(error, polls) {
    res.json(polls);
  });
};
// JSON API for getting a single poll
exports.poll = function(req, res) {
  var pollId = req.params.id;
  Poll.findById(pollId, '', { lean: true }, function(err, poll) {
    if(poll) {
      var userVoted = false,
          userChoice,
          totalVotes = 0;
      for(c in poll.choices) {
        var choice = poll.choices[c];
        for(v in choice.votes) {
          var vote = choice.votes[v];
          totalVotes++;
          if(vote.ip === (req.header('x-forwarded-for') || req.ip)) {
            userVoted = true;
            userChoice = { _id: choice._id, text: choice.text };
          }
        }
```

```
            }
            poll.userVoted = userVoted;
            poll.userChoice = userChoice;
            poll.totalVotes = totalVotes;
            res.json(poll);
          } else {
            res.json({error:true});
          }
        });
      };
      // JSON API for creating a new poll
      exports.create = function(req, res) {
        var reqBody = req.body,
            choices = reqBody.choices.filter(function(v) { return v.text != ''; }),
            pollObj = {question: reqBody.question, choices: choices};
        var poll = new Poll(pollObj);
        poll.save(function(err, doc) {
          if(err || !doc) {
            throw 'Error';
          } else {
            res.json(doc);
          }
        });
      };
```

Bind data to the front-end with Angular services

At this point, the back-end is set up to enable querying and saving of polls to the database, but we need to make some changes in Angular so it knows how to communicate with it. This is easily done using Angular services, which wrap the process of communicating with the server-side into straightforward function calls:

public/javascripts/services.js

```
      angular.module('pollServices', ['ngResource']).
        factory('Poll', function($resource) {
          return $resource('polls/:pollId', {}, {
            query: { method: 'GET', params: { pollId: 'polls' }, isArray: true }
          })
        });
```

With this file created, you need to include it in your index.jade template. Add this line below the last script element in the head section:
`script(src='/javascripts/services.js')`.

You also need to tell your Angular app to use this service module. To do this, open public/javascripts/app.js and change the first line to read as follows:
`angular.module('polls', ['pollServices'])`.

Finally, change the Angular controllers to use the service to query and store polls in the database. In the public/javascripts/controllers.js file, change the `PollListCtrl` to this:.

public/javascripts/controller.js

```
      function PollListCtrl($scope, Poll) {
        $scope.polls = Poll.query();
      }
  ...
```

Update the `PollItemCtrl` function to query for a poll by its ID:

public/javascripts/controller.js (continued)

```
...
        function PollItemCtrl($scope, $routeParams, Poll) {
          $scope.poll = Poll.get({pollId: $routeParams.pollId});
          $scope.vote = function() {};
        }
 ...
```

Similarly, change the `PollNewCtrl` function so it sends the new poll data to the server when the form is submitted.

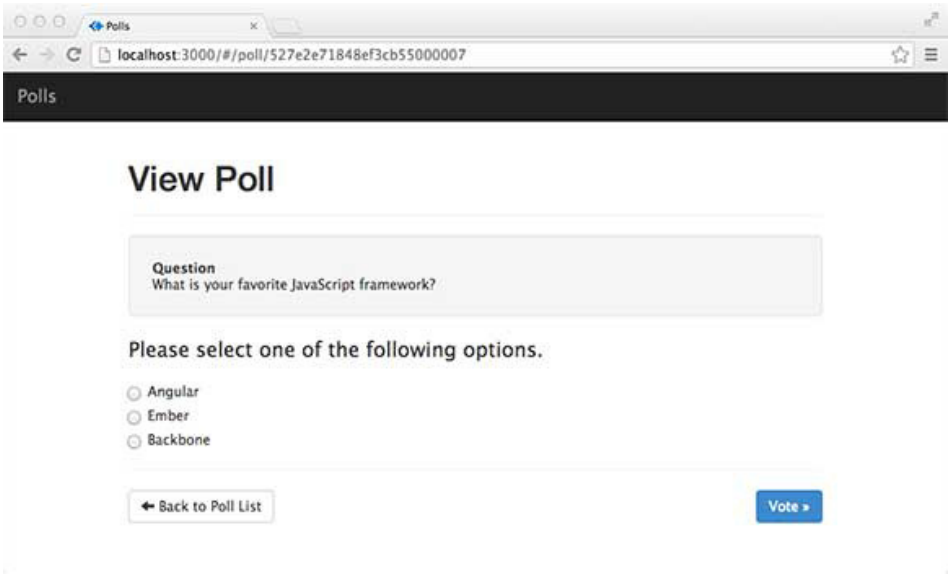
public/javascripts/controller.js (continued)

```
...
        function PollNewCtrl($scope, $location, Poll) {
          $scope.poll = {
            question: '',
            choices: [ { text: '' }, { text: '' }, { text: '' }]
          };
          $scope.addChoice = function() {
            $scope.poll.choices.push({ text: '' });
          };
          $scope.createPoll = function() {
            var poll = $scope.poll;
            if(poll.question.length > 0) {
              var choiceCount = 0;
              for(var i = 0, ln = poll.choices.length; i < ln; i++) {
                var choice = poll.choices[i];
                if(choice.text.length > 0) {
                  choiceCount++
                }
              }
              if(choiceCount > 1) {
                var newPoll = new Poll(poll);
                newPoll.$save(function(p, resp) {
                  if(!p.error) {
                    $location.path('polls');
                  } else {
                    alert('Could not create poll');
                  }
                });
              } else {
                alert('You must enter at least two choices');
              }
            } else {
              alert('You must enter a question');
            }
          };
        }
```

Run the app

You're almost there! At this point, the app should allow users to view and search polls, create new polls, and view the voting options of an individual poll. Before you run the app, make sure that you have MongoDB running locally. This is typically as simple as opening a terminal or command prompt and running the command `mongod`. Be sure to leave the terminal window open as you run your app:

Viewing a poll's choices



After you run the app, navigate to http://localhost:3000 in your browser and create a few polls. If you click on a poll, you will be able to see the choices available, but you won't be able to actually vote on the poll or see the results just yet. We'll cover that in the next and final step.

Step 4. Real-time voting with Socket.io

Web Sockets allow the server-side to directly communicate and send messages to the client-side.

The only feature left to build is the voting functionality. The app will allow users to vote, and as they do, the results will be updated in real-time on any connected clients. This is easily implemented using the socket.io npm module, so let's do that now.

Open the package.json file in the root of your app's directory, and add the following to the dependencies section:
`"socket.io": "~0.9.16"`.

Save the file, right-click in Package Explorer, and select **Run As** > **npm install** to install the npm module.

Next, open the app.js file in the app's root, and remove the `server.listen...` block right at the end of the file, replacing it with this:

app.js

```
...
        var server = http.createServer(app);
        var io = require('socket.io').listen(server);

        io.sockets.on('connection', routes.vote);

        server.listen(app.get('port'), function(){
          console.log('Express server listening on port ' + app.get('port'));
        });
```

Next, modify the index.jade template to include the socket.io client library. When you run the app, this library will automatically be made available at the location specified, so you don't need to worry about finding this file yourself. Be sure to include this just after the line where you included the angular-resource library in the template: `script(src='/socket.io/socket.io.js')`.

Finally, you need to create the vote function so that a new vote is stored when the user sends a message to socket.io, and so that a message is then emitted to all clients with the updated results. Add this to the end of the index.js file in the routes directory:

routes/index.js

```
        // Socket API for saving a vote
        exports.vote = function(socket) {
          socket.on('send:vote', function(data) {
            var ip = socket.handshake.headers['x-forwarded-for'] ||
socket.handshake.address.address;
            Poll.findById(data.poll_id, function(err, poll) {
              var choice = poll.choices.id(data.choice);
              choice.votes.push({ ip: ip });
              poll.save(function(err, doc) {
                var theDoc = {
                  question: doc.question, _id: doc._id, choices: doc.choices,
                  userVoted: false, totalVotes: 0
                };
                for(var i = 0, ln = doc.choices.length; i < ln; i++) {
                  var choice = doc.choices[i];
                  for(var j = 0, jLn = choice.votes.length; j < jLn; j++) {
                    var vote = choice.votes[j];
                    theDoc.totalVotes++;
                    theDoc.ip = ip;
                    if(vote.ip === ip) {
                      theDoc.userVoted = true;
                      theDoc.userChoice = { _id: choice._id, text: choice.text };
                    }
                  }
                }
                socket.emit('myvote', theDoc);
                socket.broadcast.emit('vote', theDoc);
              });
            });
          });
        };
```

Note: If you're wondering why the app looks for the header `'x-forwarded-for'` before the regular IP address property, this is to ensure that the correct client IP is used when the app is deployed in a load-balanced environment. If you deploy the app to Bluemix or Cloud Foundry, for example, this is critical for the app to function correctly.

Add an Angular service for sending data to Web sockets

The back-end functionality for Web Sockets is now complete. All that's left is to tie up the front-end to send and listen for socket events. The best approach to doing this is to add a new Angular service. Replace the contents of the services.js file in the public/javascripts folder with this code:

public/javascripts/services.js

```
        angular.module('pollServices', ['ngResource']).
          factory('Poll', function($resource) {
```

```
            return $resource('polls/:pollId', {}, {
              query: { method: 'GET', params: { pollId: 'polls' }, isArray: true }
            })
          }).
          factory('socket', function($rootScope) {
            var socket = io.connect();
            return {
              on: function (eventName, callback) {
                socket.on(eventName, function () {
                  var args = arguments;
                  $rootScope.$apply(function () {
                    callback.apply(socket, args);
                  });
                });
              },
              emit: function (eventName, data, callback) {
                socket.emit(eventName, data, function () {
                  var args = arguments;
                  $rootScope.$apply(function () {
                    if (callback) {
                      callback.apply(socket, args);
                    }
                  });
                })
              }
            };
          });
```

Finally, you need to edit the `PollItemCtrl` controller so that it will listen for and emit Web Socket messages for votes. Replace the original controller with this:

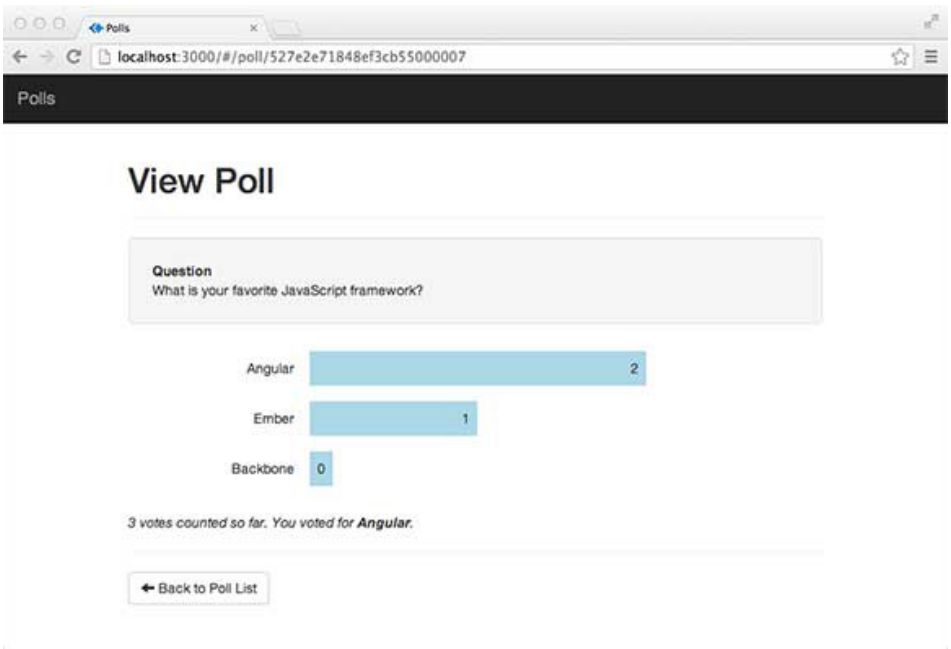
public/javascripts/controllers.js

```
...
        function PollItemCtrl($scope, $routeParams, socket, Poll) {
          $scope.poll = Poll.get({pollId: $routeParams.pollId});
          socket.on('myvote', function(data) {
            console.dir(data);
            if(data._id === $routeParams.pollId) {
              $scope.poll = data;
            }
          });
          socket.on('vote', function(data) {
            console.dir(data);
            if(data._id === $routeParams.pollId) {
              $scope.poll.choices = data.choices;
              $scope.poll.totalVotes = data.totalVotes;
            }
          });
          $scope.vote = function() {
            var pollId = $scope.poll._id,
                choiceId = $scope.poll.userVote;
            if(choiceId) {
              var voteObj = { poll_id: pollId, choice: choiceId };
              socket.emit('send:vote', voteObj);
            } else {
              alert('You must select an option to vote for');
            }
          };
        }
    ...
```

View the final product in action

The polls app is now complete. Make sure that mongod is still running, and run the Node application again in Eclipse. Point your browser at http://localhost:3000, and navigate to a poll and vote. You should see the result. To see the real-time updates, find your local IP address and replace `localhost` with it. Then navigate to it using a different machine or even a smartphone or tablet device on your local network. When you vote on another device, the result will be shown there, and the result will also be pushed out to your main computer's browser automatically:

Viewing poll results



Next steps: further development and deployment

The polls app you have just created is a decent starting point, but there is much that can be improved. When planning apps like this, I like to follow an agile methodology of defining user stories and epics, and breaking the project into sprints. I used DevOps Services for this project, which makes development really simple by keeping all of your project collateral and source code together in a cloud-hosted repository.

When you are satisfied with your app, the next step is to share it with the world. In the past, deploying even a simple application could be a bit of a nightmare, but thankfully those days are over. Using IBM's Cloud Foundry-compatible Bluemix platform, you can deploy your applications to the cloud in minutes with minimal configuration and even less fuss.

Conclusion

It's a wonderful time to be a developer. We have an abundance of frameworks and tools at our disposal that make developing great apps not only simpler and faster, but also more enjoyable.

In this article, you learned how to build an app using what is being referred to as the MEAN stack (Mongo, Express, Angular, Node). This stack may one day surpass the LAMP stack (Linux, Apache, MySQL, PHP) that is synonymous with simplicity when it comes to Web application development and deployment. I, for one, can't wait.

Related topics

- Node.js — beyond the basics
- An introduction to MongoDB
- Project kickoff on DevOps Services
- The SDK for Node.js runtime
- Node.js
- MongoDB
- JavaScript